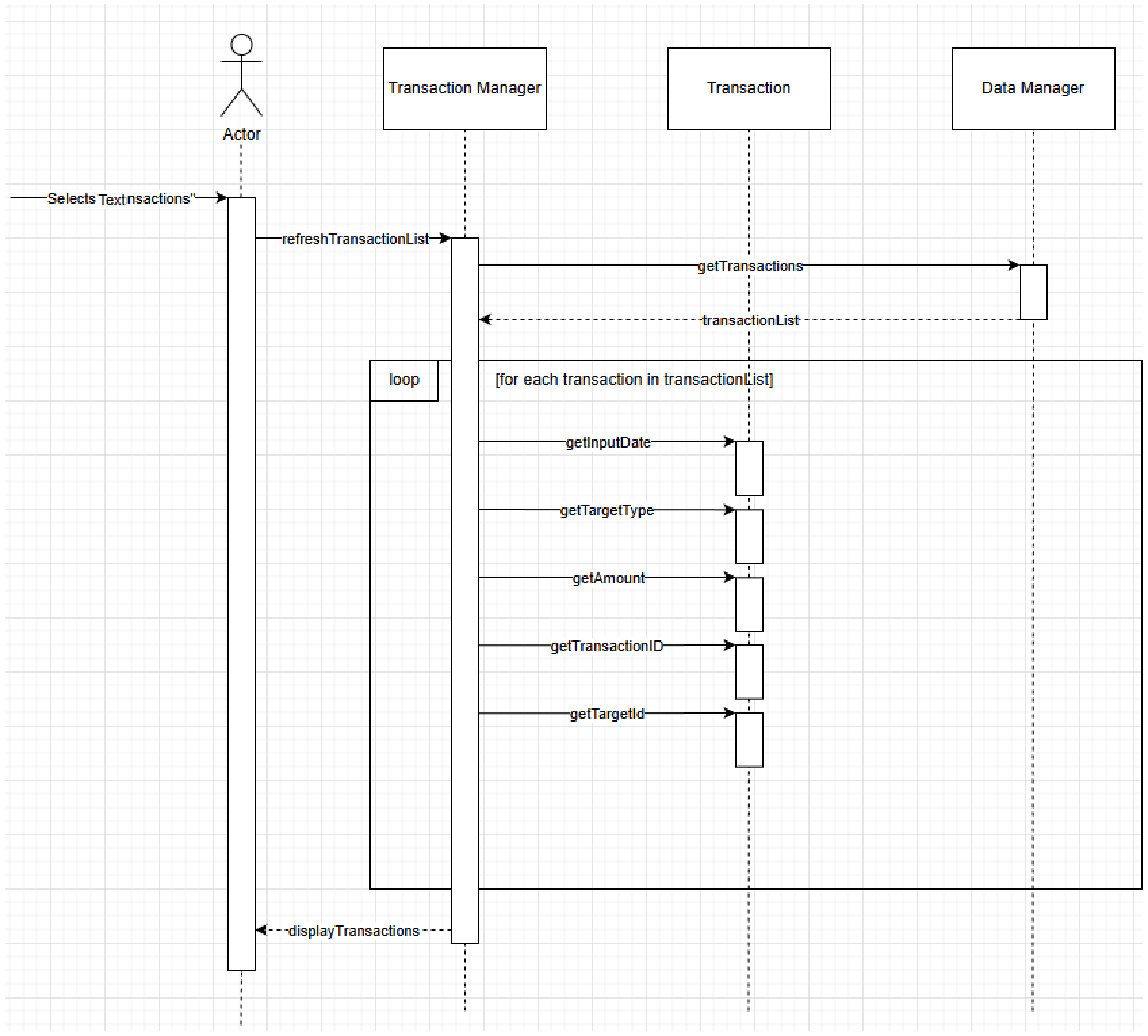
CS3307 - Finance Tracker

Deliverable 3

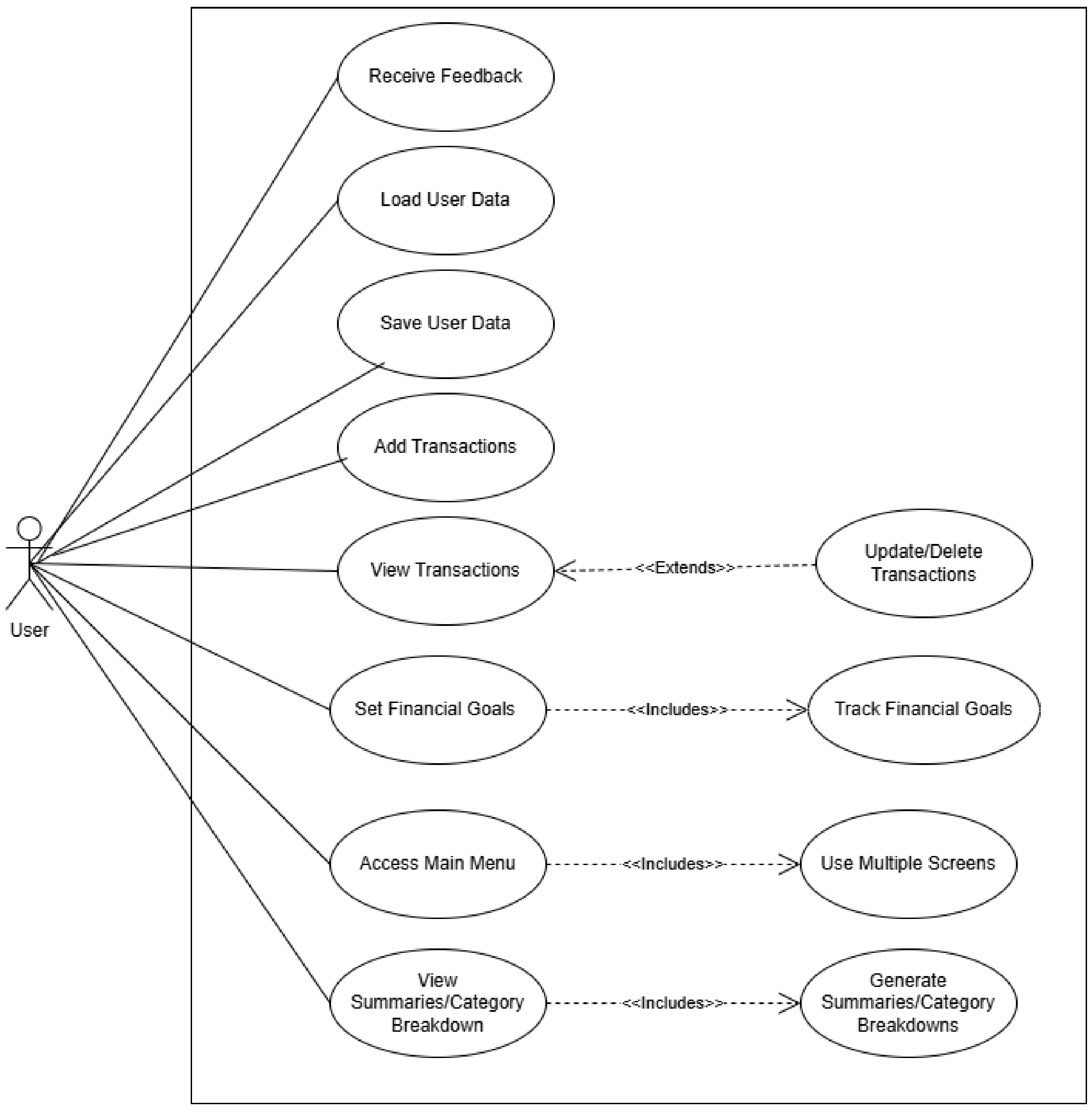
Muhammad Shayaan Ali - 251373910 - mali484@uwo.ca
Novak Vukojicic - 251359686 - nvukojic@uwo.ca

Final UML Diagrams:

Sequence Diagram:

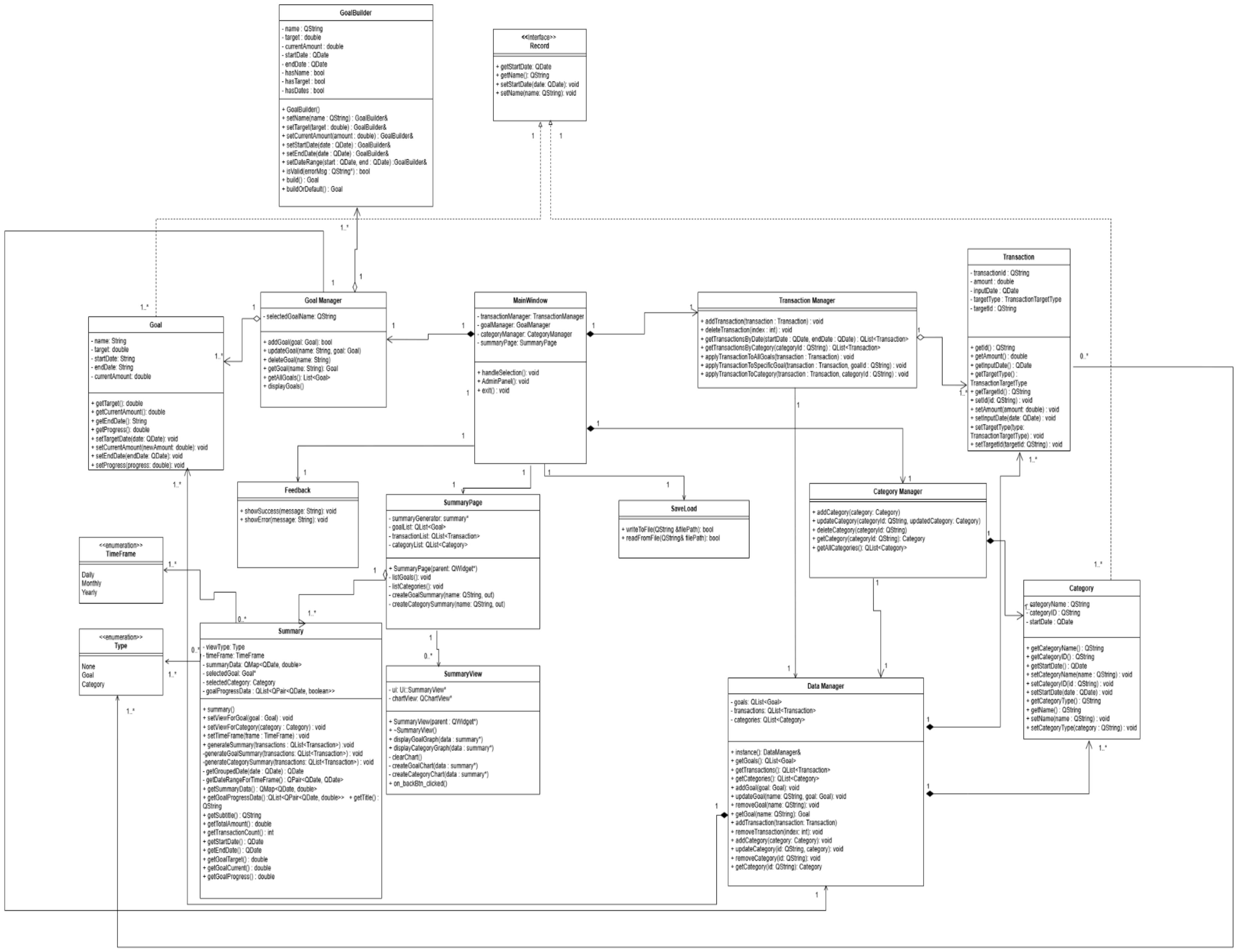


Use Case Diagram:



Class Diagram:

https://drive.google.com/file/d/1FxPgP6LEIOoPkLT6q4QQdYJInrJU1vF6/view?usp=drive_link (for visiblity)


GoalBuilder
- name : QString
- target : double
- currentAmount : double
- startDate : QDate
- endDate : QDate
- hasName : bool
- hasTarget : bool
- hasDates : bool
+ GoalBuilder()
+ setName(name : QString) : GoalBuilder&
+ setTarget(target : double) : GoalBuilder&
+ setCurrentAmount(amount : double) : GoalBuilder&
+ setStartDate(date : QDate) : GoalBuilder&
+ setEndDate(date : QDate) : GoalBuilder&
+ setDateRange(start : QDate, end : QDate) :GoalBuilder&
+ isValid(errorMsg : QString*) : bool
+ build() : Goal
+ buildOrDefault() : Goal
<<Interface>>
Record
+ getStartDate: QDate
+ getName(): QString
+ setStartDate(date: QDate): void
+ setName(name: QString): void
Goal
- name: String
- target: double
- startDate: String
- endDate: String
- currentAmount: double
+ getTarget(): double
+ getCurrentAmount(): double
+ getEndDate(): String
+ getProgress(): double
+ setTargetDate(date: QDate): void
+ setCurrentAmount(newAmount: double): void
+ setEndDate(endDate: QDate): void
+ setProgress(progress: double): void
Goal Manager
- selectedGoalName: QString
+ addGoal(goal: Goal): bool
+ updateGoal(name: String, goal: Goal)
+ deleteGoal(name: String)
+ getGoal(name: String): Goal
+ getAllGoals(): List<Goal>
+ displayGoals()
MainWindow
- transactionManager: TransactionManager
- goalManager: GoalManager
- categoryManager: CategoryManager
- summaryPage: SummaryPage
+ handleSelection(): void
+ AdminPanel(): void
+ exit() : void
Transaction Manager
+ addTransaction(transaction : Transaction) : void
+ deleteTransaction(index : int) : void
+ getTransactionsByDate(startDate : QDate, endDate : QDate) : QList<Transaction>
+ getTransactionsByCategory(categoryId : QString) : QList<Transaction>
+ applyTransactionToAllGoals(transaction : Transaction) : void
+ applyTransactionToSpecificGoal(transaction : Transaction, goalId : QString) : void
+ applyTransactionToCategory(transaction : Transaction, categoryId : QString) : void
Transaction
- transactionId : QString
- amount : double
- inputDate : QDate
- targetType : TransactionTargetType
- targetId : QString
+ getId() : QString
+ getAmount() : double
+ getInputDate() : QDate
+ getTargetType() : TransactionTargetType
+ getTargetId() : QString
+ setId(id: QString) : void
+ setAmount(amount: double) : void
+ setInputDate(date: QDate) : void
+ setTargetType(type: TransactionTargetType) : void
+ setTargetId(targetId: QString) : void
Feedback
+ showSuccess(message: String): void
+ showError(message: String): void
SummaryPage
- summaryGenerator: summary*
- goalList: QList<Goal>
- transactionList: QList<Transaction>
- categoryList: QList<Category>
+ SummaryPage(parent: QWidget*)
- listGoals(): void
- listCategories(): void
- createGoalSummary(name: QString, out)
- createCategorySummary(name: QString, out)
SaveLoad
+ writeToFile(QString &filePath): bool
+ readFromFile(QString& filePath): bool
Category Manager
+ addCategory(category: Category)
+ updateCategory(categoryId: QString, updatedCategory: Category)
+ deleteCategory(categoryId: QString)
+ getCategory(categoryId: QString): Category
+ getAllCategories(): QList<Category>
<<enumeration>>
TimeFrame
Daily
Monthly
Yearly
<<enumeration>>
Type
None
Goal
Category
Summary
- viewType: Type
- timeFrame: TimeFrame
- summaryData: QMap<QDate, double>
- selectedGoal: Goal*
- selectedCategory: Category
- goalProgressData : QList<QPair<QDate, boolean>>
+ summary()
+ setViewForGoal(goal : Goal) : void
+ setViewForCategory(category : Category) : void
+ setTimeFrame(frame : TimeFrame) : void
+ generateSummary(transactions : QList<Transaction>) :void
-generateGoalSummary(transactions: QList<Transaction>) : void
-generateCategorySummary(transactions: QList<Transaction>) : void
- getGroupedDate(date : QDate) : QDate
- getDateRangeForTimeFrame() : QPair<QDate, QDate>
+ getSummaryData() : QMap<QDate, double>
+ getGoalProgressData() :QList<QPair<QDate, double>> + getTitle() : QString
+ getSubtitle() : QString
+ getTotalAmount() : double
+ getTransactionCount() : int
+ getStartDate() : QDate
+ getEndDate() : QDate
+ getGoalTarget() : double
+ getGoalCurrent() : double
+ getGoalProgress() : double
SummaryView
- ui: Ui::SummaryView*
- chartView: QChartView*
+ SummaryView(parent : QWidget*)
+ ~SummaryView()
+ displayGoalGraph(data : summary*)
+ displayCategoryGraph(data : summary*)
- clearChart()
- createGoalChart(data : summary*)
- createCategoryChart(data : summary*)
+ on_backBtn_clicked()
Category
- categoryName : QString
- categoryID : QString
- startDate : QDate
+ getCategoryName() : QString
+ getCategoryID() : QString
+ getStartDate() : QDate
+ setCategoryName(name : QString) : void
+ setCategoryID(id : QString) : void
+ setStartDate(date : QDate) : void
+ getCategoryType() : QString
+ getName() : QString
+ setName(name : QString) : void
+ setCategoryType(category : QString) : void
Data Manager
- goals: QList<Goal>
- transactions: QList<Transaction>
- categories: QList<Category>
+ instance(): DataManager&
+ getGoals(): QList<Goal>
+ getTransactions(): QList<Transaction>
+ getCategories(): QList<Category>
+ addGoal(goal: Goal): void
+ updateGoal(name: QString, goal: Goal): void
+ removeGoal(name: QString): void
+ getGoal(name: QString): Goal
+ addTransaction(transaction: Transaction)
+ removeTransaction(index: int): void
+ addCategory(category: Category): void
+ updateCategory(id: QString, category): void
+ removeCategory(id: QString): void
+ getCategory(id: QString): Category
1..*
1
0..*

Architecture Overview:

The Finance Tracker application is structured using a modular architecture that separates user interaction, data management, and domain logic by using different classes such as managers and ui classes. This separation ensures that each component has a clear responsibility, improving maintainability, scalability and testability of the system.

The User Interface module:

The user interface module is implemented using Qt, and is responsible for displaying financial data, handling user input and providing feedback when errors occur. It does not directly manipulate stored data. Instead, it forwards user actions such as adding goals, transactions, or categories to the data management layer which is controlled by managers. The UI module also includes a summary section where visualization is created showing a summary of the financial data. The component retrieves its data from the data management layer and presents a visual depending on the users wants, for example showing a graph of progress towards a goal or the amount spent in the last year in a category, broken down per month. This allows users to quickly understand trends and overall financial status. This design prevents tight coupling between the UI and the core logic and allows for a more modular approach.

The Data Management Module:

The data management module is centered around the DataManager class which has single instances of other managers such as CategoryManager, TransactionsManager and GoalManager, acting as a central coordinator. There are only single instances of all of these managers, controlled by DataManager allowing it to single handedly manage the collection of goals, transactions and categories and provide methods for creating, updating, removing these entities. It also emits notification using Qt Signal system to allow other components to react when data is changed without direct dependencies, allowing all data across the program to be consistent.

The Domain Model Module:

The domain model consists of classes that act as entities such as the Goal, Transaction and Category classes. They represent core financial entities and encapsulate the data required to model the system’s functionality. They are not directly connected to the UI and instead act as a form of representing data, with each class being able to be used in multiple instances, representing different values of data, having multiple different goals for example).

Overall, data flow begins when the user performs an action in the UI, allowing a request to be passed to the DataManage. This manager will process the information and update the relevant domain objects. Once updated the manager will make the new data available and notify the observers, allowing the UI to refresh and display the updated state.

Object-Oriented Design (OOD) Principles:

The system was designed following core Object-Oriented Design principles to promote modularity, clarity, and ease of testing.

Encapsulation:

Applied throughout the domain model classes (goal, transaction, and category). These classes store their data as private members and expose controlled access through public getters and setters, preventing unauthorized modification of internal state and ensures that all changes follow defined logic

Abstraction:

This is achieved through the DataManager as a high-level interface for data operations. Other components interact with the system's data through this manager without understanding how the data is stored or changed within the system. This allows the implementation details to be hidden, simplifying interactions between components in the system.

Inheritance:

Inheritance is achieved using the record interface, which is implemented by both the goal and category classes. This shared interface defines common behaviors such as retrieving names and identifies, allowing different record types to be treated consistently within the system. By inheriting from a common interface, these classes reuse shared concepts while still providing their own specialized data and behavior

Polymorphism:

Achieved by enabling interaction with objects through the Record interface rather than their concrete types. Components that operate on record can work with either goal or category instances interchangeably, relying on the interface's contract. This allows the system to handle different record types uniformly and makes it easier to extend the system with new record types in the future without having to modify existing logic.

Design Patterns:

Singleton Pattern:

The Singleton pattern ensures that a class has only one instance throughout the application's lifetime and provides a global access point to the instance. In the project, we created a new manager class called DataManager which implements this pattern through a private constructor and a static instance(). This design ensures that all parts of the application access the same centralized data stored for goals, transactions and categories, preventing data inconsistency issues that could occur when having multiple instances.

Observer Pattern:

The Observer pattern establishes a one-to-many dependency between objects, where when one object (the subject) changes state, all its dependents (observers) are automatically notified and updated. Our project implements this pattern using Qt's signal-slot mechanism, where DataManager acts as the subject by emitting functions such as goalsChanged() whenever data is modified. The summaryPage class for example acts as an observer by connecting with goalsChanged() signal, automatically triggering a function to refresh the UI whenever goals are affected. This decouples the data layer from the presentation layer and ensures that the UI stays synchronized without explicit update calls, making the code more maintainable.

Builder Pattern:

The builder pattern is used to manage the creation of Goal objects in a structured and reliable way. This pattern is implemented through the GoalBuilder class, which constructs a Goal instance step by step by setting attributes such as name and target amount to the instance. By separating object construction and validation from the Goal class, the design keeps the domain model simple while ensuring that only valid goals are created. Validation logic is encapsulated within the builder through the isValid() method, allowing user input to be checked before a Goal is built. This approach improves robustness, reduces error-prone construction in the UI layer and results in cleaner and more maintainable code within its respective manager.

Testing Report:

The testing report and files aimed to ensure the system's correctness using manual testing during development with the use of Google Test (GTest) and Google Mock (gMock). The Transactions and DataManager classes were chosen for the unit testing as they are central components in the system's functionality. Google Test was used to verify state-based behavior, such as correctly adding, updating, and removing goals, transactions, and categories, as well as handling edge cases like cascading deletions. Google Mock was used to simulate dependent components and test interaction-based behavior by verifying that expected methods were called using EXPECT_CALL. This approach allowed individual components to be tested in isolation without relying on the user interface or external dependencies, improving test reliability and maintainability.

DataManager Class:

- **Purpose:** Serves as the central coordinator and ensures data consistency across the system. The class also emits notifications when data changes occur, allowing other components to react accordingly
- **Critical Functionality Tested:**
    - **Add a Goal:** Verified that the addGoal() method correctly adds a new goal to the system and makes it retrievable using getGoal().
    - **Cascading Deletion:** Tested that removing a goal also removes any associated transactions linked to that goal.
    - **Add and Update a Category:** Verified that categories are added correctly and that the update method updateCategory() updates category details while preserving other attributes.
    - **Data Retrieval:** Ensured that getter methods such as getGoals(), getTransactions() and getCategory() return accurate and up-to-date data.
- **Edge Cases:**
    - **Empty Data Sets:** Tests were executed starting from a cleared data state to ensure operations behaved correctly when no goals, transactions, or categories were present.
    - **Safe Deletion Logic:** Cascading deletion tests ensure that dependent data is handled safely without runtime errors.
- **Google Mock Usage:**
  Google Mock was used to demonstrate interaction-based testing through the creation of mock observer objects. Using EXPECT_CALL, tests verified expected method calls on mock objects, illustrating how interactions between components can be validated independently of the user interface or Qt signal-slot connections. This demonstrated correct usage of Google Mock for verifying behavior rather than state.

- **Test Results:**
    - All unit tests were executed successfully
    - Core data management functionality behaved as expected
    - Cascading deletion logic maintains system consistency
    - Mock-based tests successfully demonstrating interfaction verification via EXPECT_CALL

Transaction Class:

Purpose:

- Represent a financial transaction by storing a transaction ID, amount, target type, target identifier, and input date

Critical Functionality Tested:

- Default constructor initializes all attributes to their expected default values
- Transaction ID and target ID are initialized as empty strings
- Amount is initialized to 0.0
- Target type defaults to None
- Input date defaults to the current system date
- Parameterized constructor correctly initializes amount, target type, target ID, and input date using provided values
- Setter methods correctly update transaction ID, amount, target type, target ID, and input date
- Getter methods return the correct values after updates

Edge Cases Considered:

- Handling of empty string values for transaction ID and target ID
- Use of default enum values without causing unexpected behavior
- Preservation of explicitly provided dates versus default date initialization

Test Results:

- All constructors initialized Transaction objects as expected

- Setters and getters behaved consistently and returned accurate values
- Default and empty values were handled safely
- The Transaction class demonstrated stable and predictable behavior suitable for use across the system

Reflection:

1. Managing UI Logic and Business Logic in TransactionManager

- Challenge: The TransactionManager class initially risked becoming overly complex due to handling UI events, validation logic, and direct updates to goals, categories, and transactions within a single class. This increased coupling between the UI and data layers, making the code harder to test and maintain.

- Solution: Responsibilities were clearly separated by keeping Transaction as a lightweight data model and delegating data persistence and state updates to DataManager. This reduced duplication and kept TransactionManager focused primarily on coordinating user interactions.

2. Handling Multiple Transaction Target Types

- Challenge: Supporting transactions for individual goals, all goals, and categories introduced conditional logic that was prone to errors, particularly when updating goal balances or reversing transactions during deletion.

- Solution: Dedicated helper methods (applyTransactionToAllGoals, applyTransactionToSpecificGoal, and applyTransactionToCategory) were introduced to isolate logic for each transaction type. This improved readability and reduced the likelihood of logic errors.

3. Implementation of Summary Graph feature

- Challenge: One of the main challenges faced during development was implementing the summary feature, which generates visual representations of financial data based on goals or categories. This required aggregating transaction data across different time frames, while ensuring accuracy and meaningful presentation. This became complex as synchronizing the summary logic with the UI proved to be complex.
- Solution: To address this challenge, the summary logic was separated into a dedicated summary class responsible solely for data aggregation and calculation, while visualization responsibilities were handled by the SummaryView class. This

separation simplified debugging and improved clarity. Additionally, date grouping logic was refined to ensure correct alignment with selected time frames. By isolating summary generation from the UI and refining data handling incrementally, the system became more reliable and maintainable

4. Centralizing Data Management and Handling Signals with DataManager
   - Challenge: Another significant challenge was managing shared data consistently across multiple UI components while keeping the system loosely coupled. Initially, there was a risk of different parts of the application becoming out of sync when goals, transactions, or categories were modified. We needed to ensure that changes in the data layer were automatically reflected in the UI without introducing tight coupling.
   - Solution: This challenge was resolved by centralizing all data operations within the DataManager class and using Qt's signal-slot mechanism to implement an observer-style update system. By emitting signals whenever data was modified, UI components could react automatically to changes without directly depending on data manipulation logic. This approach improved consistency across the application, reduced duplicated update code, and made the system easier to test and maintain.

Video Walkthrough Youtube Link:

https://youtu.be/zV34plTpKbc