Advanced Software Design: Expense Tracker Application

Introduction

Our project aims to develop an expense tracker and budgeting application utilising common software architectures and heuristics. The objective is to create a functional software product while emphasising efficiency, communication, teamwork, and effective management throughout the development process.

Objectives:

- Implement common software architectures such as factories, bridges, singletons, and interfaces effectively. Maintain code readability, modularity, and efficiency through rigorous refactoring, ensuring that the codebase remains scalable and easy to maintain.
- Achieve all requirements specified in the project specifications, meeting both functional and non-functional requirements to deliver a comprehensive solution.
- Create an aesthetically pleasing design with interesting architectural and visual elements. This includes intuitive user interfaces, visually appealing graphics, and a cohesive design language throughout the application.Ensure user-friendliness through intuitive navigation using direction keys and selection menus, prioritising ease of use for all users.

Team Collaboration: The following roles were delegated to team members to ensure efficient task management and collaboration:

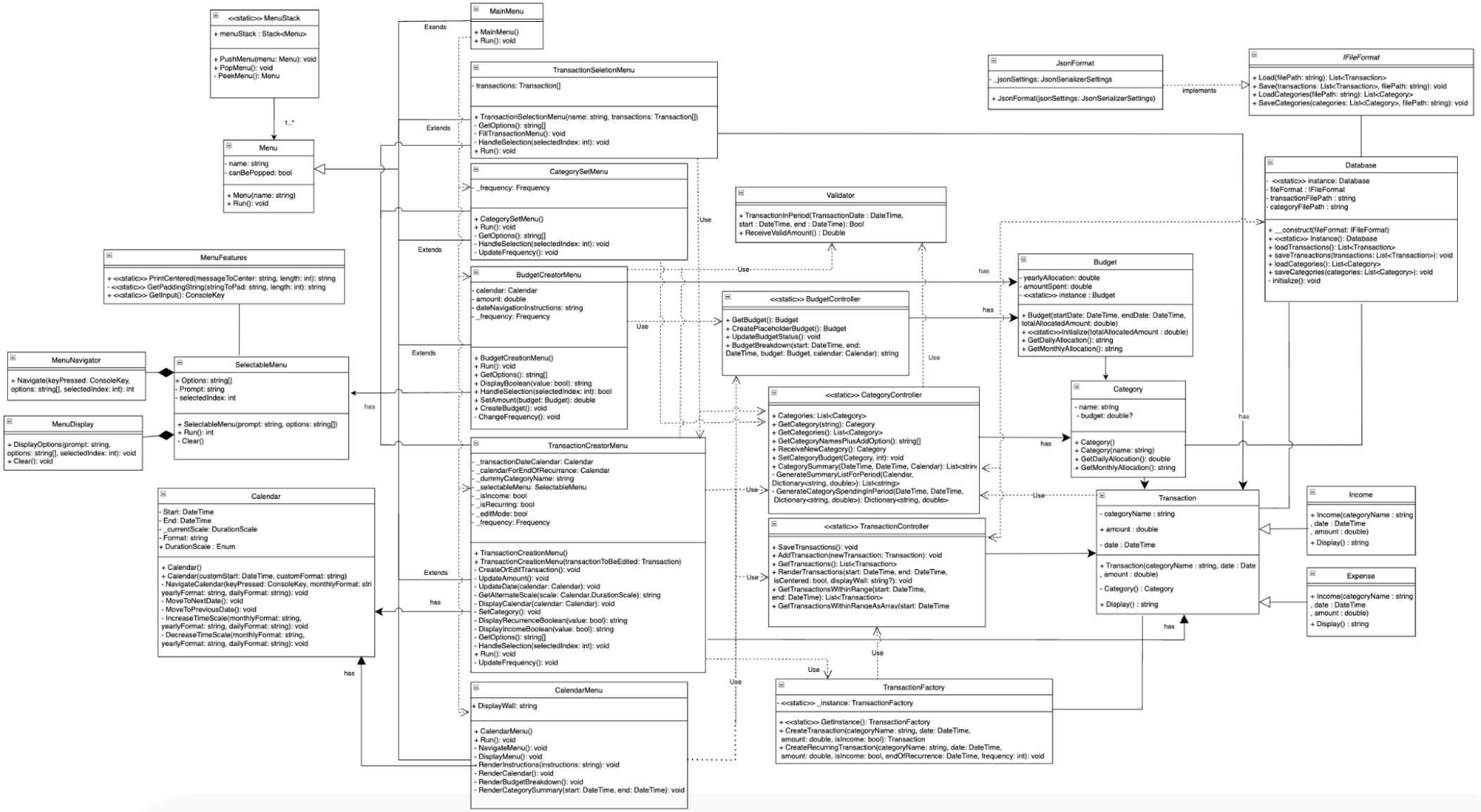
Ben Hogg - *Lead Designer & Developer*
Ben MacKellar - *Lead Developer & Designer*
Brad Prosser - *Project Manager & Tester*

Each member contributed to various aspects of the project, leveraging their expertise to achieve our collective goals. The team would meet up at least once weekly for an in-person team meeting to discuss progress against previous goals and set future targets.

Expected Outcome: The project aims to deliver a fully functional expense tracking application that meets all specified requirements while adhering to high standards of design, flexibility, usability, and potential expandability for future enhancements.

Blog Link: https://7seng003wbbb.blogspot.com/

UML


<<static>> MenuStack
+ menuStack : Stack<Menu>
+ PushMenu(menu: Menu): void
+ PopMenu(): void
- PeekMenu(): Menu
1..*
Menu
- name: string
- canBePopped: bool
+ Menu(name: string)
+ Run(): void
MainMenu
+ MainMenu()
+ Run(): void
Exends
TransactionSeletionMenu
- transactions: Transaction[]
+ TransactionSelectionMenu(name: string, transactions: Transaction[])
- GetOptions(): string[]
- FillTransactionMenu(): void
- HandleSelection(selectedIndex: int): void
+ Run(): void
Extends
CategorySetMenu
- _frequency: Frequency
+ CategorySetMenu()
+ Run(): void
- GetOptions(): string[]
- HandleSelection(selectedIndex: int): void
- UpdateFrequency(): void
Extends
BudgetCreatorMenu
- calendar: Calendar
- amount: double
- dateNavigationInstructions: string
- _frequency: Frequency
+ BudgetCreationMenu()
+ Run(): void
+ GetOptions(): string[]
+ DisplayBoolean(value: bool): string
+ HandleSelection(selectedIndex: int): bool
+ SetAmount(budget: Budget): double
+ CreateBudget(): void
- ChangeFrequency(): void
Extends
TransactionCreatorMenu
- _transactionDateCalendar: Calendar
- _calendarForEndOfRecurrance: Calendar
- _dummyCategoryName: string
- _selectableMenu: SelectableMenu
- _isIncome: bool
- _isRecurring: bool
- _editMode: bool
- _frequency: Frequency
+ TransactionCreationMenu()
+ TransactionCreationMenu(transactionToBeEdited: Transaction)
- CreateOrEditTransaction(): void
- UpdateAmount(): void
- UpdateDate(calendar: Calendar): void
- GetAlternateScale(scale: Calendar.DurationScale): string
- DisplayCalendar(calendar: Calendar): void
- SetCategory(): void
- DisplayRecurrenceBoolean(value: bool): string
- DisplayIncomeBoolean(value: bool): string
- GetOptions(): string[]
- HandleSelection(selectedIndex: int): void
+ Run(): void
- UpdateFrequency(): void
Extends
CalendarMenu
+ DisplayWall: string
+ CalendarMenu()
+ Run(): void
- NavigateMenu(): void
- DisplayMenu(): void
RenderInstructions(instructions: string): void
- RenderCalendar(): void
- RenderBudgetBreakdown(): void
- RenderCategorySummary(start: DateTime, end: DateTime): void
MenuFeatures
+ <<static>> PrintCentered(messageToCenter: string, length: int): string
- <<static>> GetPaddingString(stringToPad: string, length: int): string
+ <<static>> GetInput(): ConsoleKey
MenuNavigator
+ Navigate(keyPressed: ConsoleKey, options: string[], selectedIndex: int): int
MenuDisplay
+ DisplayOptions(prompt: string, options: string[], selectedIndex: int): void
+ Clear(): void
SelectableMenu
+ Options: string[]
- Prompt: string
- selectedIndex: int
+ SelectableMenu(prompt: string, options: string[])
+ Run(): int
- Clear()
has
Calendar
- Start: DateTime
- End: DateTime
- _currentScale: DurationScale
- Format: string
+ DurationScale : Enum
+ Calendar()
+ Calendar(customStart: DateTime, customFormat: string)
- NavigateCalendar(keyPressed: ConsoleKey, monthlyFormat: string, yearlyFormat: string, dailyFormat: string): void
- MoveToNextDate(): void
- MoveToPreviousDate(): void
- IncreaseTimeScale(monthlyFormat: string, yearlyFormat: string, dailyFormat: string): void
- DecreaseTimeScale(monthlyFormat: string, yearlyFormat: string, dailyFormat: string): void
has
Validator
+ TransactionInPeriod(TransactionDate : DateTime, start : DateTime, end : DateTime): Bool
+ ReceiveValidAmount() : Double
Use
<<static>> BudgetController
+ GetBudget(): Budget
+ CreatePlaceholderBudget(): Budget
+ UpdateBudgetStatus(): void
+ BudgetBreakdown(start: DateTime, end: DateTime, budget: Budget, calendar: Calendar): string
Use
<<static>> CategoryController
+ Categories: List<Category>
+ GetCategory(string): Category
+ GetCategories(): List<Category>
+ GetCategoryNamesPlusAddOption(): string[]
+ ReceiveNewCategory(): Category
+ SetCategoryBudget(Category, int): void
+ CategorySummary(DateTime, DateTime, Calendar): List<strin
- GenerateSummaryListForPeriod(Calendar, Dictionary<string, double>): List<string>
- GenerateCategorySpendingInPeriod(DateTime, DateTime, Dictionary<string, double>): Dictionary<string, double>
Use
<<static>> TransactionController
+ SaveTransactions(): void
+ AddTransaction(newTransaction: Transaction): void
+ GetTransactions(): List<Transaction>
+ RenderTransactions(start: DateTime, end: DateTime, isCentered: bool, displayWall: string?): void
+ GetTransactionsWithinRange(start: DateTime, end: DateTime): List<Transaction>
+ GetTransactionsWithinRangeAsArray(start: DateTime
Use
TransactionFactory
- <<static>> _instance: TransactionFactory
+ <<static>> GetInstance(): TransactionFactory
+ CreateTransaction(categoryName: string, date: DateTime, amount: double, isIncome: bool): Transaction
+ CreateRecurringTransaction(categoryName: string, date: DateTime, amount: double, isIncome: bool, endOfRecurrence: DateTime, frequency: int): void
Use
JsonFormat
- _jsonSettings: JsonSerializerSettings
+ JsonFormat(jsonSettings: JsonSerializerSettings)
implements
IFileFormat
+ Load(filePath: string): List<Transaction>
+ Save(transactions: List<Transaction>, filePath: string): void
+ LoadCategories(filePath: string): List<Category>
+ SaveCategories(categories: List<Category>, filePath: string): void
Database
- <<static>> instance: Database
- fileFormat : IFileFormat
- transactionFilePath : string
- categoryFilePath : string
+ __construct(fileFormat: IFileFormat)
+ <<static>> Instance(): Database
+ loadTransactions(): List<Transaction>
+ saveTransactions(transactions: List<Transaction>): void
+ loadCategories(): List<Category>
+ saveCategories(categories: List<Category>): void
- initialize(): void
has
Budget
- yearlyAllocation: double
- amountSpent: double
- <<static>> instance : Budget
+ Budget(startDate: DateTime, endDate: DateTime, totalAllocatedAmount: double)
+ <<static>>Initialize(totalAllocatedAmount : double)
+ GetDailyAllocation(): string
+ GetMonthlyAllocation(): string
has
Category
- name: string
- budget: double?
+ Category()
+ Category(name: string)
+ GetDailyAllocation(): double
+ GetMonthlyAllocation(): string
has
Transaction
- categoryName : string
+ amount : double
- date : DateTime
+ Transaction(categoryName : string, date : Date , amount : double)
- Category() : Category
+ Display() : string
has
Income
+ Income(categoryName : string , date : DateTime , amount : double)
+ Display() : string
Expense
+ Income(categoryName : string , date : DateTime , amount : double)
+ Display() : string

Design, Architecture, and Patterns - Ben Hogg

MVC Core and the Flow of Information

The fundamental principle behind the design was to follow a pattern that closely mimics the widely used Model-View-Controller pattern. Whilst typically used in larger scale web based applications, this structure provides a core framework to work from whilst promoting clear responsibilities that separate concerns, with distinct layers for data structures (models), menus (views) and logic connecting the two (controller).

Whilst in typical MVC systems data is stored in an external database, the controllers implemented here act as model repositories, storing the model objects as they're created. For the purposes of this assessment, JSON files storing the data are used to save/load data during runtime in order to mimic the use of a database. This approach is suitable for the purposes of this assessment/small scale apps where integration of a full database is not necessary. Whilst simple, the use of flat files to demonstrate persistence is considered appropriate within the scope.

Generally, the flow of information in the application follows some, if not all of the following steps.

User Action: User initiates an action through the application interface, such as adding a transaction or selecting a budget category.

Controller Interaction: The controller associated with the current view receives the user's action and processes it accordingly. Based on the action, the controller will perform some task, such as storing a model as an instantiated object or fetching and processing data from its repository of objects. For example, when adding a transaction, the TransactionController manages the incoming transaction object from the TransactionFactory and stores it in the transaction repository.

Model Interaction: Models act as blueprints for the most important and intrinsic objects in the program. They include the Budget class, Category class, and the Transaction class - and it's derived types, income and expense Models encapsulate the applications most important data structures and provide a solid foundation upon which to build an OOP system.

Data Persistence: The application utilises a database class and file format interface to manage data persistence. Controllers interact with the database to save or load transaction and category

data, ensuring that information is preserved across application sessions. Whenever a transaction is added to the repository, the database is saved. Whenever the application is executed, the database is loaded into the appropriate repositories.

View Rendering: Once data processing is complete, controllers update the view to reflect changes or present relevant information to the user. Views are responsible for rendering data in a clear and understandable format, utilising menus, text displays, and graphical elements as needed.

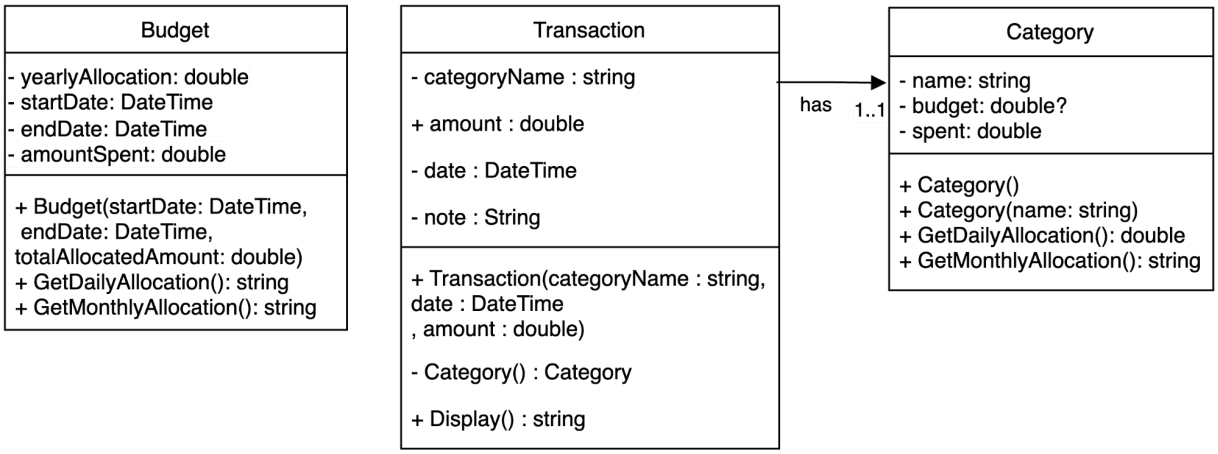
User Feedback: The updated view is displayed to the user, providing feedback on the action performed and presenting relevant information. Users can then continue to interact with the application, initiating new actions or navigating to different views as needed.

By following this structured flow of information, MVC forms the core of our application's functionality, enabling data handling and user interaction.

Models

Modelling Transaction, Budget and Category appropriately, with controllers handling the flow of data between models allows the models to serve as well abstracted/encapsulated data structures for clear readable organisation of the project. This design route was taken to allow models to exist without necessarily being tightly coupled. Adopting this approach allowed for a clear separation of concerns (of which it’s benefits are later described), allowing models to encapsulate data specific to its role within the application. Not overlapping concerns within unclear/overcomplicated data structures was chosen to promote readability whilst encouraging cleaner/easier implementation of the design later in the project. Strong models also helps the view architecture more easily display information via the controllers, for example transactions within a certain time frame/category can be easily fetched due to the relevant attributes assigned to each model.

It is important to note in this design that a user may choose to set an overall budget, via instantiation of a Budget object (singleton). It is not mandatory for a user to set an overall budget should they not wish. This optionality is also applied to categories. Each Transaction object has a Category associated with it, but the transaction amount will not be compared against a spending for that category should the user not wish to set a category budget for food, for example. This design provides the user with more features/flexibility in how they wish to track their spending within the app. To enable this functionality, the models are designed as follows:



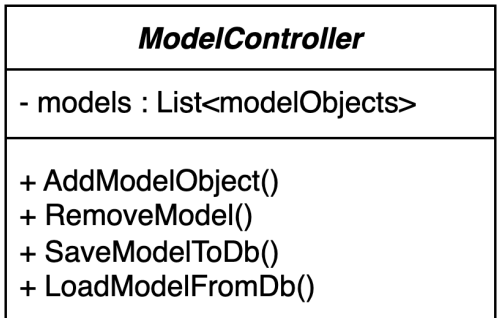
Controllers

Controllers act as the intermediary between the view (menu's) and the models (data structures). Having a designated controller for handling the actions relevant to each controller, e.g. TransactionController that updates, edits, retrieves transactions allows for a consistent, centralised approach that can be taken anytime manipulation/the flow of data is concerned. Abstracting any of this responsibility away from the data structures themselves, the models, separates concerns and allows controllers to provide data to the views without calling upon methods within the models themselves.

Importantly these controllers act as centralised repositories for storing and managing objects of the models.

The following design approach was taken with each controller:

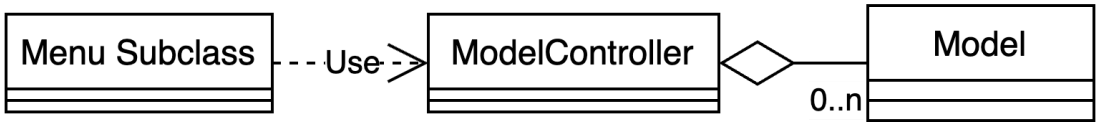
1) Contain/store a list of the model object in which it is controlling (i.e. TransactionController storing the list of transactions).
2) Render/retrieve information relevant to each model for the UI, e.g. providing a list of transactions with a certain date
3) Work with other controllers when one models awareness of another calls for data to be updated (e.g. A category of Food will need to be aware of when a Transaction of category food is added/removed). This is performed at runtime via methods that render the current state of transactions.
4) Responsible for working with the database class to save/ retrieve relevant model data from the database after it has changed the state of any particular model object.



Views

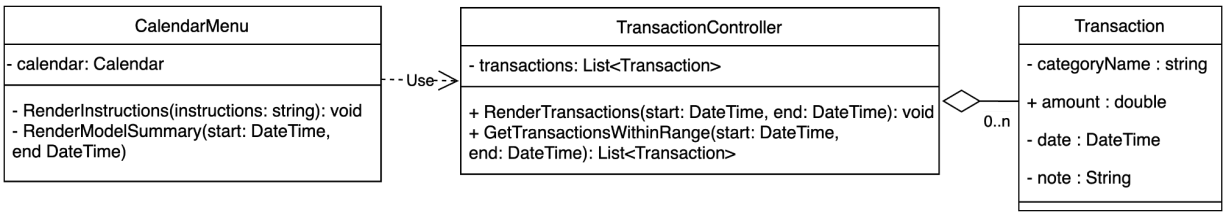
To provide a comprehensive view/UI experience for the user it was decided that a menu based system would be best to assign responsibilities to different views within the system. For example, when adding a transaction or viewing a transaction, a different UI is presented for the user to interact with. This structure allows succinct and readable navigation throughout the system, allowing the user to find features easily. Breaking up the UI into multiple menus reduces clutter for the user and provides a familiar interface in which to interact.

In console based applications, a way of rendering multiple views is by storing objects derived from a Menu base class in a stack structure, this allows for the Menu at the peak to be displayed to the user, whilst utilising the Pop and Push features of a stack in order to navigate between Menus. To achieve this design a class, MenuStack, is responsible for Popping and Pushing menus as appropriate, with all subsequent menus sitting on a MainMenu (the menu shown at the beginning of the program). An important feature here is that to be operated on within the stack, all menus that are navigated to and from must inherit from a Menu base class (See LSP).



A priority in budget/transaction tracking apps focus on providing the user with a view that encompasses their current ingoing/outgoings for a given, selectable period of time e.g. a summary of recent transactions, with their associated category, within that day, or week.

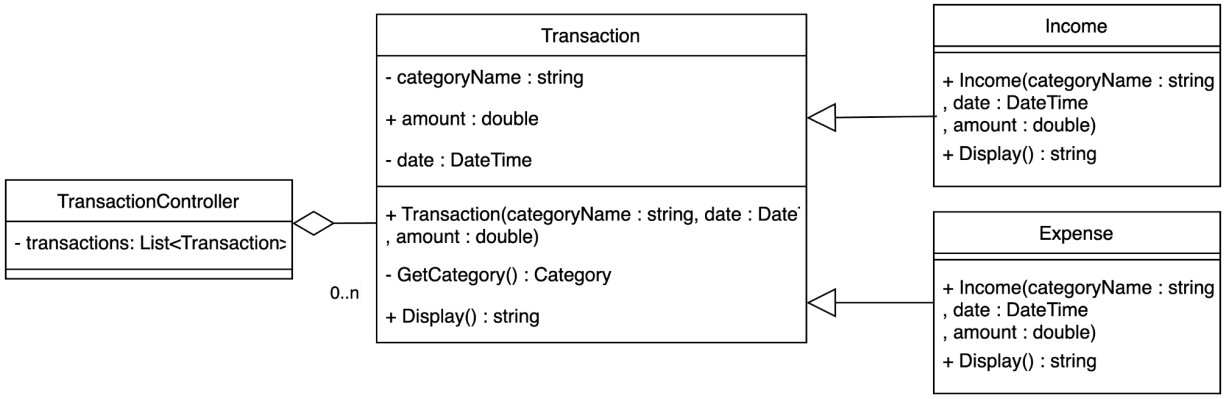
In line with industry norms, a design incorporating a calendarMenu has been chosen that allows for easy user selection of the timeframe displayed via keypresses. This menu aims to offer a complete summary of transactions (including their type, expense or income), spending vs budget comparisons as well as category specific comparisons. Consideration was taken on this decision centred around the extensive implementation required of this design compared to a more static approach, ultimately it was decided an app closely visually resembling those seen on the market would be a worthwhile decision.



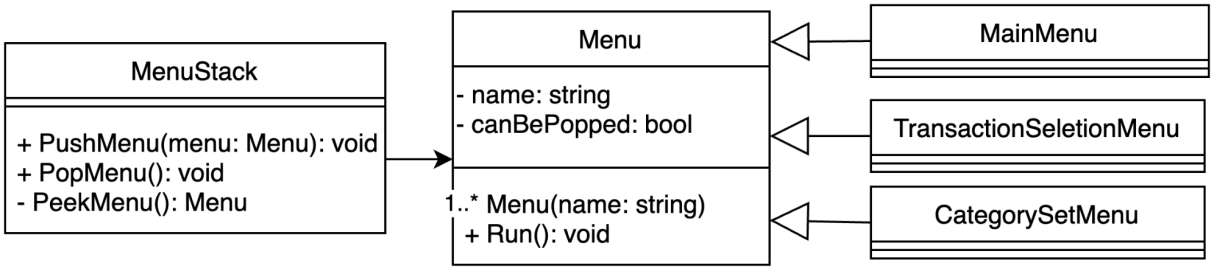
This diagram describes a high level overview (n.b. This diagram is for demonstration purposes and detailed class relationships can be seen in the full UML) of the design, with an example given for the Transaction model. As the user navigates through the menu to select the timeframe they wish to see, the Calendar will call the render methods implemented in the appropriate controller to display summaries relevant to the timeframe the user currently has selected.

Liskov Substitution Principle

Transactions exist in two general forms, an expense and an income, designing Income and Expense classes to inherit from a Transaction base class allows the TransactionController to manage and store derived transaction objects as each object can be used interchangeably, adhering to the LSP principle. This simplifies the operations and allows all Transactions to be managed by one static entity, e.g. adding or deleting Transactions regardless of their derived type.



This approach has also been taken with the MenuStack, in which its responsibility is simply popping and pushing Menu type objects from its stack. This provides the benefit of allowing a range of Menu types to be managed centrally and in a unified manner. This approach also accommodates extension/flexibility, should the program expand and require additional menus to be integrated.



Singletons

Singletons are effectively utilised in the program for specific objects such as the database and budget classes. These classes are designed to have a single instance throughout the application's execution. By employing singletons, the program ensures that only one instance of these classes exists through a single point of access, preventing potential issues that may arise if developers or users somehow created multiple instances. This streamlined approach enhances code clarity and simplifies resource management, contributing to an easier and less error prone implementation and execution of the code base.

Bridge

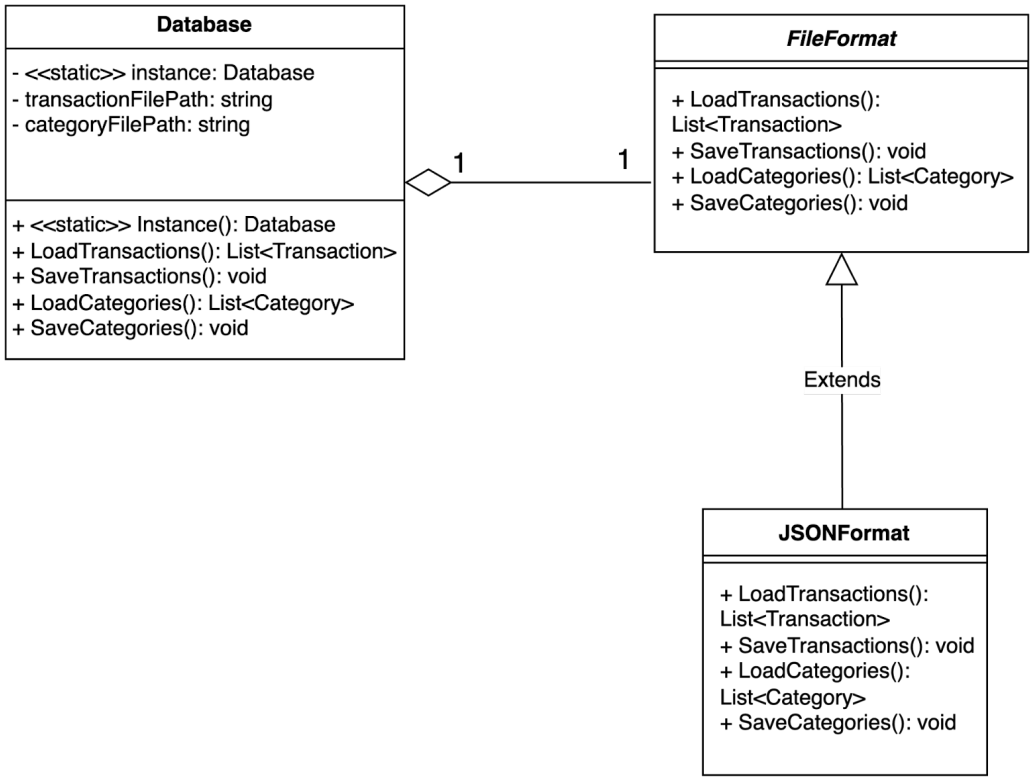
The database is designed to utilise the bridge pattern to manage saving and loading operations from JSON files - a subclass of FileType, thereby ensuring flexibility for potential expansion to other file formats such as .txt or .xml. The bridge pattern allows for this flexibility by decoupling the database class from the specifics of file handling, allowing for a smooth interchangeable interaction with different file types.

In this design, the bridge has two components:

1. Abstraction: The abstraction, represented in the Database, defines saving and loading operations without being tied to any specific file format. It contains references to the methods for saving and loading, which are delegated to the actual implementations across the bridge within classes implementing the IFileFormat interface.

2. Implementer: The implementer, represented by the IFileFormat interface defines the file handling operations such as saving and loading. It is a common interface for different file types that provide the concrete implementations.

By using the bridge pattern, the database class can interact with different file types through the implementers interface without being aware of the specific implementations. This allows for easy swapping of file types and makes the addition of new file formats with little impact on the existing code.



Factory

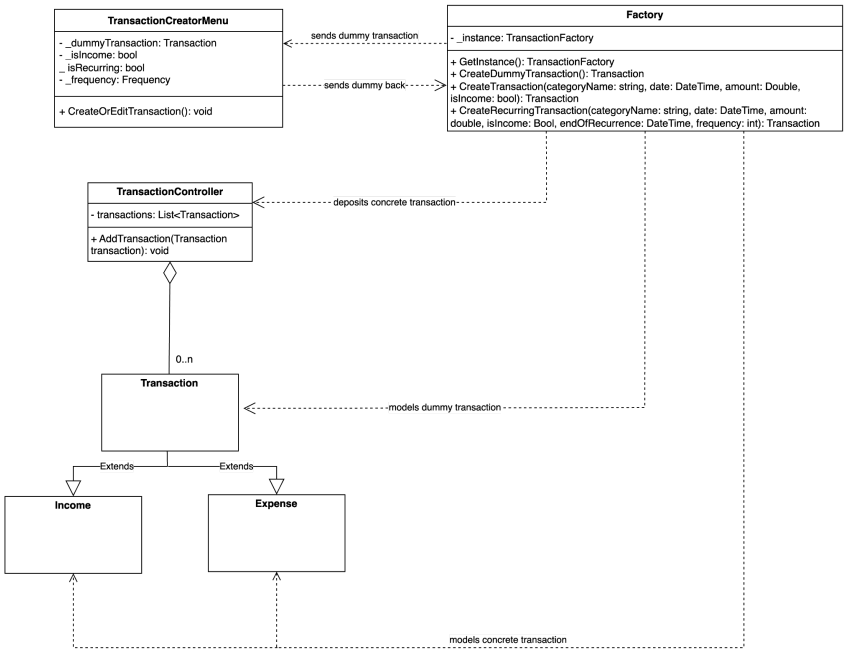
With Expense and Income objects being inherited from the Transaction subclass, designing a Transaction factory allows great flexibility in creating objects of different subclasses based on conditions provided or received from the UI. This allows one factory to create Transaction subclass objects flexibly based on the users requirements.

The design of this class is not a traditional abstract or concrete factory, but it incorporates the fundamental aspects of the Factory pattern and adapts them to the specific requirements of the application. The main goal of this design is to separate the creation logic of Transaction objects from other parts of the program, promoting flexibility and separation of concerns.

The TransactionFactory class follows a singleton pattern, ensuring that only one instance of the factory exists throughout the application.

The factory receives a dummy model of a transaction from the user input, and based on this model, it creates a new concrete object of either Expense or Income type. This approach prevents the need to cast the transaction into a reference type that it doesn't belong to and hides the subclasses' true type.

In addition to creating single transactions, the factory also has the ability to create recurring transactions. This is done by calculating the total number of transactions based on the frequency and duration of the recurrence, then creating individual transactions for each occurrence.c



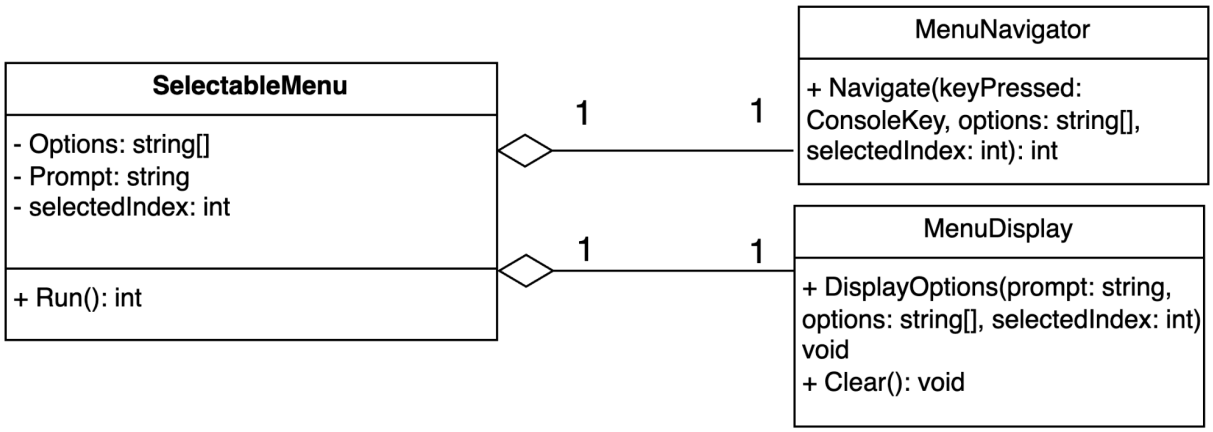
Composition

Composition is a powerful design pattern that promotes reusability and separation of responsibilities between different classes. This pattern makes clean, extendable code.

In our program, we adopted this design pattern and applied it to a selectablemenu class. This menu would allow developers to easily create a menu object that can respond to arrow keys, option selection, and display a dynamic menu.

Composition allowed us to break down the functionality of the selectablemenu into two manageable components. Each component is responsible for a specific aspect of the menu functionality, making the code very easy to understand and work with. Composition helps separate the concerns into navigation logic and display logic. This made the code very easy to maintain and test. If there is an issue with the display, it's very easy to know where that problem likely lies. It also makes the selectablemenu class very easy to modify without affecting the

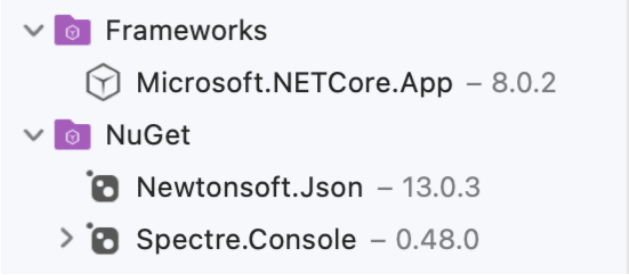
overall functionality of the menu. For example, if we needed to change the display style, it's easy to modify the MenuDisplay class without affecting the rest of the codebase. Furthermore, this also encapsulates the functionality of each component within its own class, reducing coupling between different parts of the system.



Implementation Section - Ben MacKellar

Approach, Tools and Technologies

This design was implemented using C#, .NETCore 8, utilising a powerful language that supports the use of OOP principles, leveraging the benefits of encapsulation, inheritance and code organisation. Using C# we are able to promote code reuse and generally apply a hierarchical structure to the system, making it robust and modular and in some aspects scalable should the scope of the project increase. NewtonSoft and Spectre packages were also used to provide JSON Serialization and ASCII text formatting capabilities repsectively.



Whilst a lead developer initially implemented the conversion of initial design to a codebase, and took on refactoring responsibilities, a collaborative approach was taken for the implementation, allowing for knowledge sharing and collaborative problem solving.

Version control was maintained through the use of a Git Repository to ensure team members could implement and test changes prior to deployment into the central repository (main). This version control also allowed for previous versions to be visited should a bug or error be carried through to the central repository inadvertently.

This implementation section will focus on a few key implementations that effectively summarise how the flow of data/UI is achieved.

Code Implementation

Models

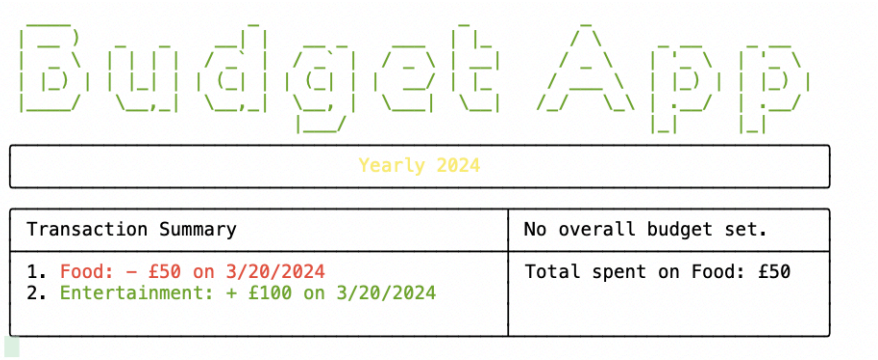
Implementing the models ultimately was a straightforward task provided the design. Initially during the design phase when the budget, transaction and categories were more tightly coupled. For example, a design including the budget/category storing a list of transactions was discussed, adding complexity to the models, however this responsibility was abstracted away to be managed by the appropriate controllers.

With the model controllers storing the model objects as lists of objects, it means the interaction between different model objects themselves was at a minimum (i.e budget not aware of transactions, or categories directly). As discussed later, the controllers are responsible for drawing together the insights and summaries from the list of data they store. More specifically for example, a summary about spending in a certain category is rendered at runtime by collating all the transactions under that category type within a given period, and is recalculated dynamically as the user adds/removes transactions or changes the time frame, without the category objects themselves having an awareness of any transaction derived objects. To implement an MVC style architecture, the models here are primarily data structures for encapsulating behaviours and relationships representing real world entities. The budget model is described in further detail as an example of model implementation.

Polymorphism is exhibited in the Transaction derived classes, Expense and Income, allowing each method to implement its own display method, creating unique displays within the list of transactions within the application. Additionally, deriving from the Transaction base class enables straightforward implementation of the Liskov Substitution Principle set out in the design, allowing the Transaction Controller to centrally manage all Transaction objects, irrespective of derived types.

```
public class Expense : Transaction
{
    public Expense(string categoryName, DateTime month, double amount) : base(categoryName, month, amount)
    {
    }

    public override string Display => $"[red]{Category.name}: - £{Amount} on {Date.ToShortDateString()}[/]";
}
```

The different display method implementation for Expense and Income objects can be seen below, displayed in the Calendar Menu of the application.



Singleton and Model Case: Budget.cs

The Budget model, designed as a Singleton, acts as the overall budget for the system, offering a central repository for tracking spending over a yearly budget allocation, regardless of individual category budgets.

The model implements dynamic scaling methods to allow the user to view their budget spending across their selected timeframe (displayed within the dynamic/adjustable calendar menu), allowing the budget to be more readable/applied to different timeframes. This implementation tackles the inherent inflexibility of a single budget instance, and allows for a greatly flexible UI interface mimicking a real world budget application, and is considered to outweigh this limitation of a singleton in light of the project's scope.

```
public double GetDailyAllocation() => Math.Round((yearlyAllocation / 365), 2);
public double GetMonthlyAllocation() => Math.Round((yearlyAllocation / 12), 2);
```

Clear, expression bodied methods coupled with the C# Math class offers clean, concise readable code that displays the double values in a currency format. These methods however assume a year consists of 12 even sized months and is an improvement that could be implemented in a project with a larger scope.

An instance of the Budget object can only be initialised through the Initialize method, which in turn calls the constructor of the class, which is only allowed to be lazily initialised if a budget object doesn't already exist. This implementation efficiently provides a Singleton design pattern as designed complete with exceptions.

```
private static Budget _instance;

public static Budget Instance
{
    get
    {
        if (_instance == null)
        {
            throw new Exception("Budget instance has not been initialized.");
        }
        return _instance;
    }
}

public static void Initialize(DateTime startDate, DateTime endDate, double totalAllocatedAmount)
{
    if (_instance != null)
    {
        throw new Exception("Budget instance has already been initialized.");
    }
    _instance = new Budget(startDate, endDate, totalAllocatedAmount);
}
```

A drawback of this implementation, and in part made more difficult by the singleton design is the ability to set one budget. Having a budget instance that tracks spending across the year via a controller demonstrates strong functionality, however makes implementing a budget for the following year a more challenging task. Alternatively, had the Budget not been a singleton, multiple budgets would need to be managed and instantiated. Whilst more intensive implementation, it would allow budgets to be set over more granular, flexible periods of time, whilst also allowing budgets to be set for the following calendar year.
Implementing multiple ‘budgets’ or periods of tracked expenditure in a Singleton design requires more nuisance.

Controller Case: TransactionController.cs

As described initially, the controllers play a key role in fetching and supplying information to the menus based on the user's selection, storing model objects through aggregation, and maintaining data integrity through closely working with the database instance.

Each controller stores an instance or list (utilised due to its dynamism) of the model pertaining to it, effectively also functioning as a repository for model data. Firstly in the context of the TransactionController, a private list stores all transactions, and is also populated at runtime via loading existing transactions from the database instance, utilising the persistence functionality the database offers whilst not needing to know the concrete implementations of this class. GetTransactions handles the loading process transparently by invoking the LoadTransactions method if the transactions list is initially null, ensuring this list is always populated with the latest transactions. Controllers calling these methods anytime a Model object is added/removed/edited

ensure data is managed effectively and consistently, maintaining data integrity and persistence between sessions.

```
public static class TransactionController
{
    private static List<Transaction> transactions;

    private static void LoadTransactions()
    {
        transactions = Database.Instance.LoadTransactions();
    }

    public static List<Transaction> GetTransactions()
    {
        if (transactions == null)
        {
            LoadTransactions();
        }
        return transactions;
    }
```

Controllers are responsible for adding and removing the models they control/store to their repository of models (in the case of Transaction and Category Controllers. This requires implementing logic based on how each model when added/removed affects another. For example, as shown below, a new transaction will contribute towards a budget's spending, only if a budget is set (and the transaction is also of type Expense). Anytime an object is removed or added from a controller, the database instance is called to ensure an up to date persistence of data.

```
public static void AddTransaction(Transaction newTransaction)
{
    if (newTransaction.Category == null)
    {
        throw new ArgumentException("Transaction must have a valid category.");
    }

    transactions.Add(newTransaction);

    if (BudgetController.GetBudget() != null)
    {
        BudgetController.UpdateBudgetStatus();
    }
    SaveTransactions();
}

public static void DeleteTransaction(Transaction transaction)
{
    transactions.Remove(transaction);

    if (BudgetController.GetBudget() != null)
    {
        BudgetController.UpdateBudgetStatus();
    }

    SaveTransactions();
}
```

To fulfil the key design of the calendar based menu, summary methods were implemented into the controllers. The methods receive two DateTime objects (representing the start and end of the period to be summarised) which subsequently return a string, summarising the status of the models within the controller. The calendar menu can then provide the user selected timeframe as Start and End frame DateTime objects into the summary methods to render a summary for the selected period. As a result, these methods can be dynamically utilised as the user navigates (changes dates) through the calendar menu.

```
public static string TransactionSummary(DateTime start, DateTime end)
{
    List<Transaction> list = GetTransactionsWithinRange(start, end);
    StringBuilder sb = new StringBuilder();
    int i = 1;
    foreach (var transaction in list)
    {
        string transactionDisplay = $"{i}. {transaction.Display}";
        sb.AppendLine(transactionDisplay);
        i++;
    }
    if (list.Count == 0)
    {
        sb.AppendLine("No transactions for this period");
    }
    return sb.ToString();
}
```

C# classes such as StringBuilder are utilised to facilitate the handling and generation of large strings. The process of retrieving the transactions within this period is encapsulated into a GetTransactionsWithinRange method. This method returns a list of transactions regardless of their derived type (Expense or Income). This allows the virtual Display method (transaction.Display) to be called, without the type of object being known until runtime, increasing the flexibility of the method's ability to summarise. Additionally, the LINQ library is used to promote readability.

```
public static List<Transaction> GetTransactionsWithinRange(DateTime start, DateTime end) =>
  transactions.Where(c => c.Date >= start && c.Date <= end).ToList();
```

Whilst meeting the dynamic and intensive demands of the Calendar Menu display (rendering new information dynamically as the user interacts with the time frame), creating a new list within the controllers summary methods each time it is called leads to unnecessary memory allocation. In small scale systems like this, this issue is reduced slightly, however in larger scale real world applications with many more transactions it is very memory inefficient.

Alternatively, a cache based approach in which the list is only updated when changes are made, compared to recreating it every render, would improve performance.

Menu Stack: MenuStack.cs

To understand how the application navigates between views, it is important to understand the implementation of the MenuStack and purpose of the Menu base class before looking at specific menus. To implement multiple menus in a console based application, methods of the Stack data structure in C# offer a way to store and move ‘forwards’ and ‘backwards’ through different displays using the Push and Pop methods respectively. When a new menu is opened, it is pushed onto the stack. When a menu is closed, it is popped from the stack.

```
public static class MenuStack
{
    public static Stack<Menu> menuStack = new Stack<Menu>();

    public static void PushMenu(Menu menu)
    {
        menuStack.Push(menu);
        menu.Run();
    }
```

Implementing a PushMenu method that pushes the desired, or ‘next’ menu to be displayed to the top of the stack, and calling its Run method, allows navigation to be achieved when a user makes a selection that triggers a new menu to be displayed in the UI. The same implementation is achieved via a PopMenu method, to navigate ‘backwards’, to the previous menu.

Crucially, all menus requiring navigation to and from other menus, must derive from the Menu base class, allowing the Liskov Substitution Principle to be harnessed. This implementation allows all Menus to be stored in one stack, regardless of their true type, for centralised management of menus, whilst also allowing Menu derived objects to override the Run method with their own implementation.

In earlier phases of the design when only a Calendar Menu was the only interactive menu proposed, this implementation was not necessary. As other menu ideas were suggested, it became necessary to have a significant portion of the application displayed through Menu objects for consistency. Whilst significantly increasing complexity in managing navigation, the extensive UI was considered to be worthwhile.

View Case: CalendarMenu.cs

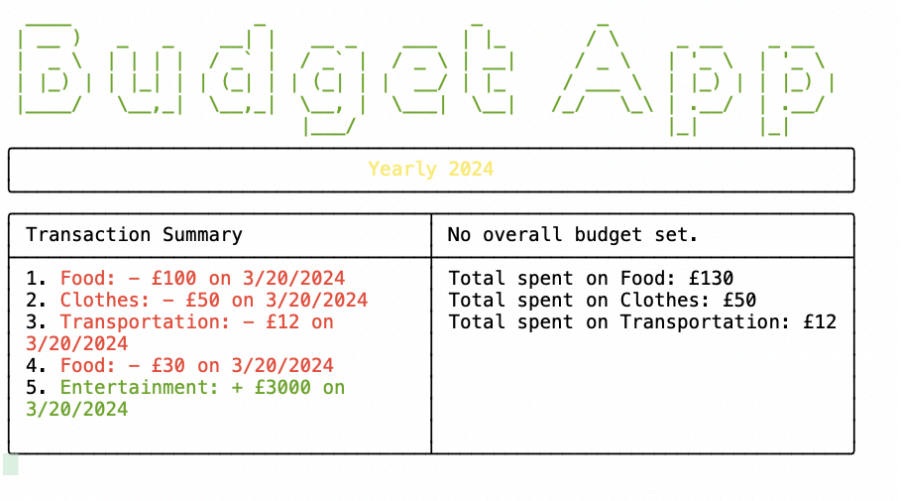
It is the responsibility of the Calendar Menu to provide the main user interface of the application, rendering a complete summary of transactions, spending vs budget comparisons (if set) as well as category specific comparisons depending on the time frame selected by the user, as described in the design. This menu incorporates render methods, using the menus calendar attribute to render all the summary methods for the selected time, via use of the controllers previously discussed.

The class uses an instance of the Calendar class to manage and display dates. The calendar class provides its own methods to navigate through time scales and change the date range displayed in response to keys pressed by the user. The date range has a beginning and an end, represented by two DateTime objects, that can be passed into the controller summary methods to be rendered in turn by the CalendarMenu.

```
private string RenderTransactionSummary()
{
    return TransactionController.TransactionSummary(calendar.Start, calendar.End);
}

private string RenderCategorySummary()
{
    var sb = new StringBuilder();
    foreach (var summary in CategoryController.CategorySummary(calendar.Start, calendar.End, calendar))
    {
        sb.AppendLine(summary);
    }
    return sb.Length == 0 ? "No categories to display" : sb.ToString();
}
```

Presenting the dynamic rendering visually through ASCII art enhances the UI's appeal, creating an engaging and interactive user experience.

```
private void DisplayMenu()
{
    Console.Clear();
    var asciiArt = new FigletText("Budget App");
    AnsiConsole.Render(asciiArt.Color(Color.Green));

    var table1 = new Table() { Border = TableBorder.Rounded };

    table1.AddColumn(new TableColumn("[yellow]" + calendar._currentScale + " " +
      calendar.Start.ToString(calendar.Format) + "[/]").Centered());
    table1.Width(75);

    AnsiConsole.Render(table1);

    var table2 = new Table() { Border = TableBorder.Rounded };

    table2.AddColumn(new TableColumn("Transaction Summary").LeftAligned());
    table2.AddColumn(new TableColumn(RenderBudgetBreakdown()).LeftAligned());
    table2.AddRow(new Markup(RenderTransactionSummary()).LeftJustified(), new
      Markup(RenderCategorySummary()).Centered());
    table2.Width(75);

    AnsiConsole.Render(table2);
}
```



By extending the Menu class, CalendarMenu is able to be used by the MenuStack class to manage navigation between other menus, via pop and push methods.

```
public override void Run()
{
    while (true)
    {
        DisplayMenu();
        NavigateMenu();
    }
}

private void NavigateMenu()
{
    var keyPressed = MenuFeatures.GetInput();
    if (keyPressed == ConsoleKey.Backspace)
    {
        MenuStack.PopMenu();
    }
    else if (keyPressed == ConsoleKey.Enter)
    {
        var transactions = TransactionController.GetTransactionsWithinRangeAsArray(calendar.Start, calendar.End);
        if (transactions.Length > 0)
        {
            MenuStack.PushMenu(new TransactionSelectionMenu("Transaction Selection Menu", transactions));
        }
    }
    else
    {
        calendar.NavigateCalendar(keyPressed, "yyyy/MM", "yyyy", "yyyy/MM/dd");
    }
}
```

The implementation follows a 3 step process:

1. The CalendarMenu class is instantiated and its Run method is called. The method contains a loop that continuously displays the menu and waits for user input.

2. The DisplayMenu method is called to display the current state of the menu. This includes the date range and renders data within that range, including transactions, category budgets, and overall budgets.

3. The NavigateMenu method is called to handle user input. Depending on the key pressed by the user, different actions are performed. For example, if the user presses the backspace key, the PopMenu method is called from the MenuStack. If the enter key is pressed, the PushMenu method is invoked and the TransactionSelectionMenu is rendered. If the user presses the arrow keys, the calendar is navigated, switching dates and time spans.

TransactionSelectionMenu, see below, gives the user the ability to edit, remove, or add a note to a transaction.

Bridge Case: Database.cs

The persistence of data in the application is achieved through three elements: the Database class, the IFileFormat interface, and its concrete implementation in the JsonFormat class.

Database class: This class is a singleton, ensuring that only one instance of the database exists throughout the application. It utilises the IFileFormat interface to interact with the file system, abstracting away the details of how data is saved and retrieved, unconcerned with the concrete implementation.

```
public List<Transaction> LoadTransactions()
{
    return fileFormat.Load(transactionFilePath);
}

public void SaveTransactions(List<Transaction> transactions)
{
    fileFormat.Save(transactions, transactionFilePath);
}
```

IFileFormat interface: This interface specifies the methods, or design contract, that any file format class must implement. It includes methods for loading and saving Transaction and Category objects.

```
public interface IFileFormat
{
    public abstract List<Transaction> Load(string filePath);
    public abstract void Save(List<Transaction> transactions, string filePath);
    public abstract List<Category> LoadCategories(string filePath);
    public abstract void SaveCategories(List<Category> categories, string filePath);
}
```

JsonFormat class: This class implements the IFileFormat interface. It uses the Newtonsoft.Json library to serialise and deserialise Transaction and Category objects to and from a JSON format. The Load and Save methods read and write data to the file system through a file path string.

```
public List<Transaction> Load(string filePath)
{
    string jsonContent = File.ReadAllText(filePath);
    List<Transaction> transactions = JsonConvert.DeserializeObject<List<Transaction>>(jsonContent, settings);
    return transactions;
}

public void Save(List<Transaction> transactions, string filePath)
{
    string jsonContent = JsonConvert.SerializeObject(transactions, settings);
    File.WriteAllText(filePath, jsonContent);
}
```

With this implementation, the Database class is decoupled from the JsonFormat class, abstracted through the IFileFormat interface. This design allows for the easy addition of formats such as XML or .txt by creating a new class that implements the IFileFormat interface.
The Database class wouldn't need to change as it interacts with the interface, not the classes that have concrete implementations, leveraging the main advantages of the bridge design in the implementation.

Factory Case: TransactionFactory.cs

The TransactionFactory contains a private static instance, which is used to implement the singleton pattern. The GetInstance method checks if the _instance is null and if so, creates the new TransactionFactory object. From here the TransactionFactory is responsible for creating transaction objects based upon user input gathered in the TransactionCreatorMenu.

As a result of this design, the TransactionCreatorMenu, used to receive inputs from the user about a new transaction, holds a Transaction object, known as the dummyTransaction which stores the user's input about the transaction. Once 'Create Transaction' is selected within the Menu, the Transaction Factory uses the attributes of the dummyTransaction to create the new transaction object, and the dummyTransaction object can be used for subsequent data entry about a transaction.

The CreateTransaction method takes the category name, date, amount, and a boolean indicating whether the transaction is an income or expense. It creates a new Transaction object of the appropriate type and adds it to the repository in the TransactionController.

```
public Transaction CreateDummyTransaction(string categoryName, DateTime date, double amount) => new
  Transaction(categoryName, date, amount);

public Transaction CreateTransaction(string categoryName, DateTime date, double amount, bool isIncome)
{
    Transaction newTransaction = isIncome ? new Income(categoryName, date, amount) : new
      Expense(categoryName, date, amount);
    TransactionController.AddTransaction(newTransaction);
    return newTransaction;
}
```

Creating a new Income or Expense object based on a user-specified boolean collected from the UI, with a Transaction reference allows the Transaction Controller to manage both derived types of Transaction within a method, regardless of its true type. Conditional expressions, or ternary operators) are used for concise readable code.

The CreateRecurringTransaction method is used to create multiple transactions that occur and reoccur over a specific period. It calculates the total number of transactions based on the frequency and the duration of the recurrence, and then creates individual transactions for each occurrence. The frequency can be daily, weekly, monthly, or yearly, and the method adjusts the date of each transaction accordingly.

The TransactionFactory class encapsulates the logic for creating Transaction objects, separating this responsibility from other parts of the application. Whilst not designed as a typical factory, the effort made to implement the factory has encapsulated the function of creating individual and recurring transactions. This added utility and singleton functionality isn't typical however serves the purpose of this application well.

Composition Case: SelectableMenu.cs

The implementation of the SelectableMenu class follows a compositional design. This class is composed of two main parts: the navigation logic and the display logic. Each component part is encapsulated in its own class, MenuNavigator and MenuDisplay and is responsible for a specific aspect of the menu's functionality.

Their key methods Navigate and DisplayOptions from the MenuNavigator and MenuDisplay classes, are held in a loop until an option is selected. In this way, the display is re-rendered each time a new option is chosen.

```
public class SelectableMenu
{
    private MenuNavigator _menuNavigator;
    public MenuDisplay _menuDisplay;
    public string[] Options { get; set; }
    private string Prompt { get; }
    private int selectedIndex;

    public SelectableMenu(string prompt, string[] options)
    {
        this._menuNavigator = new MenuNavigator();
        this._menuDisplay = new MenuDisplay();
        this.Options = options;
        this.Prompt = prompt;
        this.selectedIndex = 0;
    }

    public int Run()
    {
        while (true)
        {
            _menuDisplay.DisplayOptions(Prompt, Options, selectedIndex);
            var keyPressed = MenuFeatures.GetInput();
            selectedIndex = _menuNavigator.Navigate(keyPressed, Options, selectedIndex);
            if (keyPressed == ConsoleKey.Enter)
            {
                return selectedIndex;
            }
        }
    }
}
```

1. Navigation: The navigation logic handles the user's input and moves through the menu options. This includes moving the selection up and down when the arrow keys are pressed, and returning the selected option when the enter button is pressed. It does this through a series of if statements which tracks the index of the selected option. For example if the user is currently on index 0 and presses the right arrow key, the selectedIndex is incremented by 1. If this is done on the last index, it moves back to index 0, looping back around to the first option.
2. Display: The display logic is responsible for rendering the menu on the screen. It displays the menu options and highlights the currently selected option, surrounding the selected button on the menu in asterisks.

In this implementation, the SelectableMenu delegates responsibility for different tasks to its component parts, the navigation and the display. This separation of concerns makes the code easier to understand, test, and modify.

Reflections on Implementation

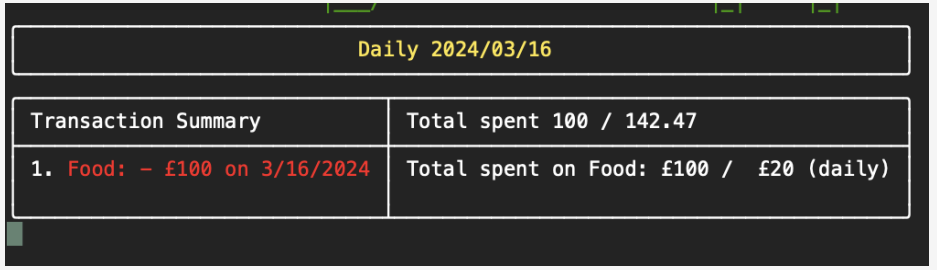
Changes in design inevitably add complexity to the implementation. This mainly occurred when the addition of multiple dynamic menus was added to the design, changing the logic from step by step console interaction, to interacting with the console via a constantly re rendered UI (i.e. CalendarMenu)I that's navigation provided via keys pressed. In the example of entering date

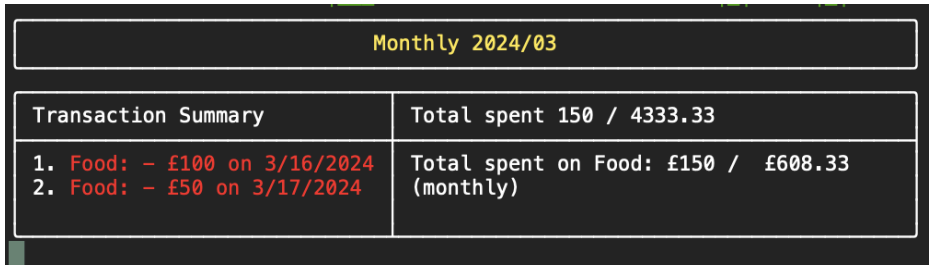
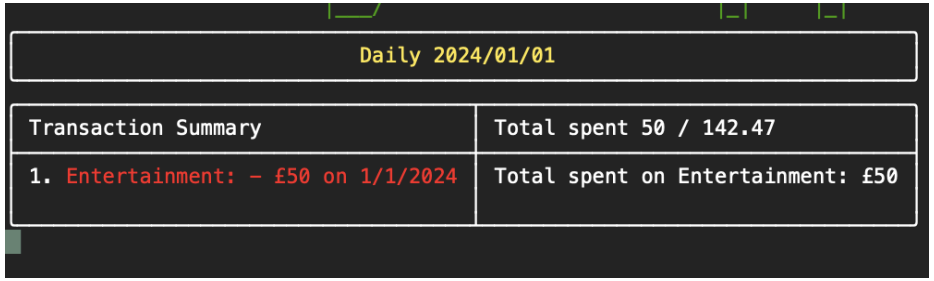
information, in the original design dates would be typed into the console, only requiring simple error handling and validation to ensure a valid date was entered by the user. In the final design, selecting the date via changing scrolling through values using the arrow keys required a series of classes and methods to be implemented, handling changes in scale based upon keys pressed.

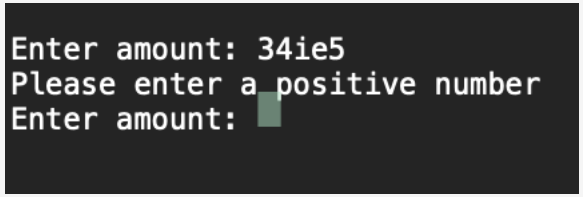
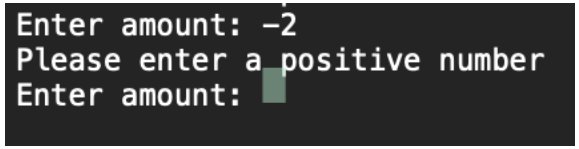
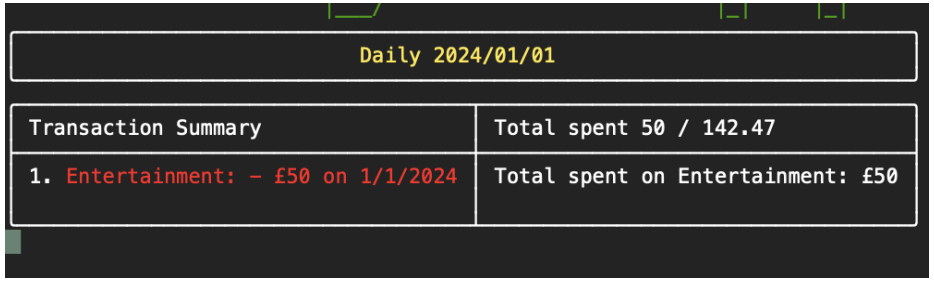
Whilst the role of the developer is to develop the prototype from the detailed design, the ambition of the team lead to significantly more implementation after the initial prototype, ultimately leading to a few lengthy classes and methods that lose the high standard of and modularity initially implemented from the original design. Despite this, the final product provides significant functionality and demonstrates the benefit of the design patterns implemented.

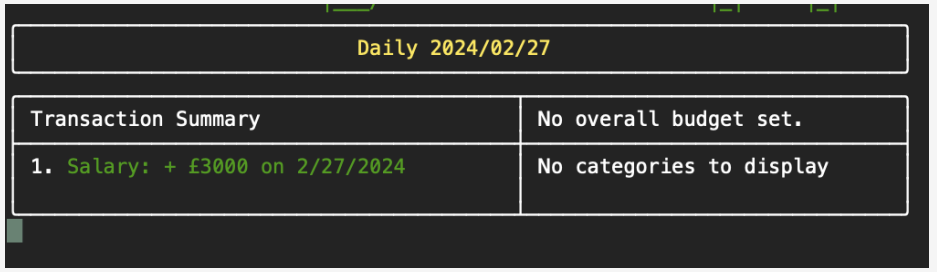
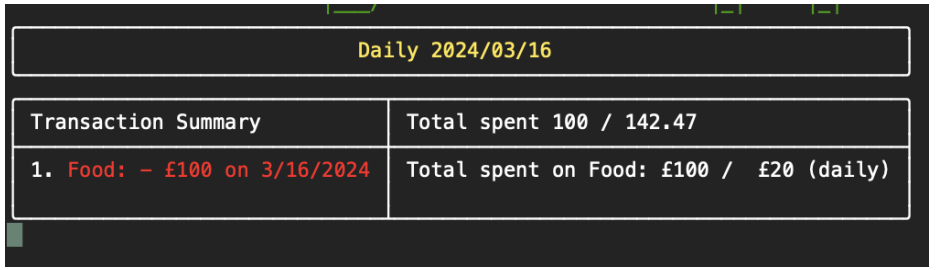
Testing - Brad Prosser

Testing Tables

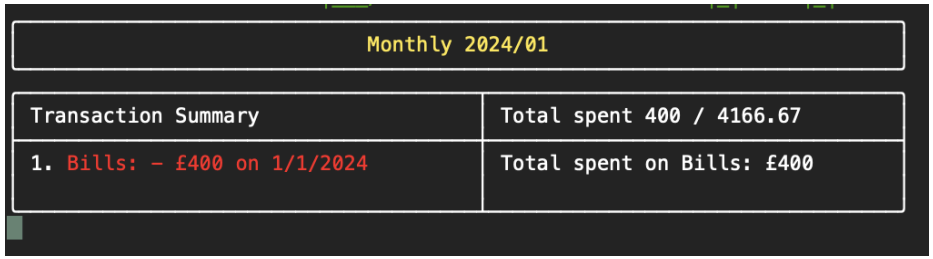
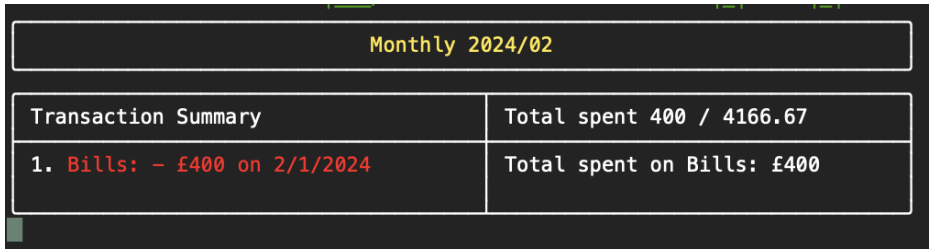
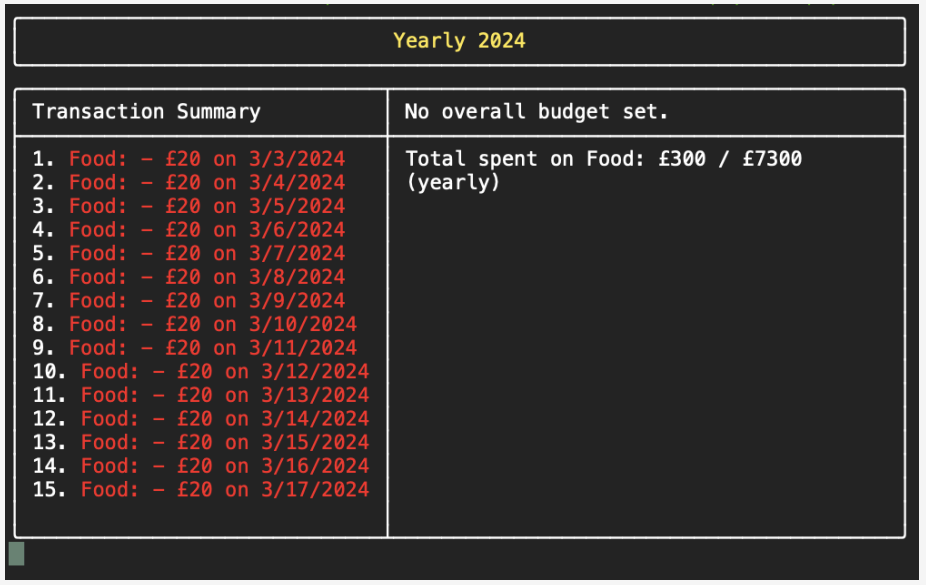
Test	Test Input Data/Action	Expected Outcome	Outcome Screenshot	Pass/ Fail
View main menu	Run application	Application runs successfully and displays the main menu		Pass
View transactions - Daily	Press enter on 'View Transactions' and scroll to Daily	List of daily transactions displayed (related to any budget allocations)		Pass

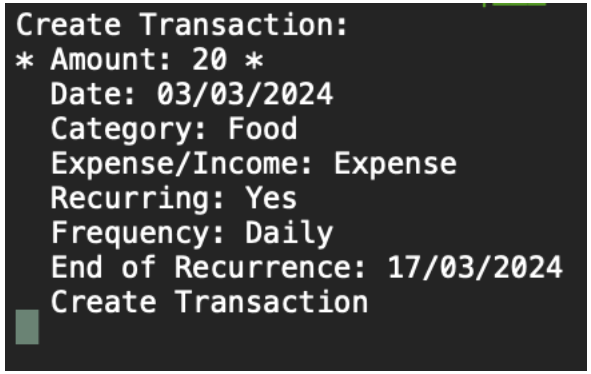
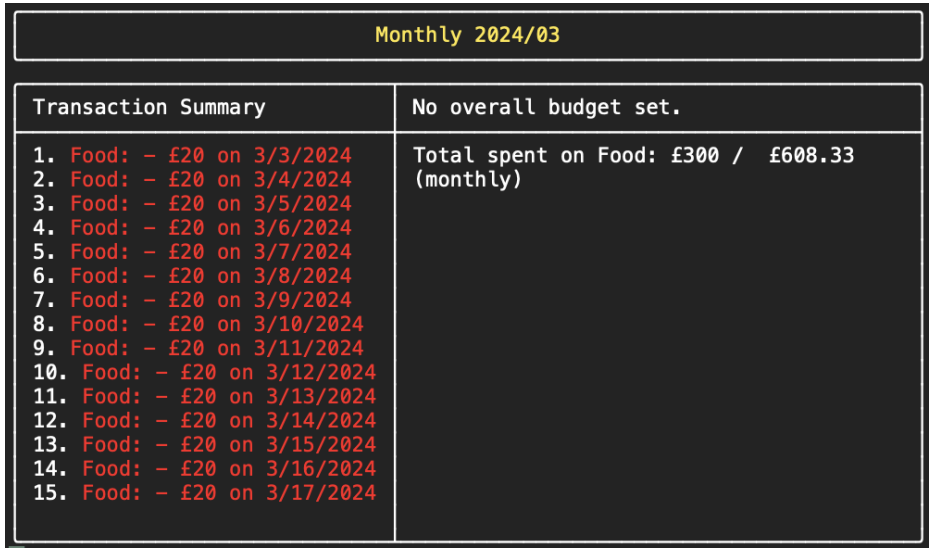
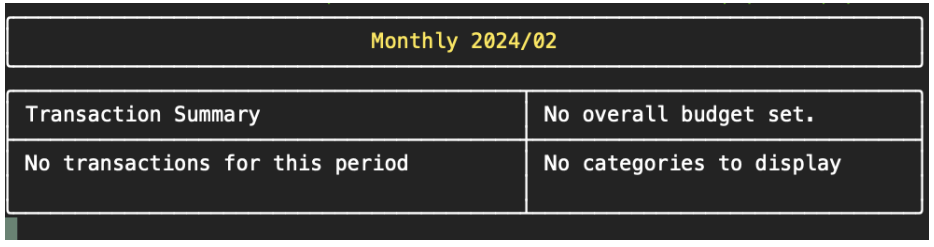
Test	Test Input Data/Action	Expected Outcome	Outcome Screenshot	Pass/ Fail
View transactions - Monthly	Press enter on ‘View Transactions' and scroll to Monthly	List of monthly transactions displayed (related to any budget allocations)		Pass
View transactions - Yearly	Press enter on 'View Transactions’ and scroll to Yearly	List of yearly transactions displayed (related to any budget allocations)		Pass
Add transaction - Amount	Press enter on ‘Add Income/Expense’ and enter 50 for Amount of chosen entry	Chosen entry should display £50 in View Transactions menu		Pass

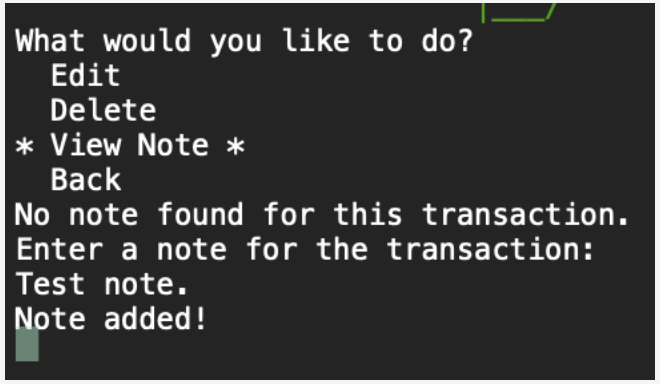
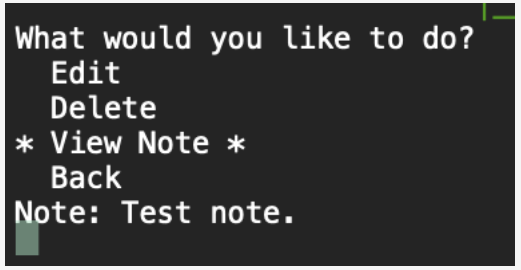
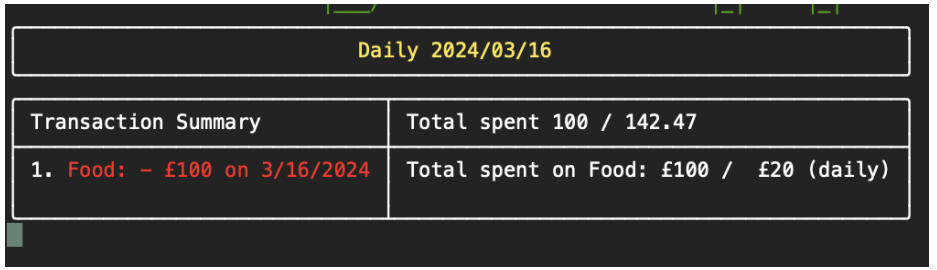
Test	Test Input Data/Action	Expected Outcome	Outcome Screenshot	Pass/ Fail
Add transaction - Invalid parameter (characters)	Press enter on ‘Add Income/Expense’ and enter ‘34ie5’	Application will reject entry and request a positive number		Pass
Add transaction - Invalid parameter (negative number)	Press enter on ‘Add Income/Expense’ and enter ‘-2’	Application will reject entry and request a positive number		Pass
Add transaction - Invalid parameter (0)	Press enter on ‘Add Income/Expense’ and enter ‘0’	Application will reject entry and request a positive number *Fail reason: Application allows an entry of 0*		Fail
Add transaction - Expense	Press enter on ‘Add Income/Expense’ and select ‘Expense’ for chosen entry	Chosen entry should be displayed in red and be preceded by the ‘-’ symbol		Pass

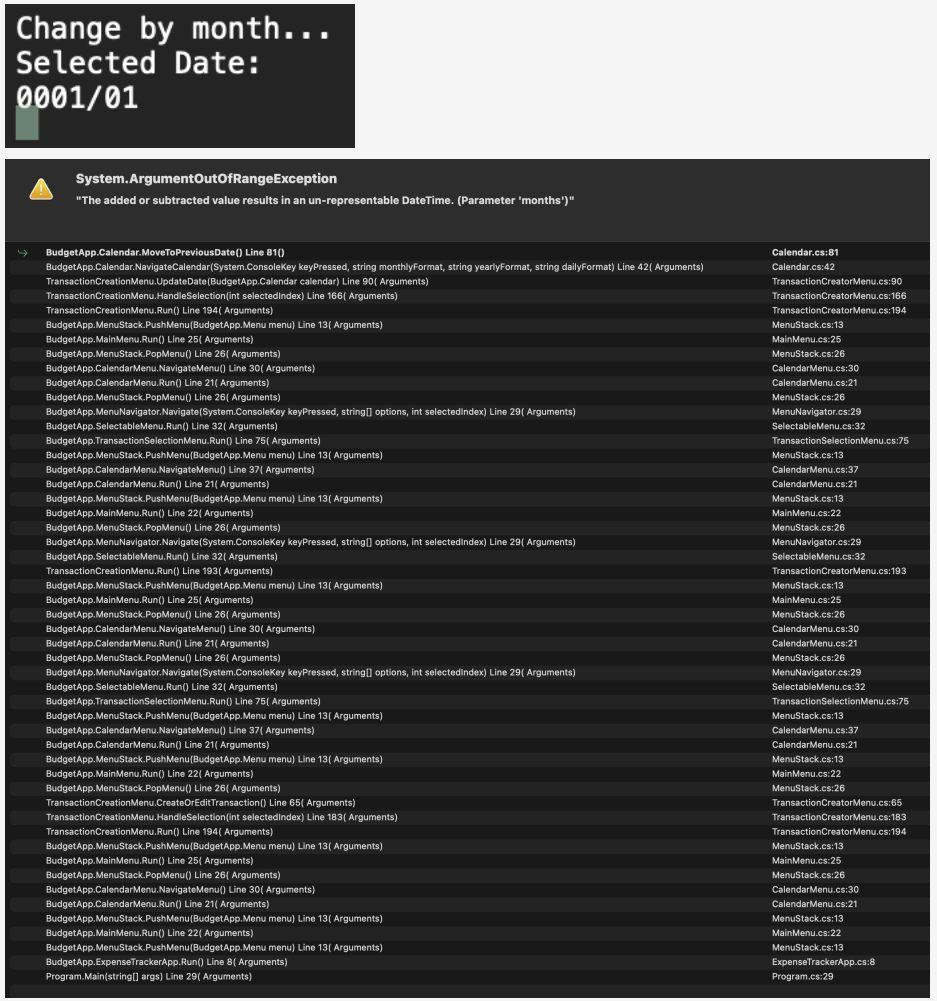
Test	Test Input Data/Action	Expected Outcome	Outcome Screenshot	Pass/ Fail
Add transaction - Income	Press enter on ‘Add Income/Expense’ and select ‘Income’ for chosen entry	Chosen entry should be displayed in green and be preceded by the ‘+’ symbol		Pass
Add transaction - Category	Press enter on ‘Add Income/Expense’ and select ‘Food’ for category of chosen entry	Chosen entry should display the ‘Food’ category in the View Transactions menu		Pass

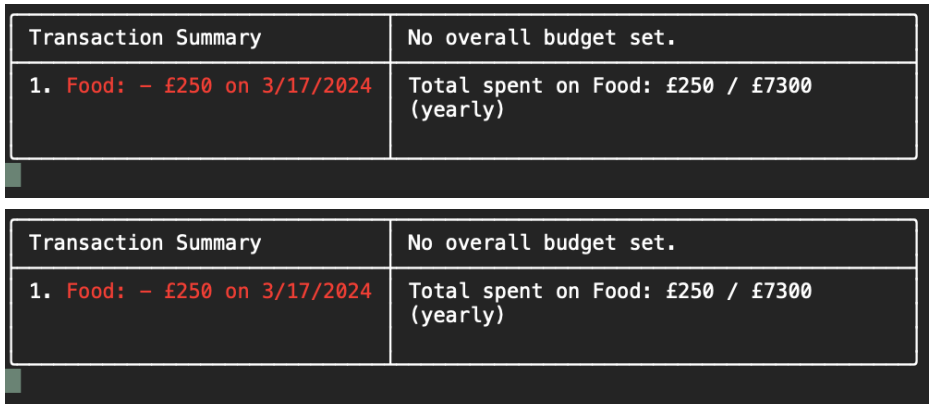
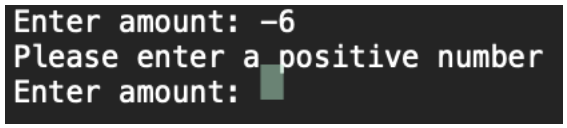
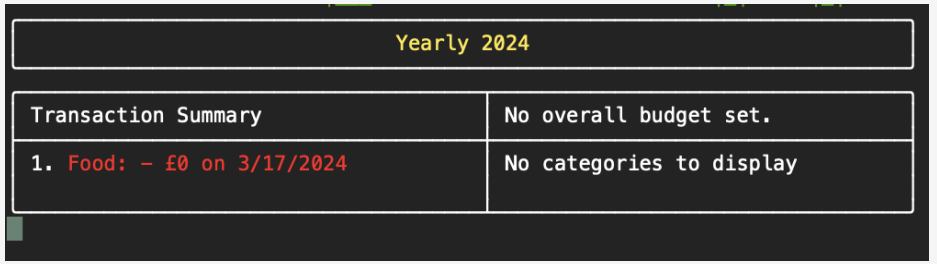
Test	Test Input Data/Action	Expected Outcome	Outcome Screenshot	Pass/ Fail
Add transaction - Recurring (yearly)	Press enter on ‘Add Income/Expense’ , toggle ‘Recurring’ to yes, and select ‘Yearly’ and input an entry	Chosen entry should display the same transaction recurring yearly (in line with chosen start and end date) in View Transactions menu		Pass

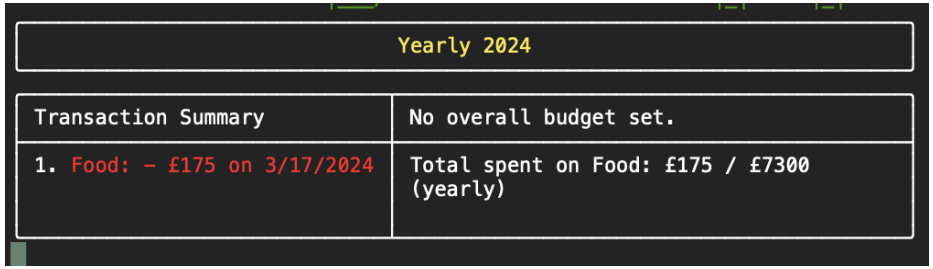
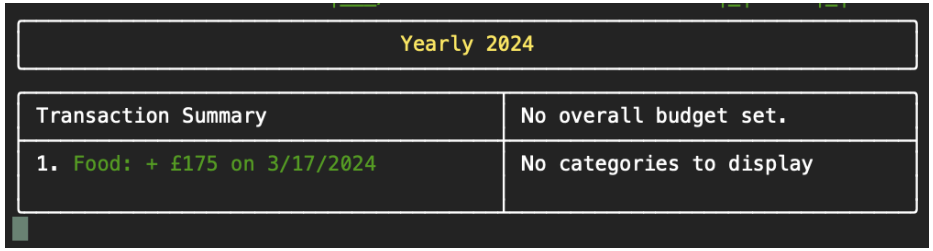
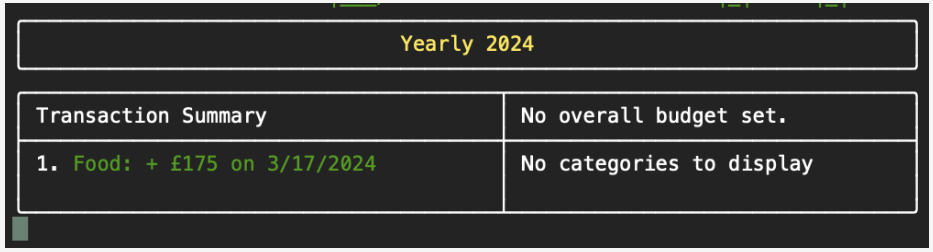
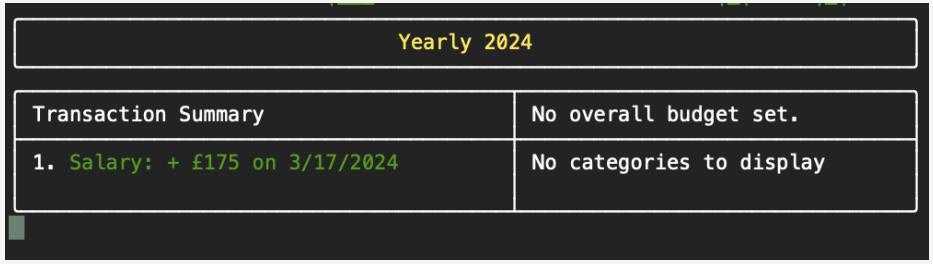
Test	Test Input Data/Action	Expected Outcome	Outcome Screenshot	Pass/ Fail
Add transaction - Recurring (monthly)	Press enter on ‘Add Income/Expense’, toggle ‘Recurring’ to yes, and select ‘Monthly’ and input an entry	Chosen entry should display the same transaction recurring monthly (in line with chosen start and end date) in View Transactions menu	 	Pass
Add transaction - Recurring (daily)	Press enter on ‘Add Income/Expense’, toggle ‘Recurring’ to yes, and select ‘Daily’ and input an entry	Chosen entry should display the same transaction recurring daily (in line with chosen start and end date) in View Transactions menu		Pass

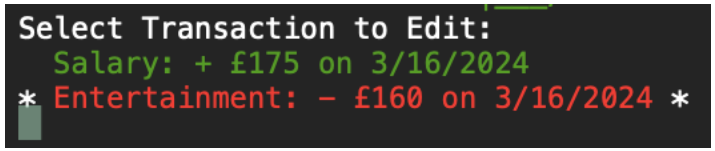
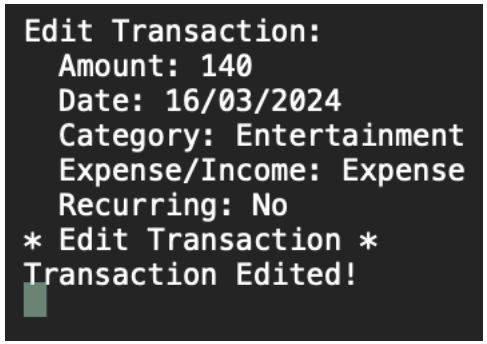
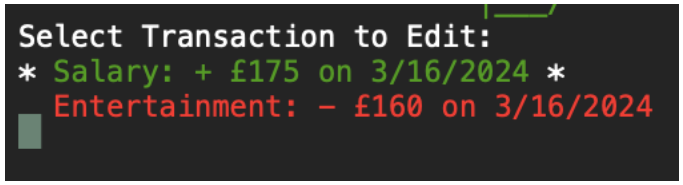
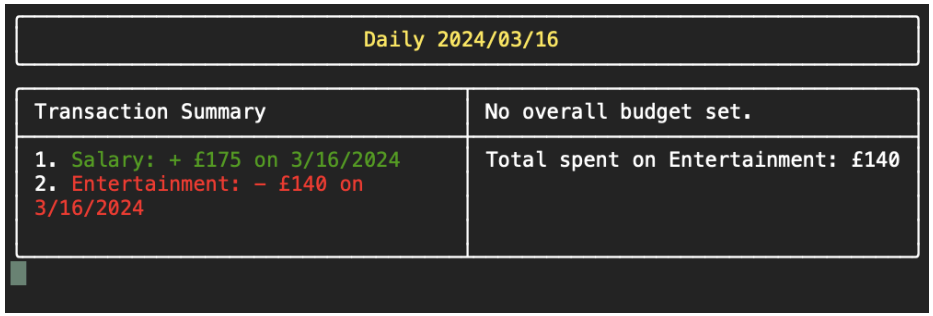
Add transaction - Recurring (date check)	Press enter on ‘Add Income/Expense’, toggle ‘Recurring’ to yes, and select ‘Daily’ and input an entry that only exists within one month	Chosen entry should display the same transaction recurring daily - only within the single month date parameters set - in View Transactions menu	  	Pass

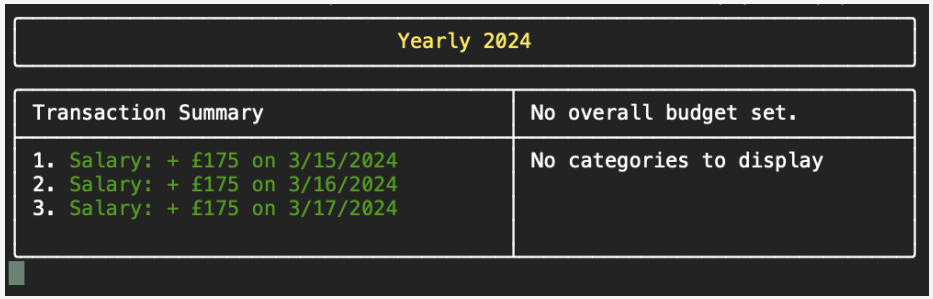
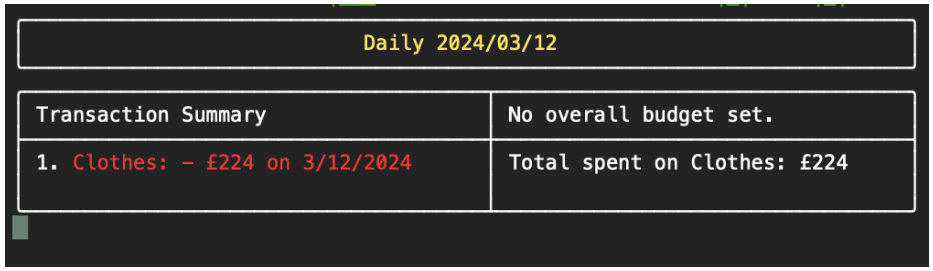
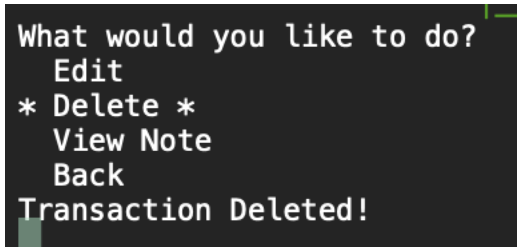
Test	Test Input Data/Action	Expected Outcome	Outcome Screenshot	Pass/ Fail
Add transaction - Add note	Press enter on ‘View Transactions’ and then press enter again to edit a transaction	Press enter on ‘View Note’, application will say no note is found and provide a screen to enter note. Re-entering chosen entry and selecting ‘View Note’ will display note	 	Pass
Add transaction - Date selection	Press enter on ‘Add Income/Expense’ choose desired date	Chosen entry should display the transaction with the chosen date in View Transactions		Pass

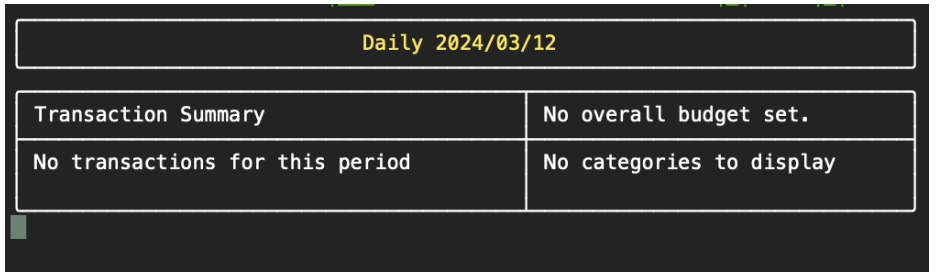
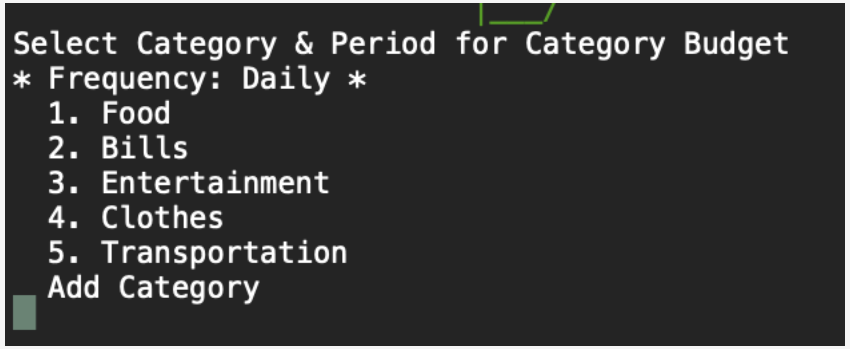
Test	Test Input Data/Action	Expected Outcome	Outcome Screenshot	Pass/ Fail
Add transaction - Date selection scroller	Press enter on ‘Add Income/Expense’ and choose desired date through scrolling the daily/monthly/yearly options	Application should allow scrolling through options of daily/monthly/yearly date selections without any errors *Fail reason: Application allows scrolling down through dates, and then afterwards in either direction - but if entering and immediately scrolling up, the date year resets to 0001. Further toggling the date below this 0001 will also crash the application.*		Fail

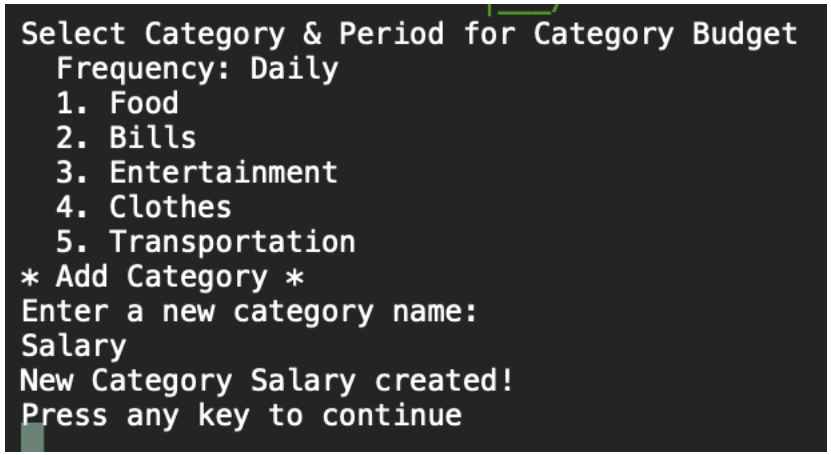
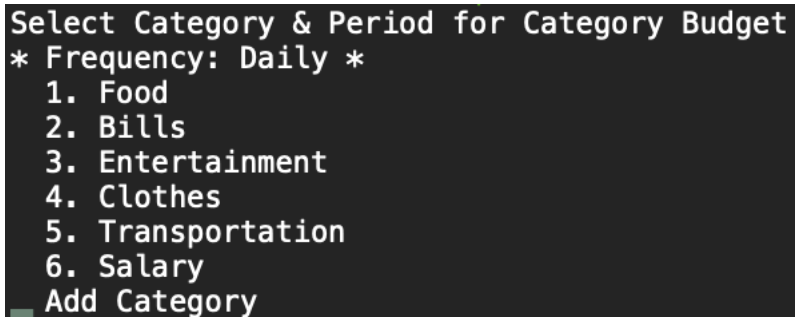
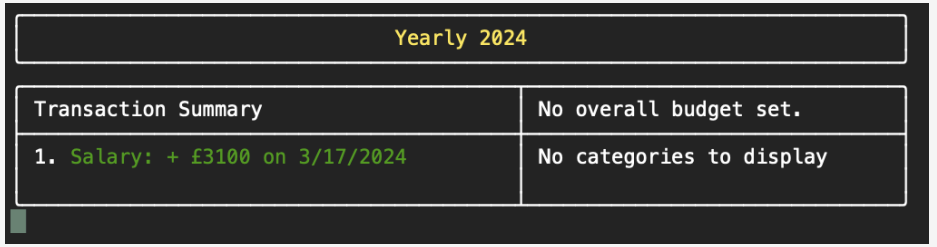
Test	Test Input Data/Action	Expected Outcome	Outcome Screenshot	Pass/ Fail
Edit transaction - Amount	Press enter on ‘View Transactions’ and then press enter again to edit a transaction, modify amount	Chosen entry to edit should be successfully updated with new amount		Pass
Edit transaction - Invalid parameter (characters)	Modify transaction amount to ‘37s3d’	Application will reject entry and request a positive number		Pass
Edit transaction - Invalid parameter (negative number)	Modify transaction amount to ‘-6’	Application will reject entry and request a positive number		Pass
Edit transaction - Invalid parameter (0)	Modify transaction amount to ‘0’	Application will reject entry and request a positive number *Fail reason: Application allows an entry of 0*		Fail

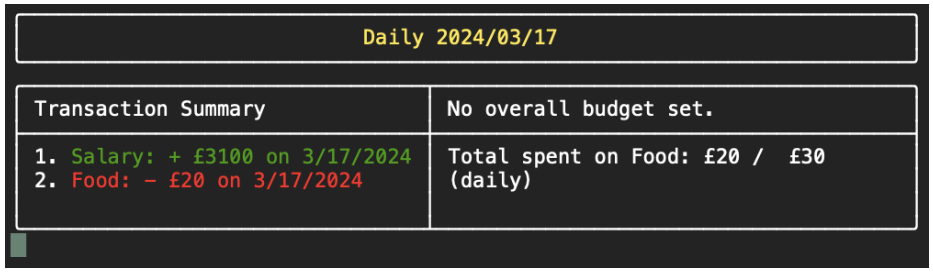
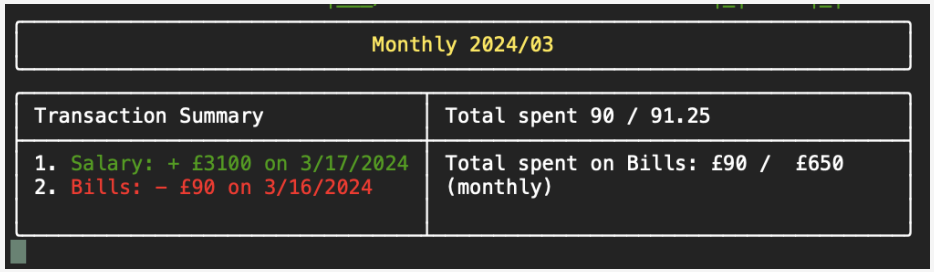
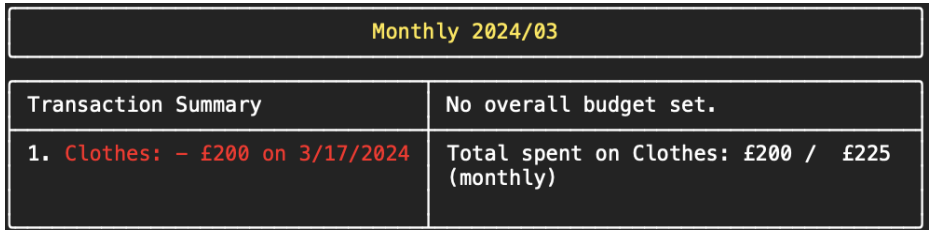
Test	Test Input Data/Action	Expected Outcome	Outcome Screenshot	Pass/ Fail
Edit transaction - Income/Expense	Toggle transaction type between income and expense	Application will convert an expense to an income or an income to an expense if requested	 	Pass
Edit transaction - Category	Modify category of a transaction	Application will successfully change the category to whatever is requested	 	Pass

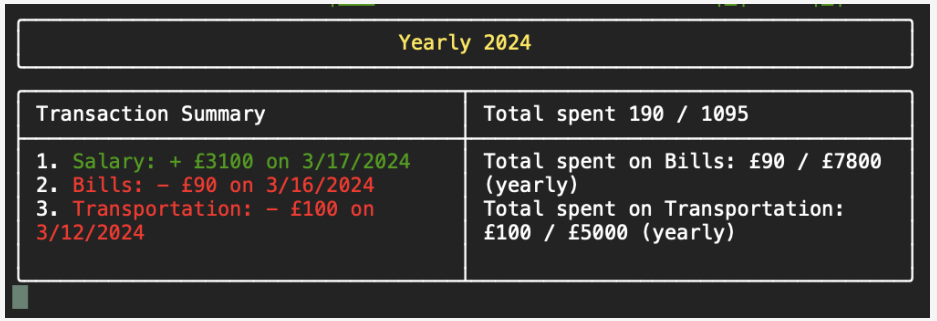
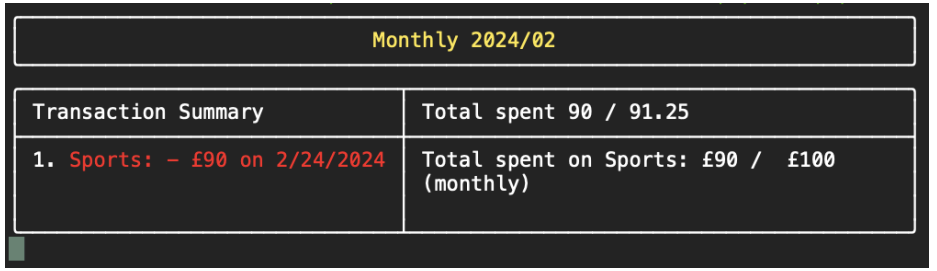
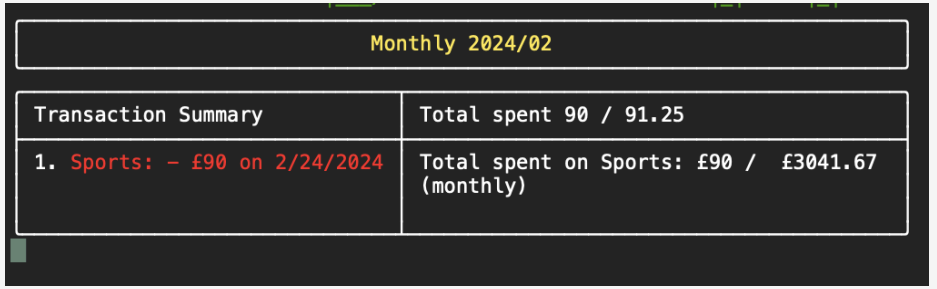
Test	Test Input Data/Action	Expected Outcome	Outcome Screenshot	Pass/ Fail
Edit transaction - Fully updated menu navigation	Modify a transaction and go back one menu level	Application will successfully update a transaction with the edits in the previous menu *Fail reason: Although the application successfully edits each transaction in every requested way in both the database and in the main View Transactions menu, the old transaction is still briefly displayed when exiting the edit transaction menu to get back to the main menu (as displayed in the third screenshot).*	   	Fail

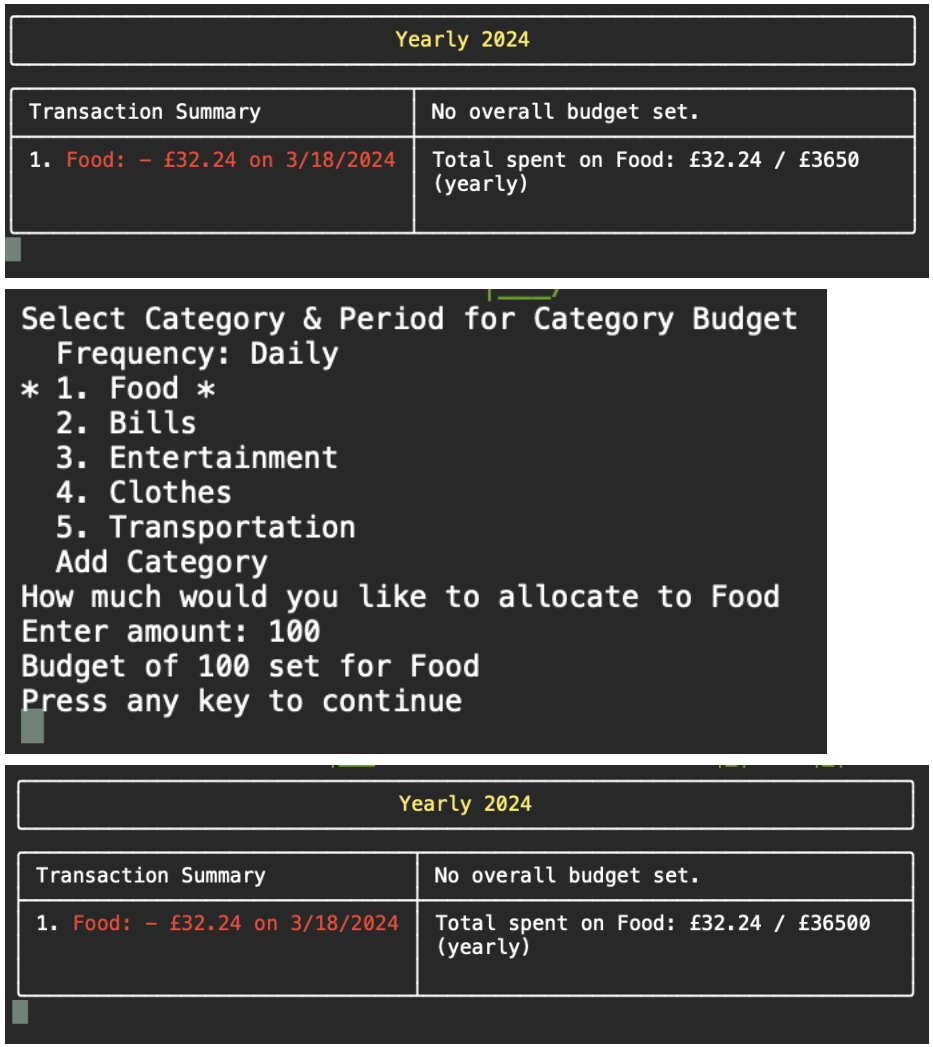
Test	Test Input Data/Action	Expected Outcome	Outcome Screenshot	Pass/ Fail
Edit transaction - Recurring	Modify a transaction to become recurring	Application will successfully convert a transaction to recurring (within requested date parameters)		Pass
Delete transaction	Press enter on 'View Transactions' and then press enter again to display the option to delete a transaction, then delete it	Application will successfully delete the transaction	 	Pass

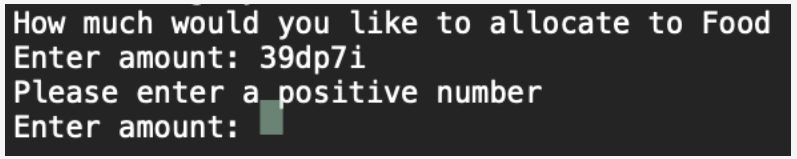
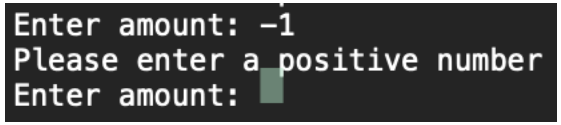
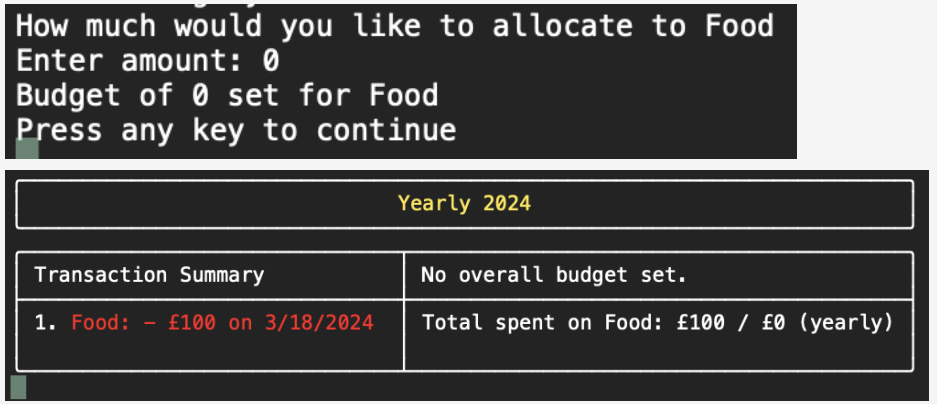
Test	Test Input Data/Action	Expected Outcome	Outcome Screenshot	Pass/ Fail
				
View list of preset categories	Press enter on ‘Manage Categories’ to view the preset categories on startup	Application will display a list of the preset categories		Pass

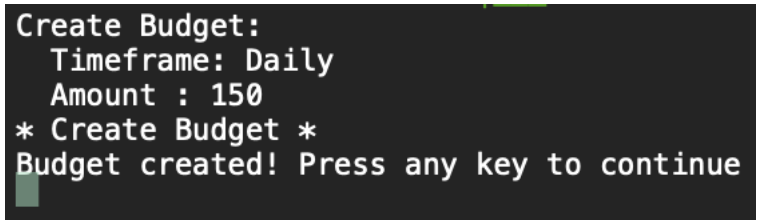
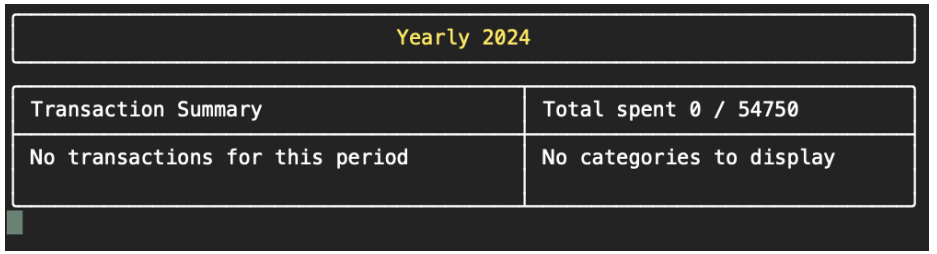
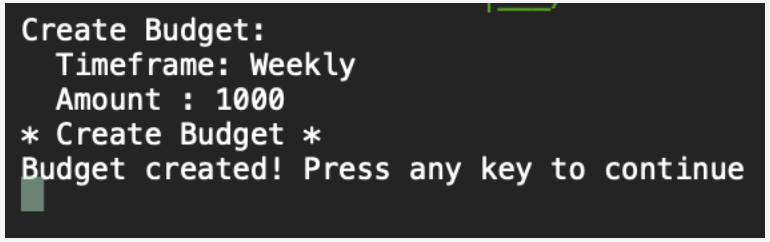
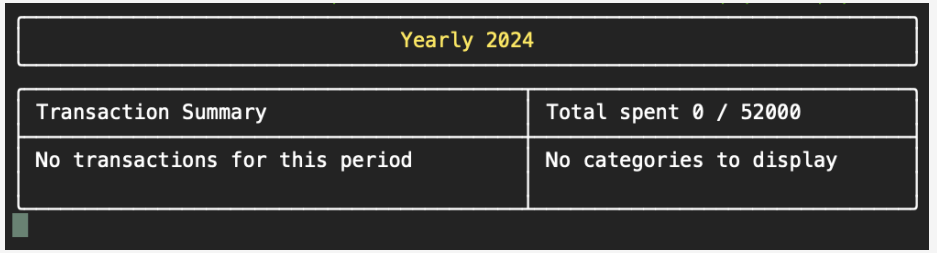
Test	Test Input Data/Action	Expected Outcome	Outcome Screenshot	Pass/ Fail
Add a new category	Press enter on ‘Manage Categories’ and then press enter on ‘Add Category’	Application will display the newly added category in the list of categories	 	Pass
Add a transaction for the newly created category	Press enter on ‘Add Income/Expense’ and enter a transaction for the newly created Salary category	Application will successfully add a transaction under the newly created category		Pass

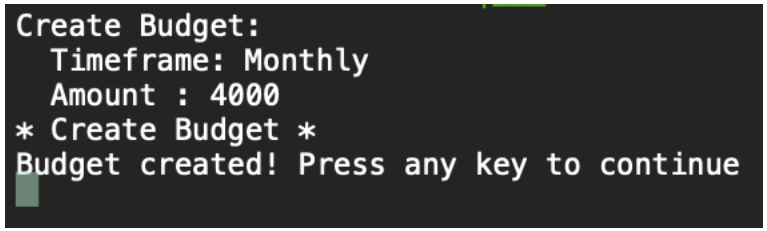
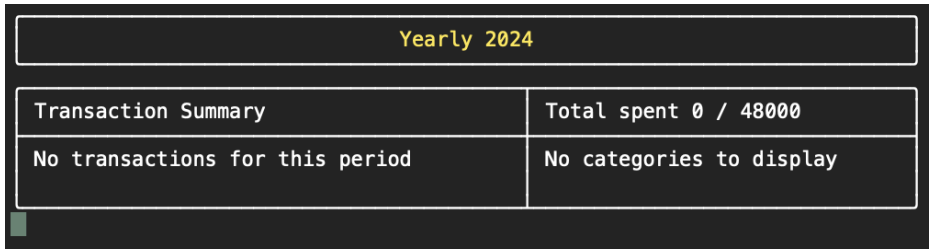
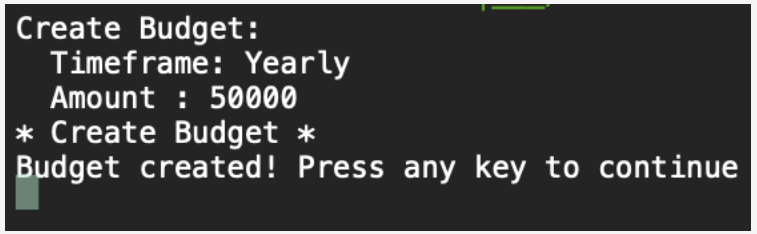
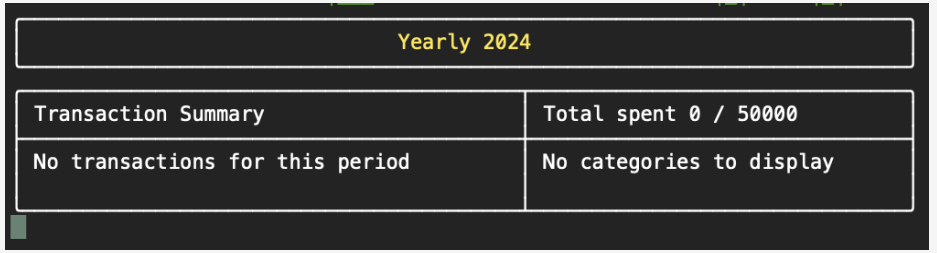
Test	Test Input Data/Action	Expected Outcome	Outcome Screenshot	Pass/ Fail
Enter category budget - Daily	Press enter on 'Manage Categories' and set a daily budget	Application will display the daily budget for chosen category in 'View Transactions' menu		Pass
Enter category budget - Weekly	Press enter on 'Manage Categories' and set a weekly budget	Application will display the weekly budget for chosen category in 'View Transactions' menu *Note: Pass because budget is successfully set weekly, but only able to be displayed monthly in View Transactions*		Pass
Enter category budget - Monthly	Press enter on 'Manage Categories' and set a monthly budget	Application will display the monthly budget for chosen category in 'View Transactions' menu		Pass

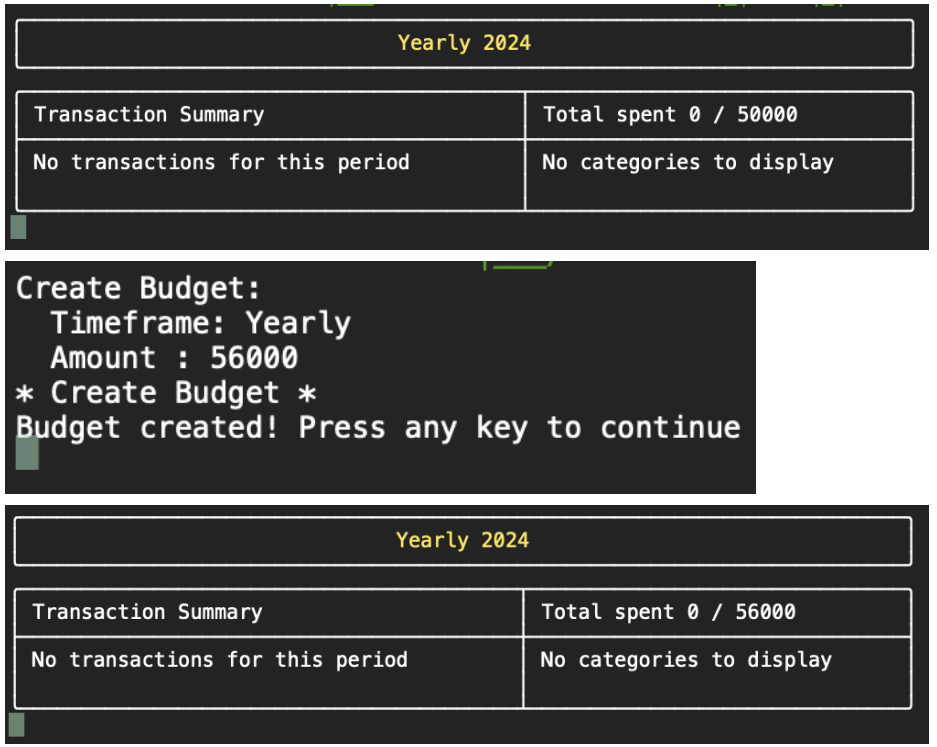
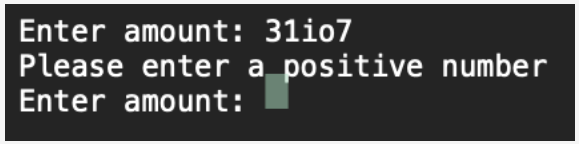
Test	Test Input Data/Action	Expected Outcome	Outcome Screenshot	Pass/ Fail
Enter category budget - Yearly	Press enter on 'Manage Categories' and set a yearly budget	Application will display the yearly budget for chosen category in 'View Transactions' menu		Pass
Enter category budget - Newly created category	Create a new category, assign it a monthly budget, then View Transactions	Application will successfully display the budget attached to the newly created category		Pass
Enter category budget - Increase an individual category's budget	Add an additional budget for an individual category that already has an assigned budget	Application will add the additional budget amount to the existing budget amount, hence increasing it for the selected category		Pass

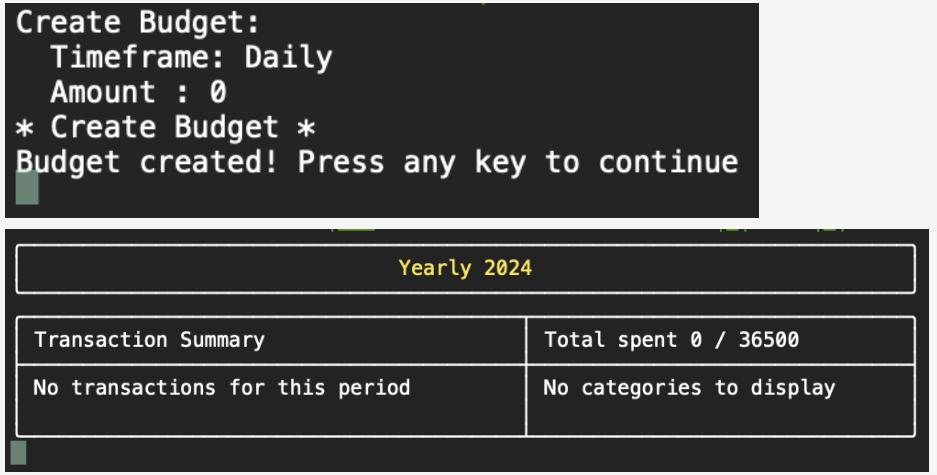
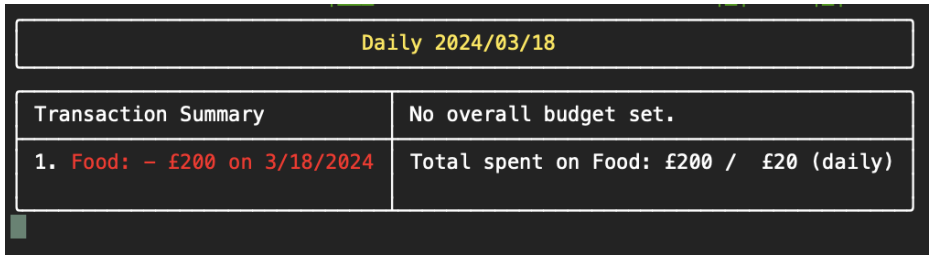
Test	Test Input Data/Action	Expected Outcome	Outcome Screenshot	Pass/ Fail
Enter category budget - Replace/Over write existing budget	Decrease/increase budget for an individual category already assigned a budget	Application will decrease/increase budget amount for a category as requested, overwriting the existing budget with the new budget		Pass

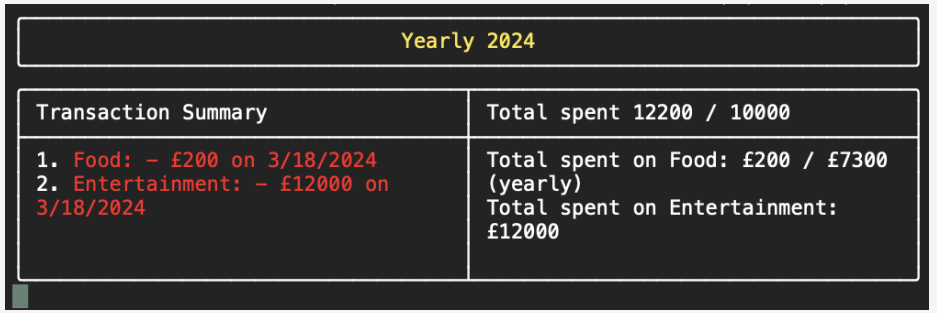
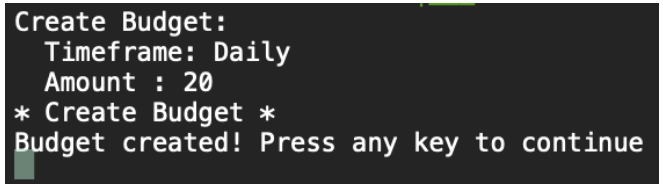
Test	Test Input Data/Action	Expected Outcome	Outcome Screenshot	Pass/ Fail
Enter category budget - Invalid parameter (characters)	Enter amount as ‘39dp7i’	Application will reject entry and request a positive number		Pass
Enter category budget - Invalid parameter (negative number)	Enter amount as ‘-1’	Application will reject entry and request a positive number		Pass
Enter category budget - Invalid parameter (0)	Enter amount as ‘0’	Application will reject entry and request a positive number *Fail reason: Application allows an entry of 0, and further adds this budget to the View Transactions menu*		Fail

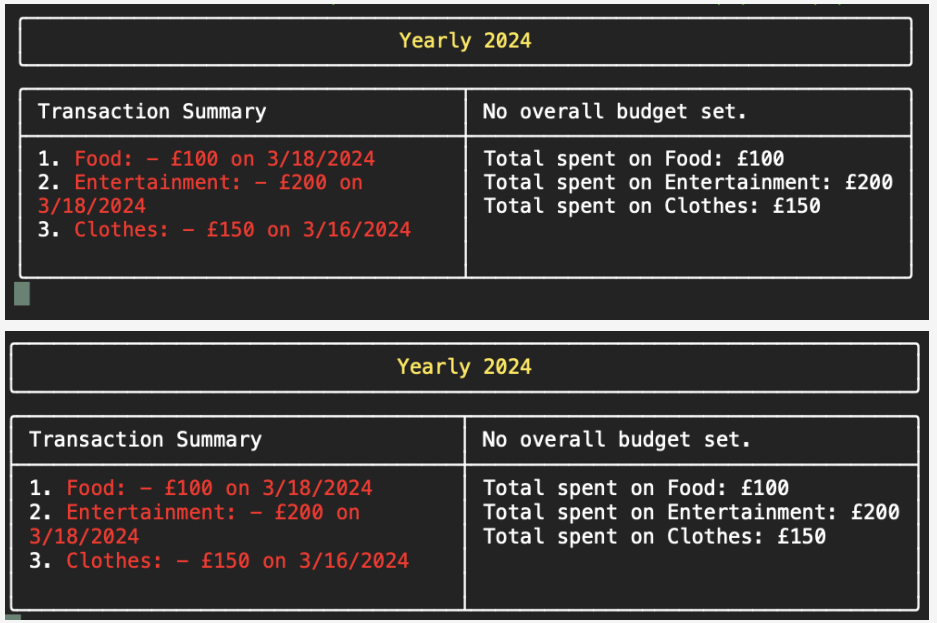
Test	Test Input Data/Action	Expected Outcome	Outcome Screenshot	Pass/ Fail
Enter overall budget - Daily	Press enter on 'Add Budget' and enter an overall daily budget	Application will successfully create a daily overall budget, which will appear in the View Transactions menu	 	Pass
Enter overall budget - Weekly	Press enter on 'Add Budget' and enter an overall weekly budget	Application will successfully create a weekly overall budget, which will appear in the View Transactions menu	 	Pass

Test	Test Input Data/Action	Expected Outcome	Outcome Screenshot	Pass/ Fail
Enter overall budget - Monthly	Press enter on 'Add Budget' and enter an overall monthly budget	Application will successfully create a monthly overall budget, which will appear in the View Transactions menu	 	Pass
Enter overall budget - Yearly	Press enter on 'Add Budget' and enter an overall yearly budget	Application will successfully create a yearly overall budget, which will appear in the View Transactions menu	 	Pass

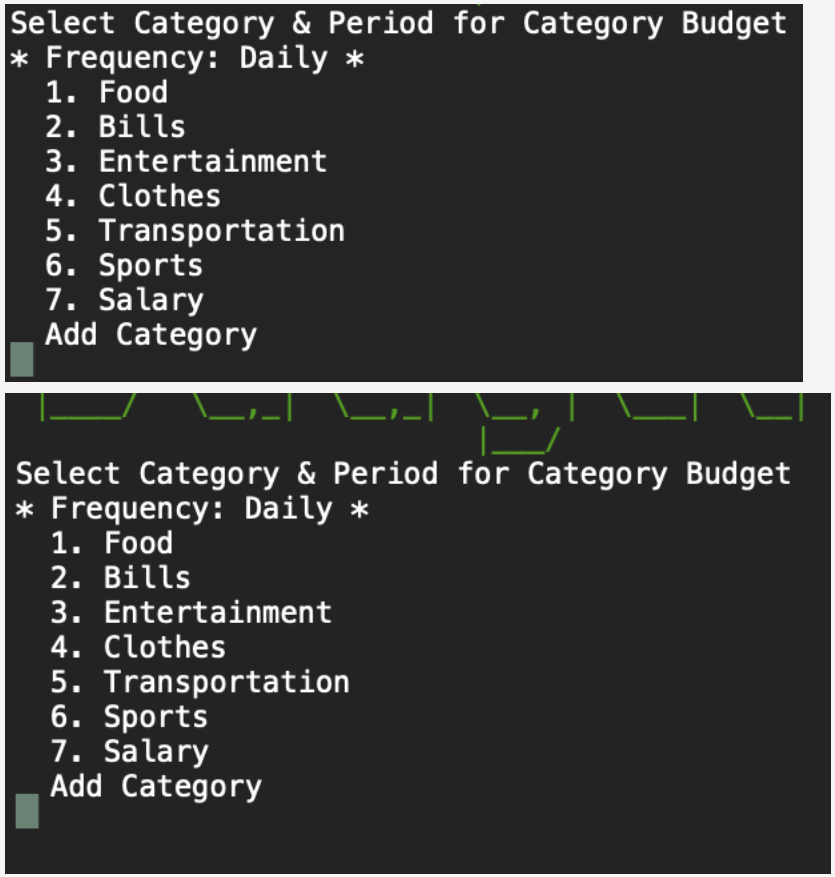
Test	Test Input Data/Action	Expected Outcome	Outcome Screenshot	Pass/ Fail
Enter overall budget - Replace/Over write existing budget	Press enter on 'Add Budget' and enter a new overall budget while one already exists	Application will successfully replace the existing overall budget with the newly entered overall budget		Pass
Enter overall budget - Invalid parameter (characters)	Press enter on 'Add Budget' and enter '31io7' for the amount	Application will reject entry and request a positive number		Pass

Test	Test Input Data/Action	Expected Outcome	Outcome Screenshot	Pass/ Fail
Enter overall budget - Invalid parameter (negative number)	Press enter on ‘Add Budget’ and enter ‘-1’ for the amount	Application will reject entry and request a positive number		Pass
Enter overall budget - Invalid parameter (0)	Press enter on ‘Add Budget’ and enter ‘0’ for the amount	Application will reject entry and request a positive number *Fail reason: Application prints a ‘budget created’ message for an entry of 0 - though this does not impact an already set budget, so does not fail the test in that regard, purely in terms of UI*		Fail
Category Budget - Allow for overspending	Set an individual category budget and enter a transaction to overspend	Application will allow the overspending to take place and reflect this accurately *Note: Decision to allow to reflect real world usage of some clients*		Pass

Test	Test Input Data/Action	Expected Outcome	Outcome Screenshot	Pass/ Fail
Overall Budget - Allow for overspending	Set an overall budget and enter a transaction to overspend	Application will allow the overspending to take place and reflect this accurately Note: Decision to allow to reflect real world usage of some clients		Pass
Overall Budget - Accurate calculations	Enter 20 as a daily overall budget, and ensure 7300 (20 x 365) is made the yearly budget	Application will show 7300 as the overall yearly budget to illustrate accurate calculations	 	Pass

Test	Test Input Data/Action	Expected Outcome	Outcome Screenshot	Pass/ Fail
Data persistence - Transactions	Create transactions, close application, restart and view transactions	Application will restart and display the same transactions as had been entered before the application was shut down		Pass

Test	Test Input Data/Action	Expected Outcome	Outcome Screenshot	Pass/ Fail
Data persistence - Budgets	Create budgets, close application, restart and view budgets	Application will restart and display the same budgets as had been entered before the application was shut down *Fail reason: Budget data did not persist after application shutdown, database did not store the information to be re-loaded*		Fail

Test	Test Input Data/Action	Expected Outcome	Outcome Screenshot	Pass/ Fail
Data persistence - Categories	Create categories, close application, restart and view categories	Application will restart and display the same categories as had been entered before the application was shut down		Pass

Testing Types

The extensive testing and tabulating process displayed above is all functional testing, which was chosen due to the menu-focused user interface of the application, and the relative simplicity of the program. The tests are centred around both program functionality and user perspective/experience. Other forms of testing (such as unit testing) were not considered relevant due to the nature of the menu interface, and the extensive work already done during the design and development process to ensure code and program functionality.

Critical Scenarios & Points of Failure

Only one main critical scenario point of failure was discovered through the testing process;

1. When creating a transaction and entering the date selection menu, if the user immediately uses the up arrows to scroll through the yearly/monthly/daily edit date options, the year will reset to 0001, and be reflected through all yearly/monthly/daily date selection menus. If the user then attempts to go below this 0001 year, the program will crash, hence is a critical point of failure.

Bugs & Errors

The bugs and errors discovered through the testing process (screenshots displayed in the preceding testing tables) are as follows;

1. Invalid parameter user entry and program display of 0 for an amount - the entry of 0 for an amount is allowed and validated in the ‘Add Income/Expense’ menu when creating a transaction, in the ‘View Transaction’ menu when editing a transaction, and in ‘Manage Categories’ when creating a budget for a category. All of these 0 values will be further displayed in the ‘View Transactions’ menu. (Note: Being able to set a value of 0 for an individual category could be useful as a way to reset/delete an individual category budget, but ideally the program would develop a separate method to delete an individual category budget instead).
2. Invalid parameter user entry (but not program display) of 0 for an amount - the entry of 0 for an amount is allowed and validated in the ‘Add Budget’ menu when creating an overall budget. However, this is rightly restricted from overwriting or resetting an existing budget, and will not display in the ‘View Transactions’ menu, so only needs to be corrected for user experience.

3. Year resets to 0001 in the date selection menu - part of the critical scenario point of failure mentioned above, when the user creates a transaction and enters the date selection menu, if using the up arrows to scroll through the yearly/monthly/daily edit date options, the year resets to 0001. If the user enters the menu and scrolls down then the date selection menu functions perfectly, and once the user has scrolled down they are free to scroll either up or down without the date resetting to 0001, making it a particularly strange bug.
4. The previous version of a transaction being displayed to the user after it's been edited - when a user enters the 'View Transaction' menu and presses enter to edit a transaction, and then edits it successfully, the program will go one menu back to the edit transactions list, but will still display the previous un-edited version of the transaction. Going back another menu level to the 'View Transactions' menu displays the updated version of the edited transaction, and confirms that editing a transaction is functioning perfectly within the program - the bug is just that through returning to the 'View Transactions' menu the previous version of the un-edited transaction is briefly displayed to the user, which could cause confusion. The suggested fix would be to return the user straight to the 'View Transactions' menu after editing a transaction, or to ensure the edit transactions menu list is updated immediately for the user.
5. Budgets have no data persistence in the attached database - it should firstly be mentioned that the functional database with persistence is an extra feature of this program, and thus the error should be seen in this ambitious context. Data persistence also functions perfectly for both transactions and categories, but budgets do not persist when the program is exited and restarted.
6. Some errors were discovered during the testing process, but still had time to be fixed via communication between the team, such as recurring transactions appearing twice for the same date, and the yearly individual category budget not appearing in the 'View Transactions' menu. These have not been displayed in the testing table for the sake of clarity, but are an example of good communication between team members to solve issues discovered during the testing process.

Potential Future Improvements & Extra Features

The following are suggestions for potential improvements and extra features, that do not involve any error or bug corrections, but could further improve the project in future with more development time;

- A toggle option to either allow or restrict the user from spending beyond an overall budget, or even an individual category budget.
- Implement a method to view all currently set budgets within one menu, rather than only in the 'View Transactions' menu once a transaction exists in that category.

- All budget allocations, whether overall or category specific, could be made in this same budget menu, with additional features of all budgets being capable of active and clear modification, resetting or deleting.
- The category management menu could exist only to add new categories, with additional features to edit category names and delete categories and related transactions.
- Additional option to input a specific date in ‘View Transactions’ menu, so the user doesn’t have to scroll to the relevant month and then scroll to the relevant day to view specific daily transactions.
- The ability for a user to view weekly transactions in the ‘View Transactions’ menu, as well as the currently existing daily, monthly and yearly transactions.
- Methods to modify or reset the yearly budgets for specific years, or the monthly budgets for the specific months, or the daily budgets for specific days - additional features that would likely be very large in scope.
- The option to delete all recurring transactions at once as a group, rather than being restricted to only deleting them one at a time once set.
- To further build upon the extra feature of the implemented database, implement a method within the application that allows the user to clear/reset the database files.
- An additional option to edit or delete notes for transactions, on top of the already provided option to create and view notes.
- A prompt for the user to press enter in order to edit transactions in the ‘View Transactions’ menu.
- Signposted user controls on the main menu (e.g. backspace to go back etc.) for those users who may not be familiar with navigating the UI in the console.
- Print a message to the user when the program has exited, for visual confirmation.

Post-Testing Analysis

The application successfully achieves all of the requirements of the project specification, including the advanced application features;

1. The application allows a user to see a list of recent transactions through the ‘View Transactions’ menu, where the user can view a list of all transactions on a given day, month or year.
2. The application allows a user to enter a new transaction through the ‘Add Income/Expense’ menu, which includes amount, income/expense toggle, the category it falls under, and the date. It also achieves the advanced application goal through allowing the user to enter or view a note attached to a transaction, and to specify whether the transaction is recurring, as well as whether this is recurring daily, monthly or yearly, and creates these recurring transactions in relation to the start and end date specified by the user.

3. The application allows a user to edit/delete transactions through pressing enter on the ‘View Transactions’ menu on either daily, monthly or yearly transactions, where the user can then select individual transactions from their requested date parameters to edit or delete.
4. The application allows a user to view a list of categories through the ‘Manage Categories’ menu, and the application comes with a preset list of five categories; Food, Bills, Entertainment, Clothes, Transportation. It also achieves the advanced application goal through allowing the user to add new categories.
5. The application allows a user to enter an overall budget through the ‘Add Budget’ menu, with a specification of daily, weekly, monthly or yearly, which is then multiplied or divided amongst the other date specifications accordingly. It also allows a user to enter specific individual category budgets, with a specification of daily, weekly, monthly or yearly, which is again then multiplied or divided amongst the other date specifications accordingly.
6. The application allows a user to track their progress against their budget by seeing how much they have spent in each category against their budget for that category, as well as the overall spending against the overall budget, through the ‘View Transactions’ menu.
7. The application goes beyond the spec of simulating a database, and instead implements the additional feature of a database with persistent storage.

Project & Testing Conclusion

Overall, it is fair to say that the application both met and exceeded the requirements of the project specification. It achieves each main goal of the specification, and further achieves each goal that would be expected of an advanced application. In addition to this, extra features have been implemented, such as the persistent database.

The testing process revealed only one critical point of failure, which would be difficult for the user to replicate through natural use of the program. The testing process also revealed some other small bugs and errors, which have been displayed in the testing tables along with all of the successful test results, and would have required more time and scope to fix beyond the project deadline. Any bugs and errors discovered during the testing process and then further fixed before the project deadline have not been displayed in the testing tables for sake of clarity. Potential further improvements and extra features illustrate the scalability and functionality of the program for additional development and expansion.

In conclusion, the design and development of the program have successfully produced a fully functional and operational expense tracker and budgeting application, which is illustrated through its durability during the rigorous testing process, passing the majority of the tests, and in achieving each main goal of the specification, both standard and advanced.

Individual Thoughts and Project Contributions

Ben Hogg (Lead Designer / Developer): Individual Report

In my role within the team, I handled both designing and developing the system. I hoped to contribute by designing solid systems and implementing them using learned design patterns, Object-Oriented Programming (OOP) principles like encapsulation and polymorphism, and industry best practices such as simplicity, decoupling, modularity, and readability.

One of the most difficult parts of the design phase was drawing up the initial core idea for the system's design. We had lots of discussions about how to organise classes and distribute objects among them. Since budgets, categories, and transactions are closely tied, we decided to simplify by giving controllers the job of managing them along with lists of instantiated objects. Eventually, we decided the Model-View-Controller (MVC) architecture would be our project's backbone. Figuring out how to implement this architecture in a console application was both rewarding and challenging. Making the MVC fit our application's needs and our team's goals was difficult, especially as we started to add dynamic, selectable menus.

Introducing dynamic selectable menus added complexity to our code design. It became clear that separating menus from instances of models was tricky. We weren't sure about this idea at first, but we came up with some clever solutions. For example, we tried using "dummy" transactions to send to a factory, or using summary methods from controllers to generate menus dynamically.

As the code base grew, sticking to the initial design and keeping good design principles intact got much harder. Looking back, I see the need for a stronger design foundation early on to make execution smoother from start to finish. This experience taught me the importance of a solid design before diving into implementation.

Despite the challenges, our team worked well together. Our meetings were productive and professional, thanks to the strength of our team. I made sure to bring problem-solving solutions to the table, listen to and collaborate with team members, and give constructive feedback on the project's strengths and weaknesses, as well as individual contributions. Maintaining a critical yet positive and professional approach was key to our teamwork.

In summary, while we faced hurdles in the design and development process, our teamwork, focus and collaborative approach helped us overcome challenges and deliver a successful project.

Ben MacKellar (Lead Developer / Designer): Individual Report

My role on the team was primarily in implementation and design, however as the ambition and large scope between the team was realised, a collaboration to all elements of the project was required to reach our goals. I feel confident at C# and have a relatively good understanding of OOP principles and good practice, and felt my strongest contributions writing good amounts of concise, readable code whilst also making suggestions to improve others code (e.g. applying LINQ queries) and trying to apply the single responsibility principle. Whilst the solution is never perfect, overall I thoroughly enjoyed working with the group and producing a functional application we are all proud of.

I enjoyed trying to implement newly learnt design patterns into the design, and the team worked really well at brainstorming how to incorporate ideas into the design and get the true value of them, rather than incorporation for the sake of it. I believe myself and the team were good at recognising this and were conscious of not over complicating areas unnecessarily.

In the beginning I emphasised we try to not code until we have agreed upon a design we are happy to progress with, aiming to avoid wasted time coding towards deadends/poor designs. This worked well with the team as it allowed design to be thorough, explored and offered learning to us all in delving into design patterns. Once a design was ready, I primarily worked on data structures and the flow of information in the beginning, i.e. Models and Controllers, whereas other developers incorporated the UI, user interaction and menu functionality. Once the complexity of the menus was realised and the difficulties faced in how they relate to the ‘back end’ I had developed, a collaborative approach was implemented successfully to ensure the link between UI and models worked correctly, and all members truly understood how the application was working. For example this involved tailoring summary methods in the controllers to receive DateTime objects to be integrated into interactive menus. I was happy with my contributions towards how the logic of how the controllers respond to transactions being added/removed, similarly with what occurs when a budget is instantiated to the app which didn’t previously have one, however the application already held transactions, for example. I also felt I made solid contributions in extracting functionality from methods in an attempt to refactor the solution, creating more readable and concise blocks of code.

It was an adjustment learning to develop software alongside other developers, initially I found difficulty in seeing a design translated differently to how I may have considered it, after new ideas were brought and discussed at the weekly team meeting. However this was quickly a non issue due to the professionalism of myself and my team, despite some disagreements as with any project, I was able to approach discussions well that were constructive and ultimately led to a better application.

Overall I believe we worked well together and each offered good communication coupled with strong problem solving ability, whilst understanding issues with the application and what we would do differently next time.

Brad Prosser (Project Manager / Tester): Individual Report

In my team role as Project Manager, my main contributions to the design and development of the application were to set objectives and goals to ensure the application progressed week by week, while also ensuring the designers and developers had the freedom to pursue their targets in the ways that worked best for them, without stifling new ideas. For example, deciding on a concept to progress by the next meeting, but then changing course at that subsequent meeting if in the process of designing and developing the application, a more efficient or better suited concept had been discovered by the designers and developers.

I would organise a team meeting at least once a week, where we would spend several hours discussing progress against previously set objectives, as well as further agenda items that I would set to continue the progress of the application, whilst the designers and developers would bring their own ideas to the table. I would oversee these group discussions on ideas, any issues encountered, and future plans, and through teamwork we would come to conclusions either as a group, or I would help to make a decision on a key matter in my capacity as project manager. I would then set further goals and objectives from these discussions for the next meeting, ensuring that the workload would be equally shared.

As project manager I also took responsibility for recording the minutes from these team meetings and writing them on the team blog, as it was helpful for me to properly manage the project's design and development progression in this way. The team worked very well together and the designers and developers were incredibly capable and driven, so my responsibility of resolving team problems was focused on ensuring we met at the best time and date for all involved, that targets and objectives were clearly set, and that developers and designers were encouraged to pursue their areas of expertise in working towards our application development.

In my team role as Tester, my main contributions to the design and development of the application were to explore every aspect and function of the completed application through an extensive series of functional tests. In this way I could ensure the durability and functionality of the application, and find any critical scenario points of failure, bugs or errors, and make suggestions as to potential future improvements and extra features. Bugs that weren't too large in scope to fix before the project deadline were communicated back to the developers for fixing, following which I would re-test the latest commit of the application and confirm the fix, contributing to the increased effectiveness of the application.

The testing table also covered every possible application use, menu or user entry, ensuring full assessment of the program's strengths and weaknesses. The application passed all tests related to achieving both the standard and advanced specification goals, while in some areas I was able to make recommendations for future improvements and application durability and usability. In large part my role as tester was to confirm the success of the design and development processes, as the application achieves all of its aims. This was especially rewarding to experience through testing, having set these objectives and milestones while overseeing the design and development processes in my other role as project manager.

Code Appendix and CS File Summary

Budget.cs: Represents a budget with properties like yearly allocation, start date, end date, and amount spent. Provides methods for daily and monthly allocations.

Category.cs: Represents a category with properties like name, budget, and spent. Provides methods for daily and monthly allocations.

TransactionController.cs: Manages transactions with methods to load, save, add, and delete transactions. Also renders transactions and retrieves transactions within a range.

CategoryController.cs: Manages categories with methods to load categories, get a category, add a new category, set a category budget, and get a summary for each category.

BudgetController.cs: Manages the budget with methods to get the budget, update the budget status, and get a breakdown of the budget.

Menu.cs: Base class for all menus with properties like name and a boolean indicating whether it can be popped. Provides a virtual Run method.

MenuStack.cs: Manages the stack of menus with methods to push, pop, and peek at menus.

MenuFeatures.cs: Provides utility methods for menus, such as printing centred text and getting user input.

SelectableMenu.cs: Represents a selectable menu with options and a selected index. Provides a Run method to display the menu and handle user input.

MenuNavigator.cs: Navigates through a selectable menu based on the user's input.
MenuDisplay.cs: Displays the options of a selectable menu.

Calendar.cs: Represents a calendar with properties like start date, end date, current scale, and format. Provides methods to navigate the calendar.

Database.cs: Manages the database with methods to load, save transactions and categories, and initialise the database.

IFileFormat.cs: Interface defining methods for loading and saving transactions and categories.

Validator.cs: Provides utility methods for validation, such as checking if a transaction is in a certain period and receiving a valid amount.

Program.cs: Entry point of the application, calling the Run method of ExpenseTrackerApp to start the application.

ExpenseTrackerApp.cs: Runs the main application loop, initialises the database, and starts the main menu.

MainMenu.cs: Represents the main menu of the application, displaying main options to the user and handling user's selection.

TransactionCreationMenu.cs: Represents the menu for creating a new transaction, allowing the user to input transaction details and creating it.

BudgetCreatorMenu.cs: Represents the menu for creating a new budget, allowing the user to input budget details and creating it.

CategorySetMenu.cs: Represents the menu for setting a category, allowing the user to input category details and set it.

CalendarMenu.cs: Represents the menu for navigating the calendar, allowing the user to navigate dates and view transactions for a specific date.

TransactionSelectionMenu.cs: Represents the menu for selecting a transaction, allowing the user to select a transaction from a list and perform actions on it.

Transaction.cs: Represents a transaction with properties like category name, amount, date, and note.

Income.cs: Represents an income transaction, inheriting from Transaction class and overriding the Display property.

Expense.cs: Represents an expense transaction, inheriting from Transaction class and overriding the Display property.

TransactionCreatorMenu.cs: Contains the TransactionCreationMenu class responsible for creating and editing transactions. It allows setting the transaction amount, date, category, type (income or expense), and handles recurring transactions.

Database.cs: Contains the Database class responsible for loading and saving transactions and categories to/from the database. It utilises the IFileFormat interface to handle different file formats.

IFileFormat.cs: Contains the IFileFormat interface defining methods for loading and saving transactions and categories. It is implemented by classes handling specific file formats.
MenuDisplay.cs: Contains the MenuDisplay class responsible for displaying menu options to the user.

SelectableMenu.cs: Contains the SelectableMenu class responsible for running a menu and handling user input to navigate through menu options.

MenuNavigator.cs: Contains the MenuNavigator class responsible for navigating through menu options based on user input.

Program.cs: Contains the Program class with the Main method, serving as the entry point of the application. It initiates the ExpenseTrackerApp.

MainMenu.cs: Contains the MainMenu class responsible for displaying the main menu of the application and handling user input to navigate through the main menu options.

BudgetCreatorMenu.cs: Contains the BudgetCreationMenu class responsible for creating a budget. It allows setting the budget amount and frequency.

Menu.cs: Contains the Menu class, serving as the base class for all menus in the application. It provides a method to run the menu.

MenuStack.cs: Contains the MenuStack class responsible for managing the stack of menus. It provides methods to push and pop menus from the stack.

MenuFeatures.cs: Contains the MenuFeatures class providing utility methods for menus, such as printing centred text and getting user input.

CalendarMenu.cs: Contains the CalendarMenu class responsible for displaying a calendar to the user and handling user input to navigate through the calendar.

TransactionSelectionMenu.cs: Contains the TransactionSelectionMenu class responsible for displaying a list of transactions to the user and handling user input to select a transaction.

```
namespace BudgetApp.Database;

public interface IFileFormat
{
    public abstract List<Transaction> Load(string filePath);
    public abstract void Save(List<Transaction> transactions, string filePath);
    public abstract List<Category> LoadCategories(string filePath);
    public abstract void SaveCategories(List<Category> categories, string filePath);
}using Newtonsoft.Json;
using System.Collections.Generic;
using System.IO;

namespace BudgetApp.Database
{
    public class JsonFormat : IFileFormat
    {
        JsonSerializerSettings settings = new JsonSerializerSettings
        {
            TypeNameHandling = TypeNameHandling.Auto,
            NullValueHandling = NullValueHandling.Ignore,
            ReferenceLoopHandling = ReferenceLoopHandling.Ignore
        };

        public List<Transaction> Load(string filePath)
        {
            string jsonContent = File.ReadAllText(filePath);
            List<Transaction> transactions =
JsonConvert.DeserializeObject<List<Transaction>>(jsonContent, settings);
            return transactions;
        }

        public void Save(List<Transaction> transactions, string filePath)
        {
            string jsonContent = JsonConvert.SerializeObject(transactions, settings);
            File.WriteAllText(filePath, jsonContent);
        }

        public List<Category> LoadCategories(string filePath)
        {
            string jsonContent = File.ReadAllText(filePath);
            List<Category> categories =
JsonConvert.DeserializeObject<List<Category>>(jsonContent, settings);
            return categories;
        }
```

```
        public void SaveCategories(List<Category> categories, string filePath)
        {
            string jsonContent = JsonConvert.SerializeObject(categories, settings);
            File.WriteAllText(filePath, jsonContent);
        }
    }
}using System;
using System.Collections.Generic;
using System.IO;
using BudgetApp;
using BudgetApp.Management;

namespace BudgetApp.Database
{
    public class Database
    {
        private static Database? instance = null;
        private IFileFormat fileFormat;
        private string transactionFilePath = Path.Combine(AppContext.BaseDirectory,
"Database/transactions.json");
        private string categoryFilePath = Path.Combine(AppContext.BaseDirectory,
"Database/categories.json");

        private Database(IFileFormat fileFormat)
        {
            this.fileFormat = fileFormat;
        }

        public static Database Instance
        {
            get
            {
                if (instance == null)
                {
                    instance = new Database(new JsonFormat());
                    instance.Initialize();
                }
                return instance;
            }
        }

        public List<Transaction> LoadTransactions()
        {
```

```
            return fileFormat.Load(transactionFilePath);
        }

        public void SaveTransactions(List<Transaction> transactions)
        {
            fileFormat.Save(transactions, transactionFilePath);
        }

        public List<Category> LoadCategories()
        {
            return fileFormat.LoadCategories(categoryFilePath);
        }

        public void SaveCategories(List<Category> categories)
        {
            fileFormat.SaveCategories(categories, categoryFilePath);
        }

        public void Initialize()
        {
            LoadCategories();

            LoadTransactions();

            SaveCategories(CategoryController.GetCategories());

            SaveTransactions(TransactionController.GetTransactions());
        }
    }
}// <autogenerated />
using System;
using System.Reflection;
[assembly:
global::System.Runtime.Versioning.TargetFrameworkAttribute(".NETCoreApp,Version=v8.0",
FrameworkDisplayName = ".NET 8.0")]
//------------------------------------------------------------------------------
// <auto-generated>
//     This code was generated by a tool.
//
//     Changes to this file may cause incorrect behavior and will be lost if
//     the code is regenerated.
// </auto-generated>
//------------------------------------------------------------------------------
```

```
using System;
using System.Reflection;

[assembly: System.Reflection.AssemblyCompanyAttribute("BudgetApp")]
[assembly: System.Reflection.AssemblyConfigurationAttribute("Debug")]
[assembly: System.Reflection.AssemblyFileVersionAttribute("1.0.0.0")]
[assembly:
System.Reflection.AssemblyInformationalVersionAttribute("1.0.0+7375b1355568f67f4c89392c7
1f65a6327767ae1")]
[assembly: System.Reflection.AssemblyProductAttribute("BudgetApp")]
[assembly: System.Reflection.AssemblyTitleAttribute("BudgetApp")]
[assembly: System.Reflection.AssemblyVersionAttribute("1.0.0.0")]

// Generated by the MSBuild WriteCodeFragment class.

// <auto-generated/>
global using global::System;
global using global::System.Collections.Generic;
global using global::System.IO;
global using global::System.Linq;
global using global::System.Net.Http;
global using global::System.Threading;
global using global::System.Threading.Tasks;
// <autogenerated />
using System;
using System.Reflection;
[assembly:
global::System.Runtime.Versioning.TargetFrameworkAttribute(".NETCoreApp,Version=v7.0",
FrameworkDisplayName = ".NET 7.0")]
//------------------------------------------------------------------------------
// <auto-generated>
//     This code was generated by a tool.
//
//     Changes to this file may cause incorrect behavior and will be lost if
//     the code is regenerated.
// </auto-generated>
//------------------------------------------------------------------------------

using System;
using System.Reflection;

[assembly: System.Reflection.AssemblyCompanyAttribute("BudgetApp")]
[assembly: System.Reflection.AssemblyConfigurationAttribute("Debug")]
[assembly: System.Reflection.AssemblyFileVersionAttribute("1.0.0.0")]
```

```
[assembly:
System.Reflection.AssemblyInformationalVersionAttribute("1.0.0+4800411528056a8c8d50ea71
04c750655fffcb96")]
[assembly: System.Reflection.AssemblyProductAttribute("BudgetApp")]
[assembly: System.Reflection.AssemblyTitleAttribute("BudgetApp")]
[assembly: System.Reflection.AssemblyVersionAttribute("1.0.0.0")]

// Generated by the MSBuild WriteCodeFragment class.

// <auto-generated/>
global using global::System;
global using global::System.Collections.Generic;
global using global::System.IO;
global using global::System.Linq;
global using global::System.Net.Http;
global using global::System.Threading;
global using global::System.Threading.Tasks;
using System;
using System.Collections.Generic;
using BudgetApp;
using BudgetApp.Database;
using BudgetApp.Menus.TrueMenus;

    public static class ExpenseTrackerApp
    {
        public static void Run()
        {
            // Initialize the Database class and load the categories from the database
            Database.Instance.Initialize();
            Budget.Initialize(0);

            // Add default categories
            var defaultCategories = new List<string> { "Food", "Bills", "Entertainment", "Clothes",
"Transportation" };
            foreach (var categoryName in defaultCategories)
            {
                if (CategoryController.GetCategory(categoryName) == null)
                {
                    var newCategory = new Category(categoryName);
                    CategoryController.GetCategories().Add(newCategory);
                }
            }

            // Save the categories to the database
```

```
            Database.Instance.SaveCategories(CategoryController.GetCategories());

            MenuStack.PushMenu(new MainMenu());
        }
    }
using System;

namespace BudgetApp
{
    public class MainMenu
    {
        DurationSelector menu3 = new DurationSelector();
        public void RunMainMenu()
        {
            //SelectableMenu mainMenu = new SelectableMenu("Welcome to the Budget App.
Please select an option:", new []{"Start", "About", "Exit"});
            //int userChoice = mainMenu.Run();

            while (true)
            {
                DisplayOptions();
                Console.WriteLine("Please select an option");
                Console.WriteLine();
                int userChoice = Convert.ToInt32(Console.ReadLine());
                switch (userChoice)
                {
                    case 1:

                        //BenDHogg to implement where dates come from
                        //TransactionController.GetAllTransactions();
                        break;
                    case 2:

                        Console.WriteLine("Budget for october set at 1000.");
                        BudgetController.CreateOverallBudget("October",new
DateTime(2024,10,01),new DateTime(2024,10,20), 1000);
                        break;
                    case 3:
                        //menu3.DisplayMenu();
                        // BudgetController.BudgetBreakdown(BudgetController.GetBudget("October"));

//BudgetController.CategorySummaryForBudget(BudgetController.GetBudget("October"));

BudgetController.CalculateOverallSpending(BudgetController.GetBudget("October"));
```

```
                break;

            case 4:
                Console.WriteLine("Please enter the category name");
                string categoryName = Console.ReadLine();
                CategoryController.CreateCategory(categoryName);

                Console.WriteLine("Would you like to set a budget for this category?");
                string response = Console.ReadLine();

                if (response.ToUpper() == "Y")
                {

CategoryController.SetCategoryBudget(CategoryController.GetCategory(categoryName));
                }
                else
                {
                    Console.WriteLine("Thanks, category added with no budget.");
                }

                    break;
            case 5:
                Console.WriteLine("Please select a category");

CategoryController.SetCategoryBudget(CategoryController.GetCategory("Food"));

                break;
            case 6:


CategoryController.SetCategoryBudget(CategoryController.GetCategory("Food"));
                CategoryController.SetCategoryBudget(CategoryController.GetCategory("Bills"));

//CategoryController.SetCategoryBudget(CategoryController.GetCategory("Entertainment"));

//CategoryController.SetCategoryBudget(CategoryController.GetCategory("Clothes"));

                Transaction food1 = new Transaction(CategoryController.GetCategory("Food"),
new DateTime(2024, 10, 02), 100, true, true);
                Transaction bills1 = new Transaction(CategoryController.GetCategory("Bills"),
new DateTime(2024, 10, 02), 100, true, true);
```

```
                Transaction grub1 = new
Transaction(CategoryController.GetCategory("Entertainment"), new DateTime(2024, 10, 02),
100, true, true);
                Transaction clothes1 = new
Transaction(CategoryController.GetCategory("Clothes"), new DateTime(2024, 10, 03), 100, true,
true);

                TransactionController.AddTransactions(food1);
                TransactionController.AddTransactions(bills1);
                TransactionController.AddTransactions(grub1);
                TransactionController.AddTransactions(clothes1);

                BudgetController.UpdateBudgetStatus(food1);
                BudgetController.UpdateBudgetStatus(bills1);
                BudgetController.UpdateBudgetStatus(grub1);
                BudgetController.UpdateBudgetStatus(clothes1);



                break;
            case 7:

                Console.WriteLine("Categories List ");
                Console.WriteLine("-------------");
                int n = 1;
                foreach (var category in CategoryController.categories)
                {

                    Console.WriteLine($"{n}. {category.name}");
                    n++;
                }
                Console.WriteLine("------------");
                Console.WriteLine();
                break;

            default:
                Console.WriteLine("Invalid choice");
                break;
        }
    }

}

public static void DisplayOptions()
```

```
        {
            Console.WriteLine("1. See All Transactions");
            Console.WriteLine("2. Set Budget");
            Console.WriteLine("3. View Budget");
            Console.WriteLine("4. Add Category");
            Console.WriteLine("5. Set Budget for Category");
            Console.WriteLine("6. Add Transaction");
            Console.WriteLine("7. See all categories");
        }
    }
}using BudgetApp;

namespace BudgetApp;


public class OpeningMenu
{
    SelectableMenu _menu = new SelectableMenu("Welcome to BudgetApp! Please select an
option:", new string[] { "start", "about", "Exit" });

    public void RunMenu()
    {
        int selection = _menu.NavigateMenu();
        switch (selection)
        {
            case 0:
                DurationSelector menu3 = new DurationSelector();
                menu3.DisplayMenu();

                break;
            case 1:
                Console.WriteLine("About BudgetApp...");
                break;
            case 2:
                Console.WriteLine("Exiting BudgetApp...");
                break;
        }
    }
}namespace BudgetApp;

public class BudgetSetMenu
{
    SelectableMenu _menu = new SelectableMenu("Please select an option:", new string[] { "Set
Budget", "Clear Budget", "Back" });
```

```
    public void RunMenu()
    {
        int selection = _menu.NavigateMenu();
        switch (selection)
        {
            case 0:
                Console.WriteLine("Set Budget...");
                break;
            case 1:
                Console.WriteLine("Clear Budget...");
                break;
            case 2:
                Console.WriteLine("Back...");
                break;
        }
    }

}namespace BudgetApp;

public static class MenuStack
{
    public static Stack<object> _menuStack = new Stack<object>();

    public static void Push(object menu)
    {
        _menuStack.Push(menu);
    }

    public static object Pop()
    {
        return _menuStack.Pop();
    }

    public static object Peek()
    {
        return _menuStack.Peek();
    }
}using System.Diagnostics;
using BudgetApp.Management;
using static BudgetApp.Calendar;

namespace BudgetApp;
```

```
public static class BudgetController
{
    public static Budget GetBudget() => Budget.Instance;

    public static void UpdateBudgetStatus()
    {
        Budget.Instance.amountSpent = 0;
        foreach (var transaction in TransactionController.GetTransactions())
        {
            if (transaction.GetType() == typeof(Expense))
            {
                Budget.Instance.amountSpent += transaction.Amount;
            }
        }
    }

    public static string BudgetSummary(DateTime start, DateTime end, Calendar calendar)
    {
        double amountSpent = 0;
        int n = 1;
        foreach (var transaction in TransactionController.GetTransactions())
        {
            if (Validator.TransactionInPeriod(transaction.Date, start, end) & transaction.GetType() ==
typeof(Expense))
            {
                amountSpent += transaction.Amount;
            }
        }
        switch (calendar._currentScale)
        {
            case Calendar.DurationScale.Yearly:
                return $"Total spent {amountSpent} / {Budget.Instance.yearlyAllocation}";
            case Calendar.DurationScale.Monthly:
                return $"Total spent {amountSpent} / {Budget.Instance.GetMonthlyAllocation()}";
            case Calendar.DurationScale.Daily:
                return $"Total spent {amountSpent} / {Budget.Instance.GetDailyAllocation()}";
        }
        return "No Overall Budget Set";
    }
}
using System;
namespace BudgetApp;
public static class Validator
{
```

```
    public static bool TransactionInPeriod(DateTime transactionDate, DateTime start, DateTime
end) => transactionDate >= start && transactionDate <= end;

    public static double ReceiveValidAmount()
    {
        while (true)
        {
            Console.Write("Enter amount: ");
            var amount = Console.ReadLine();
            if (double.TryParse(amount, out double result) && double.IsPositive(result))
            {
                return Math.Round(result, 2);
            }
            Console.WriteLine("Please enter a positive number");
        }
    }
}


using System;
using BudgetApp.Database;
using BudgetApp;
using System.Transactions;

using BudgetApp.Management;
public static class CategoryController
{
    private static List<Category> categories;

    public static List<Category> Categories
    {
        get
        {
            if (categories == null)
            {
                categories = Database.Instance.LoadCategories();
            }
            return categories;
        }
    }
    public static Category GetCategory(string categoryName)
    {
        if (Categories == null)
        {
```

```
            throw new Exception("Categories is null. Make sure to load the categories before
accessing them.");
        }

        return Categories.Find(c => c.name == categoryName);
    }

    public static List<Category> GetCategories() => Categories;

    public static string[] GetCategoryNamesPlusAddOption()
    {
        int n = 1;
        string[] categoryNames = new string[categories.Count + 1];
        for (int i = 0; i < categories.Count; i++)
        {
            categoryNames[i] = $"{n}. {categories[i].name}";
            n++;
        }
        categoryNames[categoryNames.Length - 1] = "Add Category";
        return categoryNames;
    }

    public static void UpdateCategoryAllowance(Transaction newTransaction) =>
newTransaction.Category.spent += newTransaction.Amount;

    public static Category ReceiveNewCategory()
    {
        while (true)
        {
            Console.WriteLine("Enter a new category name:");
            string categoryName = Console.ReadLine();
            if (!string.IsNullOrWhiteSpace(categoryName) && !categories.Any(c =>
c.name.ToUpper() == categoryName.ToUpper()))
            {
                Console.WriteLine($"New Category {categoryName} created!");
                Console.WriteLine("Press any key to continue");
                Console.ReadKey();
                categories.Add(new Category(categoryName));

                Database.Instance.SaveCategories(categories);

                return GetCategory(categoryName);
            }
            else
```

```
            {
                Console.WriteLine("Please type a valid category, that does not already exist.");
            }
        }
    }

    public static void SetCategoryBudget(Category category, int frequency)
    {
        Console.WriteLine($"How much would you like to allocate to {category.name}");
        double budget = Validator.ReceiveValidAmount();
        switch (frequency)
        {
            case 1:
                category.budget = budget*365;
                break;
            case 7:
                category.budget = budget * 52; ;
                break;
            case 30:
                category.budget = budget* 12;
                break;
            case 365:
                category.budget = budget;
                break;
        }
        Console.WriteLine($"Budget of {budget} set for {category.name}");
        Console.WriteLine("Press any key to continue");
        Console.ReadKey();
        //Update with existing transactions once set
        foreach (var transaction in TransactionController.GetTransactions())
        {
            if (transaction.Category == category)
            {
                category.spent += transaction.Amount;
            }
        }
    }


    public static List<string> CategorySummary(DateTime start, DateTime end, Calendar
calendar)
    {
        Dictionary<string, double> summaryPerCategory = new Dictionary<string, double>();
```

```
        //Each category summarised in a dictionary (name (key) : spending (value))
        foreach (var category in CategoryController.categories)
        {
            summaryPerCategory[category.name] = 0;
        }

        //Get spending per period selected from Menu
        foreach (var transaction in TransactionController.GetTransactions())
        {
            if (Validator.TransactionInPeriod(transaction.Date, start, end) && transaction.GetType()
== typeof(Expense))
            {
                summaryPerCategory[transaction.Category.name] += transaction.Amount;
            }
        }

        List<string> categorySummaries = new List<string>();

        foreach (var KvP in summaryPerCategory)
        {
            if (CategoryController.categories.Any(c => c.budget != null && c.name == KvP.Key))
            {
                if (KvP.Value > 0)
                {
                    double roundedValue = Math.Round(KvP.Value, 2);

                    switch (calendar._currentScale)
                    {
                        case Calendar.DurationScale.Yearly:
                            categorySummaries.Add($"Total spent on {KvP.Key}: £{roundedValue} /
£{(float)CategoryController.GetCategory(KvP.Key).budget} (yearly)");
                            break;
                        case Calendar.DurationScale.Monthly:
                            categorySummaries.Add($"Total spent on {KvP.Key}: £{roundedValue} /
£{(float)CategoryController.GetCategory(KvP.Key).GetMonthlyAllocation()} (monthly)");
                            break;
                        case Calendar.DurationScale.Daily:
                            categorySummaries.Add($"Total spent on {KvP.Key}: £{roundedValue} /
£{(float)CategoryController.GetCategory(KvP.Key).GetDailyAllocation()} (daily)");
                            break;
                    }
                }

            }
```

```
            else
            {
                if (KvP.Value > 0)
                {
                    double roundedValue = Math.Round(KvP.Value, 2);
                    categorySummaries.Add($"Total spent on {KvP.Key}: £{roundedValue}");
                }
            }
        }
        return categorySummaries;
    }
}


using System.Text;

namespace BudgetApp.Management
{
    using BudgetApp;
    using BudgetApp.Database;
    using BudgetApp.Menus;

    public static class TransactionController
    {
        private static List<Transaction> transactions;

        private static void LoadTransactions()
        {
            transactions = Database.Instance.LoadTransactions();
        }

        public static List<Transaction> GetTransactions()
        {
            if (transactions == null)
            {
                LoadTransactions();
            }
            return transactions;
        }

        public static void SaveTransactions()
        {
            Database.Instance.SaveTransactions(transactions);
        }
```

```
    public static void AddNoteToTransaction(Transaction transaction, string note)
    {
        transaction.Note = note;
        SaveTransactions();
    }

    public static string TransactionSummary(DateTime start, DateTime end)
    {
        List<Transaction> list = GetTransactionsWithinRange(start, end);
        StringBuilder sb = new StringBuilder();
        int i = 1;
        foreach (var transaction in list)
        {
            string transactionDisplay = $"{i}. {transaction.Display}";
            sb.AppendLine(transactionDisplay);
            i++;
        }
        if (list.Count == 0)
        {
            sb.AppendLine("No transactions for this period");
        }
        return sb.ToString();
    }

    public static List<Transaction> GetTransactionsWithinRange(DateTime start, DateTime
end) => transactions.Where(c => c.Date >= start && c.Date <= end).ToList();

    public static Transaction[] GetTransactionsWithinRangeAsArray(DateTime start, DateTime
end) =>
        transactions.Where(c => c.Date >= start && c.Date <= end).ToArray();


    public static void AddTransaction(Transaction newTransaction)
    {
        if (newTransaction.Category == null)
        {
            throw new ArgumentException("Transaction must have a valid category.");
        }

        transactions.Add(newTransaction);
        CategoryController.UpdateCategoryAllowance(newTransaction);
        if (BudgetController.GetBudget() != null)
        {
```

```
            BudgetController.UpdateBudgetStatus();
        }
        SaveTransactions();
    }

    public static void DeleteTransaction(Transaction transaction)
    {
        transaction.Category.spent -= transaction.Amount;
        transactions.Remove(transaction);
        if (BudgetController.GetBudget() != null)
        {
            BudgetController.UpdateBudgetStatus();
        }

        SaveTransactions();
    }
  }
}
using BudgetApp;
using Newtonsoft.Json;
public class Transaction
{
  public string CategoryName { get; set; }
  public double Amount { get; set; }
  public DateTime Date = DateTime.Today;
  public string Note { get; set; } // Added nullable note attribute

  [JsonIgnore]
  public Category Category
  {
    get
    {
      return CategoryController.GetCategory(CategoryName);
    }
  }

  public Transaction(string categoryName, DateTime month, double amount, string note = null)
  {
    if (string.IsNullOrEmpty(categoryName))
    {
      throw new ArgumentNullException(nameof(categoryName));
    }

    CategoryName = categoryName;
```

```
        Amount = amount;
        Date = month;
        Note = note; // Assign the note
    }

    public virtual string Display
    {
        get
        {
            if (Category == null)
            {
                throw new NullReferenceException("Category object is null.");
            }
            return $"{Category.name}: £{Amount} on {Date.ToShortDateString()}";
        }
    }
}
namespace BudgetApp
{
    public class Budget
    {
        public double yearlyAllocation { get; set; }
        public double amountSpent { get; set; }


        private Budget(double totalAllocatedAmount)
        {
            this.yearlyAllocation = totalAllocatedAmount;
        }

        private static Budget _instance;

        public static Budget Instance
        {
            get
            {
                if (_instance == null)
                {
                    throw new Exception("Budget instance has not been initialized.");
                }
                return _instance;
            }
        }
```

```
        public static void Initialize(double totalAllocatedAmount)
        {
            if (_instance != null)
            {
                throw new Exception("Budget instance has already been initialized.");
            }
            _instance = new Budget(totalAllocatedAmount);
        }

        public double GetDailyAllocation() => Math.Round((yearlyAllocation / 365), 2);
        public double GetMonthlyAllocation() => Math.Round((yearlyAllocation / 12), 2);
    }
}namespace BudgetApp;

public class Income : Transaction
{
        public Income(string categoryName, DateTime month, double amount) :
base(categoryName, month, amount)
        {
        }

        public override string Display => $"[green]{Category.name}: + £{Amount} on
{Date.ToShortDateString()}[/]";
}

namespace BudgetApp;

public class Expense : Transaction
{
        public Expense(string categoryName, DateTime month, double amount) :
base(categoryName, month, amount)
        {
        }

        public override string Display => $"[red]{Category.name}: - £{Amount} on
{Date.ToShortDateString()}[/]";
}

namespace BudgetApp
{
    using System;
    using System.Collections.Generic;

    public class Category
```

```
    {
        public string name { get; set; }
        public double? budget { get; set; }
        public double spent { get; set; }

        // JSON objects require a parameterless constructor
        public Category() { }

        public Category(string name)
        {
            this.name = name;
            budget = null;
        }

        public double GetDailyAllocation() => Math.Round((double)(budget/365),2);
        public double GetMonthlyAllocation() => Math.Round((double)(budget / 12), 2);
    }
}

using BudgetApp.Management;

namespace BudgetApp.Menus.TrueMenus
{
    public class TransactionFactory
    {
        private static TransactionFactory _instance;

        public static TransactionFactory GetInstance()
        {
            if (_instance == null)
            {
                _instance = new TransactionFactory();
            }

            return _instance;
        }

        public Transaction CreateDummyTransaction(string categoryName, DateTime date, double
amount) => new Transaction(categoryName, date, amount);

        public Transaction CreateTransaction(string categoryName, DateTime date, double
amount, bool isIncome)
        {
```

```
        Transaction newTransaction = isIncome ? new Income(categoryName, date, amount) :
new Expense(categoryName, date, amount);
        TransactionController.AddTransaction(newTransaction);
        return newTransaction;
    }

    public void CreateRecurringTransaction(string categoryName, DateTime date, double
amount, bool isIncome,
        DateTime endOfRecurrence, int frequency)
        {
        TimeSpan timeSpan = endOfRecurrence - date;
        double totalDays = timeSpan.TotalDays;
        int amountOfTransactionsTotal = (int)(totalDays / frequency);
        Transaction newTransaction = CreateTransaction(categoryName, date, amount,
isIncome);

        if (frequency == 1 || frequency == 7)
        {
            for (int i = 1; i <= amountOfTransactionsTotal; i++)
            {
                DateTime nextTransactionDate = newTransaction.Date.AddDays(frequency * i);
                CreateTransaction(categoryName, nextTransactionDate,
                    newTransaction.Amount, isIncome);
            }
        }

        if (frequency == 30)
        {
            for (int i = 1; i <= amountOfTransactionsTotal; i++)
            {
                DateTime nextTransactionDate = newTransaction.Date.AddMonths(i);
               CreateTransaction(categoryName, nextTransactionDate,
                    newTransaction.Amount, isIncome);
            }
        }

        if (frequency == 365)
        {
            for (int i = 1; i <= amountOfTransactionsTotal; i++)
            {
                DateTime nextTransactionDate = newTransaction.Date.AddYears(i);
                CreateTransaction(categoryName, nextTransactionDate,
                    newTransaction.Amount, isIncome);
            }
```

```
            }
        }
    }
}using System;
using BudgetApp.Menus.TrueMenus;
using Spectre.Console;

namespace BudgetApp
{
    public class MainMenu : Menu
    {
        private SelectableMenu selectableMenu = new SelectableMenu("Main Menu", new string[]
{"View Transactions", "Add Income/Expense", "Add Budget", "Manage Categories", "Exit"});

        public MainMenu() : base("Main Menu")
        {
            CanBePopped = false;
        }

        public override void Run()
        {
            int selection = selectableMenu.Run();
            switch (selection)
            {
                case 0:
                    MenuStack.PushMenu(new CalendarMenu());
                    break;
                case 1:
                    MenuStack.PushMenu(new TransactionCreationMenu());
                    break;
                case 2:
                    MenuStack.PushMenu(new BudgetCreationMenu());
                    break;
                case 3:
                    MenuStack.PushMenu(new CategorySetMenu());
                    break;
                case 4:
                    Environment.Exit(0);
                    break;
            }
        }
    }
}using System;
using System.Transactions;
```

```
namespace BudgetApp.Menus.TrueMenus
{
    public class CategorySetMenu : Menu
    {
        private SelectableMenu selectableMenu;
        private Frequency _frequency = Frequency.Daily;

        enum Frequency
        {
            Daily = 1,
            Weekly = 7,
            Monthly = 30,
            Yearly = 365,
        }
        public CategorySetMenu() : base("Category Setting Menu")
        {
            selectableMenu = new SelectableMenu("Select Category & Period for Category Budget",
GetOptions());
        }

        public override void Run()
        {
            bool categorySet = false;

            while (!categorySet)
            {
                selectableMenu.Options = GetOptions();
                int selectedIndex = selectableMenu.Run();
                HandleSelection(selectedIndex);
            }
        }

        private string[] GetOptions()
        {
            var categories = CategoryController.GetCategories();
            string[] names = new string[categories.Count + 2];

            names[0] = "Frequency: " + _frequency;

            int i = 1;
            foreach (var category in categories)
            {
                names[i] = $"{i}. {category.name}";
```

```
            i++;
        }

        names[names.Length - 1] = "Add Category";

        return names;
    }

    private string DisplayBoolean(bool value) => value ? "Yes" : "No";

    private void HandleSelection(int selectedIndex)
    {
        int categoryCount = CategoryController.GetCategories().Count;

        if (selectedIndex == 0)
        {
            UpdateFrequency();
        }
        else if (selectedIndex == categoryCount + 1)
        {
            CategoryController.ReceiveNewCategory();
            MenuStack.PopMenu();
        }
        else
        {
            Category category = CategoryController.GetCategories()[selectedIndex - 1]; // Adjust
index for categories
            CategoryController.SetCategoryBudget(category, (int)_frequency);
            MenuStack.PopMenu();
        }
    }



    private void UpdateFrequency()
    {
        switch (_frequency)
        {
            case Frequency.Daily:
                _frequency = Frequency.Weekly;
                break;
            case Frequency.Weekly:
                _frequency = Frequency.Monthly;
                break;
```

```
                case Frequency.Monthly:
                    _frequency = Frequency.Yearly;
                    break;
                case Frequency.Yearly:
                    _frequency = Frequency.Daily;
                    break;
                default:
                    _frequency = Frequency.Daily;
                    break;
            }
        }
    }
}

using BudgetApp.Management;
using BudgetApp.Menus.TrueMenus;


namespace BudgetApp;

public class TransactionSelectionMenu : Menu
{
    private SelectableMenu _selectableMenu;
    private Transaction[] transactions;

    public TransactionSelectionMenu(string name, Transaction[] transactions) : base(name)
    {
        this.transactions = transactions;
    }

    private string[] GetOptions()
    {
        string[] options = new string[transactions.Length];
        for (int i = 0; i < transactions.Length; i++)
        {
            options[i] = transactions[i].Display;
        }
        return options;
    }

    private void FillTransactionMenu()
    {
        _selectableMenu = new SelectableMenu("Select Transaction to Edit:", GetOptions());
    }
```

```
    private void HandleSelection(int selectedIndex)
    {
        Transaction selectedTransaction = transactions[selectedIndex];
        Console.WriteLine("You have selected: " + selectedTransaction.Display);
        SelectableMenu decisionMenu = new SelectableMenu("What would you like to do?", new
string[] {"Edit", "Delete", "View Note", "Back"});
        int decision = decisionMenu.Run();
        switch (decision)
        {
            case 0:
                MenuStack.PushMenu(new TransactionCreationMenu(selectedTransaction));
                break;
            case 1:
                TransactionController.DeleteTransaction(selectedTransaction);
                Console.WriteLine("Transaction Deleted!");
                Console.ReadKey();
                MenuStack.PopMenu();
                break;
            case 2:
                if (selectedTransaction.Note == null)
                {
                    Console.WriteLine("No note found for this transaction.");
                    Console.WriteLine("Enter a note for the transaction:");
                    string note = Console.ReadLine();
                    TransactionController.AddNoteToTransaction(selectedTransaction, note);
                    Console.WriteLine("Note added!");
                    Console.ReadKey();
                }
                else
                {
                    Console.WriteLine("Note: " + selectedTransaction.Note);
                    Console.ReadKey();
                }
                break;
            case 3:
                Run();
                break;
        }
    }

    public override void Run()
    {
        while (true)
```

```
        {
            FillTransactionMenu();
            int selectedIndex = _selectableMenu.Run();
            HandleSelection(selectedIndex);
        }
    }

}using System.Text;
using BudgetApp.Management;
using BudgetApp.Menus;
using Spectre.Console;

namespace BudgetApp
{
    public class CalendarMenu : Menu
    {
        private readonly Calendar calendar = new Calendar();

        public CalendarMenu() : base("Calendar")
        {
        }

        public override void Run()
        {
            while (true)
            {
                DisplayMenu();
                NavigateMenu();
            }
        }

        private void NavigateMenu()
        {
            var keyPressed = MenuFeatures.GetInput();
            if (keyPressed == ConsoleKey.Backspace)
            {
                MenuStack.PopMenu();
            }
            else if (keyPressed == ConsoleKey.Enter)
            {
                var transactions =
TransactionController.GetTransactionsWithinRangeAsArray(calendar.Start, calendar.End);
                if (transactions.Length > 0)
                {
```

```
                MenuStack.PushMenu(new TransactionSelectionMenu("Transaction Selection
Menu", transactions));
            }
        }
        else
        {
            calendar.NavigateCalendar(keyPressed, "yyyy/MM", "yyyy", "yyyy/MM/dd");
        }
    }

    private void DisplayMenu()
    {
        Console.Clear();
        var asciiArt = new FigletText("Budget App");
        AnsiConsole.Render(asciiArt.Color(Color.Green));

        var table1 = new Table() { Border = TableBorder.Rounded };
        table1.AddColumn(new TableColumn("[yellow]" + calendar._currentScale + " " +
calendar.Start.ToString(calendar.Format) + "[/]").Centered());
        table1.Width(75);
        AnsiConsole.Render(table1);

        var table2 = new Table() { Border = TableBorder.Rounded };
        table2.AddColumn(new TableColumn("Transaction Summary").LeftAligned());
        table2.AddColumn(new TableColumn(RenderBudgetBreakdown()).LeftAligned());
        table2.AddRow(new Markup(RenderTransactionSummary()).LeftJustified(), new
Markup(RenderCategorySummary()).Centered());
        table2.Width(75);
        AnsiConsole.Render(table2);
    }


    private string RenderTransactionSummary()
    {
        return TransactionController.TransactionSummary(calendar.Start, calendar.End);
    }

    private string RenderBudgetBreakdown()
    {
        if (Budget.Instance.yearlyAllocation == 0)
        {
            return "No overall budget set.";
        }
        else
```

```
            {
                return BudgetController.BudgetSummary(calendar.Start, calendar.End, calendar);
            }
        }

        private string RenderCategorySummary()
        {
            var sb = new StringBuilder();
            foreach (var summary in CategoryController.CategorySummary(calendar.Start,
calendar.End, calendar))
            {
                sb.AppendLine(summary);
            }
            return sb.Length == 0 ? "No categories to display" : sb.ToString();
        }
    }
}using System;
using System.Data;

namespace BudgetApp
{
    public class BudgetCreationMenu : Menu
    {
        private Budget budget;
        private SelectableMenu selectableMenu;
        private Calendar calendar;
        private double amount { get; set; }
        private string dateNavigationInstructions = "Use arrow keys up and down to change scale.
Use left and right to adjust date. Press Enter to confirm date.";

        private Frequency _frequency = Frequency.Daily;

        enum Frequency
        {
            Daily = 1,
            Weekly = 7,
            Monthly = 30,
            Yearly = 365,
        }

        private void ChangeFrequency()
        {
            switch (_frequency)
            {
```

```
            case Frequency.Daily:
                _frequency = Frequency.Weekly;
                break;
            case Frequency.Weekly:
                _frequency = Frequency.Monthly;
                break;
            case Frequency.Monthly:
                _frequency = Frequency.Yearly;
                break;
            case Frequency.Yearly:
                _frequency = Frequency.Daily;
                break;
        }
    }

    public BudgetCreationMenu() : base("Budget Creation Menu")
    {
        budget = BudgetController.GetBudget();
        selectableMenu = new SelectableMenu("Create Budget:", GetOptions());
    }

    public override void Run()
    {
        bool budgetCreated = false;

        while (!budgetCreated)
        {
            selectableMenu.Options = GetOptions();
            int selectedIndex = selectableMenu.Run();
            budgetCreated = HandleSelection(selectedIndex);
        }
    }

    private string[] GetOptions()
    {
        return new string[]
        {
            "Timeframe: " + _frequency,
            "Amount : " + amount,
            "Create Budget"
        };
    }

    private string DisplayBoolean(bool value) => value ? "Yes" : "No";
```

```
private bool HandleSelection(int selectedIndex)
{
    switch (selectedIndex)
    {
        case 0:
            ChangeFrequency();
            break;
        case 1:
            SetAmount(budget);
            break;
        case 2:
            CreateBudget();
            break;
    }
    return false;
}

private double SetAmount(Budget budget)
{
    Console.Clear();
    amount = Validator.ReceiveValidAmount();

    switch (_frequency)
    {
        case Frequency.Daily:
            BudgetController.GetBudget().yearlyAllocation = amount*365;
            return amount;
        case Frequency.Weekly:
            BudgetController.GetBudget().yearlyAllocation = amount * 52;
            return amount;
        case Frequency.Monthly:
            BudgetController.GetBudget().yearlyAllocation = amount * 12;
            return amount;
        case Frequency.Yearly:
            BudgetController.GetBudget().yearlyAllocation = amount;
            return amount;
    }
    return amount;
}

private void CreateBudget()
{
    Console.WriteLine("Budget created! Press any key to continue");
```

```
            Console.ReadKey();
            MenuStack.PopMenu();
        }
    }
}using System;
using BudgetApp;
using BudgetApp.Menus;
using BudgetApp.Menus.TrueMenus;

using BudgetApp.Database;
using BudgetApp.Management;

public class TransactionCreationMenu : Menu
{
    private Calendar _transactionDateCalendar = new Calendar(DateTime.Today, "dd/MM/yyyy");
    private Calendar _calendarForEndOfRecurrance = new Calendar(DateTime.Today,
"dd/MM/yyyy");
    private string _dummyCategoryName = "Food";
    private Transaction _dummyTransaction;
    private SelectableMenu _selectableMenu;
    private bool _isIncome, _isRecurring, _editMode;
    private Frequency _frequency = Frequency.Daily;

    enum Frequency
    {
        Daily = 1,
        Weekly = 7,
        Monthly = 30,
        Yearly = 365,
    }

    public TransactionCreationMenu() : base("Transaction Creation Menu")
    {
        TransactionFactory transactionFactory = TransactionFactory.GetInstance();
        _dummyTransaction =
transactionFactory.CreateDummyTransaction(_dummyCategoryName, DateTime.Today, 0);
        _editMode = false;
        _selectableMenu = new SelectableMenu("Create Transaction:", GetOptions());
    }

    public TransactionCreationMenu(Transaction transactionToBeEdited) : base("Transaction
Editor Menu")
    {
        CanBePopped = false;
```

```
        _editMode = true;
        TransactionFactory transactionFactory = TransactionFactory.GetInstance();
        _dummyTransaction =
transactionFactory.CreateDummyTransaction(transactionToBeEdited.CategoryName,
transactionToBeEdited.Date, transactionToBeEdited.Amount);
        _dummyCategoryName = transactionToBeEdited.CategoryName;
        _selectableMenu = new SelectableMenu("Edit Transaction:", GetOptions());
        TransactionController.DeleteTransaction(transactionToBeEdited);
    }

    private void CreateOrEditTransaction()
    {
        TransactionFactory transactionFactory = TransactionFactory.GetInstance();
        if (_isRecurring)
        {
            transactionFactory.CreateRecurringTransaction(_dummyCategoryName,
                _transactionDateCalendar.Start, _dummyTransaction.Amount, _isIncome,
                _calendarForEndOfRecurrance.Start, (int)_frequency);
        }
        else
        {
            transactionFactory.CreateTransaction(_dummyCategoryName,
_transactionDateCalendar.Start,
                _dummyTransaction.Amount, _isIncome);
        }
        Console.WriteLine(_editMode ? "Transaction Edited!" : "Transaction Created!");
        Console.ReadKey();

        Database.Instance.SaveCategories(CategoryController.GetCategories());
        Database.Instance.SaveTransactions(TransactionController.GetTransactions());

        MenuStack.PopMenu();
    }

    private void UpdateAmount()
    {
        Console.Clear();
        _dummyTransaction.Amount = Validator.ReceiveValidAmount();
    }

    private void UpdateDate(Calendar calendar)
    {
        Console.WriteLine("Use arrow keys to navigate the calendar and Enter to select a date.");
        Console.WriteLine("Press Enter to confirm the selected date.");
```

```
        while (true)
        {
            Console.Clear();

            string alternateScale = GetAlternateScale(calendar._currentScale);

            Console.WriteLine("Change by " + alternateScale + "...");
            Console.WriteLine("Selected Date:");
            DisplayCalendar(calendar);

            ConsoleKey keyPressed = Console.ReadKey(true).Key;
            calendar.NavigateCalendar(keyPressed, "yyyy/MM", "yyyy", "yyyy/MM/dd");

            if (keyPressed == ConsoleKey.Enter)
            {
                calendar.Format = "dd/MM/yyyy";
                break;
            }
        }
        _selectableMenu.Options = GetOptions();
    }

    private string GetAlternateScale(Calendar.DurationScale scale)
    {
        return scale switch
        {
            Calendar.DurationScale.Yearly => "year",
            Calendar.DurationScale.Monthly => "month",
            Calendar.DurationScale.Daily => "day",
            _ => "year"
        };
    }

    private void DisplayCalendar(Calendar calendar)
    {
        Console.WriteLine(calendar.Start.ToString(calendar.Format));
    }

    private void SetCategory()
    {
        var categories = CategoryController.GetCategories();
        var selectedIndex = new SelectableMenu("Select a category:",
CategoryController.GetCategoryNamesPlusAddOption()).Run();
```

```
        if (selectedIndex == categories.Count)
        {
            _dummyCategoryName = CategoryController.ReceiveNewCategory().name;
        }
        else
        {
            _dummyCategoryName = categories[selectedIndex].name;
        }
    }

    private string DisplayRecurrenceBoolean(bool value) => value ? "Yes" : "No";

    private string DisplayIncomeBoolean(bool value) => value ? "Income" : "Expense";

    private string[] GetOptions()
    {
        var options = new List<string>
        {
            "Amount: " + _dummyTransaction.Amount,
            "Date: " + _transactionDateCalendar.Start.ToString(_transactionDateCalendar.Format),
            "Category: " + _dummyCategoryName,
            "Expense/Income: " + DisplayIncomeBoolean(_isIncome),
            "Recurring: " + DisplayRecurrenceBoolean(_isRecurring),
        };

        if (_isRecurring)
        {
            options.Add("Frequency: " + _frequency);
            options.Add("End of Recurrence: " +
_calendarForEndOfRecurrance.Start.ToString(_calendarForEndOfRecurrance.Format));
        }

        options.Add(_editMode ? "Edit Transaction" : "Create Transaction");
        return options.ToArray();
    }

    private void HandleSelection(int selectedIndex)
    {
        switch (selectedIndex)
        {
            case 0:
                UpdateAmount();
                break;
```

```
            case 1:
            case 6:
                UpdateDate(selectedIndex == 1 ? _transactionDateCalendar :
_calendarForEndOfRecurrance);
                break;
            case 2:
                SetCategory();
                break;
            case 3:
                _isIncome = !_isIncome;
                break;
            case 4:
                _isRecurring = !_isRecurring;
                break;
            case 5:
                if (!_isRecurring) CreateOrEditTransaction();
                else
                    UpdateFrequency();
                break;
            case 7:
                CreateOrEditTransaction();
                break;
        }
    }

    public override void Run()
    {
        while (true)
        {
            _selectableMenu.Options = GetOptions();
            int selectedIndex = _selectableMenu.Run();
            HandleSelection(selectedIndex);
        }
    }

    private void UpdateFrequency()
    {
        switch (_frequency)
        {
            case Frequency.Daily:
                _frequency = Frequency.Weekly;
                break;
            case Frequency.Weekly:
                _frequency = Frequency.Monthly;
```

```
                break;
            case Frequency.Monthly:
                _frequency = Frequency.Yearly;
                break;
            case Frequency.Yearly:
                _frequency = Frequency.Daily;
                break;
            default:
                _frequency = Frequency.Daily;
                break;
        }
    }
}
using BudgetApp.Menus;

namespace BudgetApp
{
    public class MenuNavigator
    {
        public int Navigate(ConsoleKey keyPressed, string[] options, int selectedIndex)
        {
            if (keyPressed == ConsoleKey.UpArrow)
            {
                selectedIndex--;
                if (selectedIndex == -1)
                {
                    selectedIndex = options.Length - 1;
                }
            }
            else if (keyPressed == ConsoleKey.DownArrow)
            {
                selectedIndex++;
                if (selectedIndex == options.Length)
                {
                    selectedIndex = 0;
                }
            }
            else if (keyPressed == ConsoleKey.Backspace)
            {
                if (MenuStack.PeekMenu().CanBePopped)
                {
                    MenuStack.PopMenu();
                }
            }
```

```
            return selectedIndex;
        }
    }
}using System;
using System.Data;
using BudgetApp.Menus;
using Spectre.Console;


namespace BudgetApp
{
    public class SelectableMenu
    {
        private MenuNavigator _menuNavigator;
        public MenuDisplay _menuDisplay;
        public string[] Options { get; set; }
        private string Prompt { get; }
        private int selectedIndex;

        public SelectableMenu(string prompt, string[] options)
        {
            this._menuNavigator = new MenuNavigator();
            this._menuDisplay = new MenuDisplay();
            this.Options = options;
            this.Prompt = prompt;
            this.selectedIndex = 0;
        }

        public int Run()
        {
            while (true)
            {
                _menuDisplay.DisplayOptions(Prompt, Options, selectedIndex);
                var keyPressed = MenuFeatures.GetInput();
                selectedIndex = _menuNavigator.Navigate(keyPressed, Options, selectedIndex);
                if (keyPressed == ConsoleKey.Enter)
                {
                    return selectedIndex;
                }
            }
        }
    }
}
```

```
        using Spectre.Console;

namespace BudgetApp;

public class MenuDisplay
{
    public void DisplayOptions(string prompt, string[] options, int selectedIndex)
    {
        Clear();

        var asciiArt = new FigletText("Budget App");
        AnsiConsole.Render(asciiArt.Color(Color.Green));

        AnsiConsole.MarkupLine(prompt);
        for (int i = 0; i < options.Length; i++)
        {
            var currentOption = options[i];
            string prefix;

            if (i == selectedIndex)
            {
                prefix = "*";
            }
            else
            {
                prefix = " ";
            }

            AnsiConsole.MarkupLine($"{prefix} {currentOption} {prefix}");
        }
    }

    public void Clear()
    {
        Console.Clear();
    }
}namespace BudgetApp.Menus;

public class MenuFeatures
{
    public static string PrintCentered(string messageToCenter, int length)
    {
        string padding = GetPaddingString(messageToCenter, length);
        return padding + messageToCenter + padding;
```

```
    }

    private static string GetPaddingString(string stringToPad, int length)
    {
        int padding = (length - stringToPad.Length) / 2;
        return new string(' ', padding);
    }

    public static ConsoleKey GetInput()
    {
        ConsoleKeyInfo keyInfo = Console.ReadKey(true);
        return keyInfo.Key;
    }
}namespace BudgetApp
{
    public class Calendar
    {
        public DateTime Start { get; set; }
        public DateTime End { get; set; }

        public DurationScale _currentScale { get; set; }

        public string Format { get; set; }

        public Calendar()
        {
            Start = new DateTime(DateTime.Now.Year, 1, 1);
            End = Start.AddYears(1).AddSeconds(-1);
            _currentScale = DurationScale.Yearly;
            Format = "yyyy";
        }

        public Calendar(DateTime customStart, string customFormat)
        {
            this.Start = customStart;
            this.Format = customFormat;
        }

        public enum DurationScale
        {
            Yearly,
            Monthly,
            Daily
        }
```

```
    public void NavigateCalendar(ConsoleKey keyPressed, string monthlyFormat, string
yearlyFormat, string dailyFormat)
    {
        switch (keyPressed)
        {
            case ConsoleKey.RightArrow:
                MoveToNextDate();
                break;
            case ConsoleKey.LeftArrow:
                MoveToPreviousDate();
                break;
            case ConsoleKey.UpArrow:
                IncreaseTimeScale(monthlyFormat, yearlyFormat, dailyFormat);
                break;
            case ConsoleKey.DownArrow:
                DecreaseTimeScale(monthlyFormat, yearlyFormat, dailyFormat);
                break;
        }
    }

    private void MoveToNextDate()
    {
        switch (_currentScale)
        {
            case DurationScale.Yearly:
                Start = Start.AddYears(1);
                End = Start.AddYears(1).AddSeconds(-1);
                break;
            case DurationScale.Monthly:
                Start = Start.AddMonths(1);
                End = Start.AddMonths(1).AddSeconds(-1);
                break;
            case DurationScale.Daily:
                Start = Start.AddDays(1);
                End = Start.AddDays(1).AddSeconds(-1);
                break;
        }
    }

    private void MoveToPreviousDate()
    {
        switch (_currentScale)
```

```
        {
            case DurationScale.Yearly:
                Start = Start.AddYears(-1);
                End = Start.AddYears(1).AddSeconds(-1);
                break;
            case DurationScale.Monthly:
                Start = Start.AddMonths(-1);
                End = Start.AddMonths(1).AddSeconds(-1);
                break;
            case DurationScale.Daily:
                Start = Start.AddDays(-1);
                End = Start.AddDays(1).AddSeconds(-1);
                break;
        }
    }

    private void IncreaseTimeScale(string monthlyFormat, string yearlyFormat, string
dailyFormat)
    {
        if (_currentScale == DurationScale.Yearly)
        {
            _currentScale = DurationScale.Monthly;
            Start = new DateTime(End.Year, End.Month, 1);
            End = Start.AddMonths(1).AddSeconds(-1);
            Format = monthlyFormat;
        }
        else if (_currentScale == DurationScale.Monthly)
        {
            _currentScale = DurationScale.Daily;
            End = Start.AddDays(1).AddSeconds(-1);
            Format = dailyFormat;
        }
        else if (_currentScale == DurationScale.Daily)
        {
            _currentScale = DurationScale.Yearly;
            Start = new DateTime(End.Year, 1, 1);
            End = Start.AddYears(1).AddSeconds(-1);
            Format = yearlyFormat;
        }
    }

    private void DecreaseTimeScale(string monthlyFormat, string yearlyFormat, string
dailyFormat)
    {
```

```
            if (_currentScale == DurationScale.Yearly)
            {
                _currentScale = DurationScale.Daily;
                End = Start.AddDays(1).AddSeconds(-1);
                Format = dailyFormat;
            }
            else if (_currentScale == DurationScale.Daily)
            {
                _currentScale = DurationScale.Monthly;
                Start = new DateTime(End.Year, End.Month, 1);
                End = Start.AddMonths(1).AddSeconds(-1);
                Format = monthlyFormat;
            }
            else if (_currentScale == DurationScale.Monthly)
            {
                _currentScale = DurationScale.Yearly;
                Start = new DateTime(End.Year, 1, 1);
                End = Start.AddYears(1).AddSeconds(-1);
                Format = yearlyFormat;
            }
        }
    }
}
using System;

namespace BudgetApp;

public class Menu
{
    public string Name { get; set; }
    public bool CanBePopped { get; set; } = true;

    public Menu(string name)
    {
        Name = name;
    }

    public virtual void Run()
    {
    }
}
using System;

namespace BudgetApp
```

```
{
    public static class MenuStack
    {
        public static Stack<Menu> menuStack = new Stack<Menu>();

        public static void PushMenu(Menu menu)
        {
            Console.WriteLine("MENU COUNT " + menuStack.Count);
            menuStack.Push(menu);
            menu.Run();
        }

        public static void PopMenu()
        {
            if (menuStack.Count > 0)
            {
                menuStack.Pop();
            }

            Menu menu = PeekMenu();
            if (menuStack.Count > 0)
            {
                menu.Run();
            }
        }

        internal static Menu PeekMenu()
        {
            try
            {
                return menuStack.Peek();
            }
            catch (InvalidOperationException ex)
            {
                Console.WriteLine("You are already at the main menu" + ex.Message);
                return null;
            }
        }
    }
}using System;
using BudgetApp;

class Program
{
```

```
    static void Main(string[] args)
    {
        ExpenseTrackerApp.Run();
    }
}
```