Software Design Plan

WGU Student ID	011933973

A. Business Case

1. Problem Statement

Endothon Finance, a loan servicing company, recently launched a new web app intended to streamline the loan application process by requesting financial data from the last five fiscal years. However, an issue has been identified where the web app requests financial data from the first five fiscal years instead of the most recent ones. The current logic of the web app fails to correctly determine and request the appropriate fiscal years' financial data based on the business's establishment year, leading to incorrect loan profiles being generated.

2. Business requirements

Requirement 1: The web app must request financial data for the five most recent completed fiscal years, disregarding the current year.

Requirement 2: For businesses established less than five years ago, the web app must request financial data from all available years and forecast data for the remaining years to make up five years of data.

Current Failure: The web app currently requests data from the first five years after the business's establishment, which is incorrect and impacts the accuracy of the loan profiles shared with lenders.

3. In-scope action items

Action Item 1: Correct the web app logic to request financial data from the most recent five fiscal years for businesses established over five years ago.

> **Alignment:** This action aligns with the requirement to provide accurate loan profiles based on the most recent financial data.

Action Item 2: Update the logic to request current and projected financial data for businesses established within the last five years.

Alignment: This aligns with the requirement to gather sufficient financial information to meet the five-year data requirement.

Action Item 3: Implement automated testing to ensure the logic correctly identifies the required fiscal years.

Alignment: This supports quality assurance and ensures the web app functions as intended.

4. Out-of-scope action items

Action Item 1: Modifying the user interface to display additional financial data beyond the five years is out of scope.

Reason: The issue is specific to the logic determining the fiscal years, not the UI.

Action Item 2: Integrating additional financial forecasting tools into the web app is out of scope.

Reason: The current task focuses on correcting the logic error, not expanding the app's capabilities.

B. Requirements

1. Functional requirements

Functional Requirement 1: The web app must identify the five most recent fiscal years for financial data when a business is older than five years.

Functional Requirement 2: The web app must request current and projected financial data for the remaining years if a business is less than five years old.

Functional Requirement 3: The web app must automatically adjust the fields presented to users based on the business's establishment year.

2. Non-functional requirements

Non-Functional Requirement 1: The web app must process the financial data requests with a response time of less than 2 seconds.

Non-Functional Requirement 2: The web app must be compatible with all major web browsers to ensure broad accessibility.

C. Software Design

1. Software behavior

Input/Event 1: Business Establishment Date Entry

Intended Response: Upon entering the business establishment date, the web app will calculate and display the appropriate fiscal years for financial data entry. If the business is older than five years, the web app will display input fields for the most recent five fiscal years. If the business is less than five years old, it will display input fields for the available fiscal years and forecast years to cover the most recent five years.

Constraints: The logic must accurately differentiate between businesses older than five years and those younger, ensuring that the correct years are calculated and displayed. The system must also handle edge cases where the establishment date is exactly five years old. Additionally, the implementation in C# should ensure that these calculations are efficient and executed in real-time within the user interface.

Input/Event 2: Submission of Financial Data

Intended Response: When a user submits their financial data, the web app will verify that all required fiscal years' data are provided according to the calculated requirements. If any fiscal year's data is missing or incomplete, the app will prompt the user to fill in the gaps before allowing submission.

Constraints: The system must handle scenarios where data for the required years is incomplete or missing. In the case of businesses with less than five years of data, the app should confirm that both historical and forecasted data are provided where applicable. The C# backend should enforce these checks, ensuring that the submission process does not proceed unless all required fields are correctly filled.

Input/Event 3: Data Validation on Entry

Intended Response: As the user enters financial data for each fiscal year, the web app will validate the data in real-time to ensure it meets the necessary format and content standards (e.g., numeric values for revenue, proper date formats). If any data is incorrect, the system will highlight the error and provide feedback to the user.

Constraints: The validation logic must be robust and responsive, ensuring that it catches errors immediately without delaying the user's progress. This requires a

combination of client-side validation (using JavaScript) and server-side validation (in C#) to ensure data integrity before it is processed or stored. Additionally, the system should be able to handle various formats that users might input and convert them into a standardized form.

Input/Event 4: Forecast Data Generation

Intended Response: For businesses with less than five years of historical data, the web app will generate fields for forecasted financial data, allowing users to input projections for the missing years. The app should ensure that these forecasted data fields are clearly labeled and distinguishable from historical data fields.

Constraints: The logic must ensure that forecasted data is only requested for years that are necessary to meet the five-year requirement. Additionally, the C# implementation must ensure that these forecasted entries are stored separately from historical data and flagged accordingly in the database for future processing and analysis.

This software behavior section ensures that the web app operates correctly within the C# tech stack, efficiently handling inputs and events related to the fiscal data requirements of Endothon Finance's loan application process.

2. Software structure

The design approach for the web app will segment the development into three primary components, each implemented with a focus on modularity and maintainability within a C# tech stack. This structure ensures that the application remains flexible and scalable while addressing the specific business requirements outlined in the software behavior section.

Component 1: Fiscal Year Logic Module

- **Class:** FiscalYearService
  - **Functionality:** This class will handle all logic related to determining the appropriate fiscal years for financial data entry based on the business establishment date.
  - **Key Methods:**
    - List<int> GetRequiredFiscalYears(DateTime establishmentDate)
      - **Description:** This method calculates and returns a list of the fiscal years that the web app needs to request from the user. If the business is older than five years, it returns the most recent

five fiscal years. For businesses younger than five years, it returns the available historical years and the necessary forecasted years.

- FinancialData GetFinancialDataForYear(int year, int applicationId)
  - **Description:** Retrieves financial data for a specific year and application ID. If no data is found, it returns null.
- ForecastedData GetForecastDataForYear(int year, int applicationId)
  - **Description:** Retrieves forecasted financial data for a specific year and application ID if the historical data is unavailable.

Component 2: User Interface Module

- **Class:** FinancialDataEntryUI
  - **Functionality:** This class will manage the dynamic user interface elements, ensuring that the correct input fields are displayed based on the fiscal years calculated by the FiscalYearService.
  - **Key Methods:**
    - void DisplayFiscalYearFields(List<int> fiscalYears)
      - **Description:** Dynamically generates and displays input fields for each fiscal year provided by the FiscalYearService. The method ensures that fields for both historical and forecasted data are labeled and organized correctly in the UI.
    - void HighlightMissingDataFields()
      - **Description:** Identifies and highlights any fields where data is missing or incomplete before submission, providing immediate feedback to the user.
    - void ShowValidationMessages(List<string> errors)
      - **Description:** Displays validation messages to the user, detailing any issues that need to be addressed before data submission.

Component 3: Validation Module

- **Class:** DataValidationService
  - **Functionality:** This class will ensure that all financial data entered by the user meets the required format and content standards before submission.
  - **Key Methods:**
    - bool ValidateFinancialData(FinancialData data)

        - **Description:** Validates individual financial data entries, checking for correct formatting, completeness, and logical consistency (e.g., no negative values where they don't belong).
    - bool ValidateForecastData(ForecastedData forecast)
        - **Description:** Specifically validates forecasted financial data, ensuring it follows a logical progression from historical data.
    - List<string> GetValidationErrors(FinancialData data)
        - **Description:** Returns a list of validation errors for the provided financial data, which will be used by the FinancialDataEntryUI to display to the user.

Integration and Interaction

These components will work together to ensure the web app functions correctly:

- The FiscalYearService will be the backbone of the logic, determining which years of data are needed based on the business establishment date.
- The FinancialDataEntryUI will use the output from the FiscalYearService to dynamically display the correct input fields to the user.
- The DataValidationService will be invoked whenever the user attempts to submit their data, ensuring that all entries are valid and complete before allowing the submission to proceed.

D. Development Approach

1. Planned deliverables

This project will produce the following key deliverables, each critical to ensuring the web app's functionality aligns with the requirements of Endothon Finance.

Deliverable 1: FiscalYearService Class

- **Description:** The `FiscalYearService` class will be responsible for all logic related to determining the correct fiscal years for financial data entry based on the business establishment date. This includes calculating which years need financial data and handling both historical and forecasted data.
- **Steps to Create:**
    1. **Design the Class Structure:** Define the methods and properties required for calculating fiscal years.

2. **Implement `GetRequiredFiscalYears(DateTime establishmentDate)` Method:** Develop the method to calculate and return a list of the fiscal years needed based on the business's age.
3. **Implement `GetFinancialDataForYear(int year, int applicationId)` Method:** Code this method to retrieve historical financial data for a specified year. If data is missing, ensure it returns `null`.
4. **Implement `GetForecastDataForYear(int year, int applicationId)` Method:** Develop this method to retrieve forecasted financial data when historical data is not available.
5. **Unit Testing:** Write and execute unit tests to validate the functionality of each method using various scenarios (e.g., different establishment dates, missing data cases).

Deliverable 2: FinancialDataEntryUI Class

- **Description:** The `FinancialDataEntryUI` class will manage the user interface components, ensuring that the correct input fields are displayed dynamically based on the fiscal years provided by the `FiscalYearService`.
- **Steps to Create:**
    6. **Design the UI Layout:** Plan the layout for the financial data entry fields, ensuring clear differentiation between historical and forecasted data.
    7. **Implement `DisplayFiscalYearFields(List<int> fiscalYears)` Method:** Code the dynamic generation of input fields based on the list of fiscal years received.
    8. **Implement `HighlightMissingDataFields()` Method:** Develop functionality to identify and visually highlight any missing data fields before submission.
    9. **Implement `ShowValidationMessages(List<string> errors)` Method:** Create the logic to display validation messages to users, guiding them to correct any errors in their submissions.
    10. **UI Testing:** Perform user interface testing to ensure that fields are displayed correctly and that the user experience is intuitive and error-free.

Deliverable 3: DataValidationService Class

- **Description:** The `DataValidationService` class will be responsible for ensuring that all financial data entered by the user meets required standards before submission.

- **Steps to Create:**
    11. **Design the Validation Logic:** Outline the validation rules for both historical and forecasted financial data, covering formats, completeness, and logical consistency.
    12. **Implement `ValidateFinancialData(FinancialData data)` Method:** Code this method to validate the integrity of individual financial data entries, including checks for negative values where they aren't allowed.
    13. **Implement `ValidateForecastData(ForecastedData forecast)` Method:** Develop the logic to ensure forecasted data follows a logical progression and matches the expected financial trends.
    14. **Implement `GetValidationErrors(FinancialData data)` Method:** Write this method to return a list of validation errors, which the UI will display to users.
    15. **Validation Testing:** Test the validation logic under different scenarios to ensure that all edge cases are handled and that the validation is both comprehensive and efficient.

Deliverable 4: Integration and Testing Documentation

- **Description:** Comprehensive documentation will be produced to support the integration of these modules, including detailed testing procedures and results.
- **Steps to Create:**
    16. **Draft Integration Guidelines:** Create a document outlining how each of the classes (`FiscalYearService`, `FinancialDataEntryUI`, `DataValidationService`) should interact within the broader application.
    17. **Develop Testing Procedures:** Define procedures for unit testing, integration testing, and user acceptance testing, ensuring all methods and UI components work together seamlessly.
    18. **Record Test Results:** Document the outcomes of each testing phase, including any issues encountered and how they were resolved.
    19. **Finalize and Review Documentation:** Ensure all documentation is clear, thorough, and reviewed by the development team for accuracy.

These deliverables will be developed iteratively, allowing for continuous testing and refinement, which aligns with the Agile methodology chosen for this project. By segmenting the work into distinct, manageable components, we ensure that each aspect of the web app is thoroughly developed and tested, leading to a final product that meets all business and functional requirements.

2. Sequence of deliverables

The sequence of deliverables is structured to ensure a logical progression from core functionality development to interface implementation, validation, and final integration. This sequence is designed to maximize efficiency, allow for continuous testing, and ensure that each component is thoroughly developed before the next phase begins.

Sequence of Implementation:

2. **Deliverable 1: FiscalYearService Class**
    - **Justification:** The `FiscalYearService` class is the foundation of the project, handling the core logic for determining the correct fiscal years for financial data entry. Implementing this first allows us to establish the essential functionality that other components will rely on. By developing and testing this class first, we ensure that the logic behind fiscal year calculations is sound, which is crucial before proceeding to any UI or validation work.
3. **Deliverable 2: FinancialDataEntryUI Class**
    - **Justification:** Once the `FiscalYearService` is in place and validated, the next step is to create the user interface that will interact with this logic. The `FinancialDataEntryUI` class will use the fiscal year data generated by the `FiscalYearService` to dynamically display the appropriate fields to the user. This deliverable is sequenced second because it depends on the correct operation of the `FiscalYearService`. By focusing on UI implementation after the core logic, we ensure that the user interface is built on a solid, functional foundation.
4. **Deliverable 3: DataValidationService Class**
    - **Justification:** With both the core logic and the user interface in place, the next step is to ensure that all data entered into the system is valid and consistent. The `DataValidationService` class is critical for maintaining data integrity, and it is sequenced third because it must validate the data generated and entered through the UI. By placing this deliverable after the UI implementation, we allow for comprehensive testing of both individual entries and overall data flows, ensuring that the system can handle all expected inputs and edge cases before moving on to final integration.
5. **Deliverable 4: Integration and Testing Documentation**
    - **Justification:** The final deliverable is the integration and testing documentation, which will provide a comprehensive guide to how all the

components interact and the results of the testing processes. This deliverable is last in the sequence because it relies on the completion and testing of all other components. Once the `FiscalYearService`, `FinancialDataEntryUI`, and `DataValidationService` classes are fully implemented and tested, the integration documentation will ensure that all components work together seamlessly. This documentation will also serve as a reference for future development and maintenance, solidifying the robustness of the entire system.

Justification of the Planned Sequence:

This sequence is carefully planned to ensure that each deliverable builds upon the previous one, reducing the risk of rework and ensuring that each component is fully functional before integrating it with others. By starting with the core logic (`FiscalYearService`), then moving to the user interface (`FinancialDataEntryUI`), followed by data validation (`DataValidationService`), and finally documenting the integration, we create a strong, reliable foundation before layering on more complex interactions and ensuring that all elements work together smoothly. This approach aligns with Agile principles, allowing for continuous testing, feedback, and refinement at each stage of the project.

3. Development environment

Programming Language: C# 11

Purpose: Our code base is developed using C# 11, chosen for its robust support for modern programming paradigms, which are critical in resolving logic-based issues in the loan application process.

Development Environment: Microsoft Visual Studio 2024

Purpose: Visual Studio provides an advanced development environment that includes tools essential for refactoring and debugging. This environment is crucial for identifying and fixing the logic errors affecting the financial data retrieval in the web app.

External Libraries: Entity Framework Core 7, SQL Server 2024

Purpose: Entity Framework Core 7 manages the mapping between C# classes and database tables, allowing for accurate retrieval and manipulation of financial data.

SQL Server 2024 serves as the backend database, storing and processing the historical and forecasted financial data required for loan applications.

Version Control Tools: Git, GitHub

Purpose: Git tracks changes across the codebase, ensuring that updates to the web app's logic can be versioned and rolled back if necessary. GitHub facilitates collaboration and version control, allowing seamless updates and bug fixes to be implemented and reviewed by the development team.

Testing Framework: xUnit

Purpose: xUnit is used for writing and running unit tests, ensuring that the updated logic for financial data retrieval is thoroughly tested. This testing framework helps validate that the changes meet the specifications and that the web app returns the correct financial data for loan profiles.

4. Development Methodology

Chosen Methodology: Agile Development

Agile development has been chosen for this project due to its flexibility and ability to adapt quickly to changing requirements. This methodology is perfect for addressing the critical bug in Endothon Finance's web app, allowing us to rapidly develop, test, and refine the solution.

How Agile Informed the Development Planning Process

Agile's iterative approach directly shaped our planning. By breaking the project into manageable sprints, we can update the logic module first, deliver functional increments quickly, and continuously test and validate each step. We'll use Test Driven Development (TDD) within Agile, writing tests before the code. As we implement the logic, the goal will be to pass these tests, ensuring accuracy from the start. This iterative process allows us to catch and fix issues early, minimizing risk.

Why Agile Was Chosen Over an Alternative Methodology

We considered the Waterfall model, which is structured but rigid. Waterfall requires each phase to be completed before moving on, which could cause delays if problems arise late in the process. In contrast, Agile's flexibility allows for continuous feedback and real-time adjustments. This is crucial for quickly resolving the web app's logic error, ensuring we can

adapt without disrupting the project timeline. Agile's speed, adaptability, and ability to minimize risks made it the best choice for this project.

5. Sources

Atlassian. (n.d.). Project management: An introduction to project management. Atlassian. https://www.atlassian.com/agile/project-management/project-management-intro

Rees, J. (2023, July 7). Agile vs. Waterfall methodology: Which is right for your project? Forbes Advisor. https://www.forbes.com/advisor/business/agile-vs-waterfall-methodology/

Rigby, D. K., Elk, S., & Berez, S. (2023, October). It's time to end the battle between Waterfall and Agile. Harvard Business Review. https://hbr.org/2023/10/its-time-to-end-the-battle-between-waterfall-and-agile

Pandey, M. (2023, June 14). C# testing frameworks: A comprehensive list. LambdaTest. https://www.lambdatest.com/blog/c-sharp-testing-frameworks/