Shri Ramdeobaba College of Engineering and Management, Nagpur
Department of Computer Science and Engineering
Session 2024-2025

Course: Design Patterns
Semester: V Semester
Marks-05

Group Activity on Pattern usage (Teachers Assessment 2)

Date: 26/10/2024 (Saturday)

Group Members):

Sr. No	Name	Section - Roll number
1.	Shahid Sheikh	58
2.	Shashank Dahake	59
3.	Shehzan Sheikh	60
4.	Vidyut Chakrabarti	68

List of Design Patterns used:

Sr. No	Pattern Name
1.	Observer Pattern
2.	Strategy Pattern
3.	Singleton Pattern
4.	Template Pattern

Application Abstract:

This personal finance management system serves as a comprehensive tool for managing, planning, and organizing personal finances. It offers various features that provide users with control over their income, expenses, investments, and budgeting, streamlining the way they handle their financial activities. Here's an overview of its functionalities:
1. Financial Monitoring
The application enables users to monitor their financial activities in real time. It allows users to add multiple accounts (such as checking or savings accounts) and provides alerts whenever a new financial transaction occurs. Whether it's an income deposit or an expense, users are kept informed of their current financial status without delays. This functionality ensures that users stay aware of their spending habits, preventing overspending and aiding in better financial planning.
2. Data Management
To ensure users have a clear and consistent record of their financial activities, the system maintains centralized financial data. This feature keeps track of recent income and expense records, and users can easily access or update these data points as needed. The centralized nature of the data management ensures that all financial information is stored accurately, allowing users to review their financial history or make necessary updates without discrepancies.
3. Investment Planning
This system allows users to plan their investments based on their financial goals. Whether users want to focus on capital growth through high-risk investments or prefer low-risk strategies aimed at capital preservation, the application offers various investment options. Users can adjust their investment

strategies over time, adapting to changing financial goals. The investment planning feature provides flexibility, enabling users to make informed decisions about how they want to allocate their funds toward achieving future financial security.
4. Budget Allocation
The application includes a structured budgeting tool that helps users allocate their income across essential spending categories. It divides income into necessities (such as housing, utilities, and food), discretionary spending (such as entertainment and dining out), and savings (including retirement or emergency funds). The budget allocation tool provides users with a detailed summary of how their income is distributed across these categories, giving them a clear understanding of their financial priorities and helping them maintain a balanced approach to spending and saving.
Summary
This personal finance management system offers users a comprehensive set of tools to track financial activities, manage data, plan investments, and allocate budgets. By combining real-time financial monitoring with centralized data management, customized investment planning, and a detailed budgeting tool, it provides users with a streamlined and organized approach to managing their personal finances. This enables users to maintain control over their financial goals while gaining insight into their spending and investment patterns.

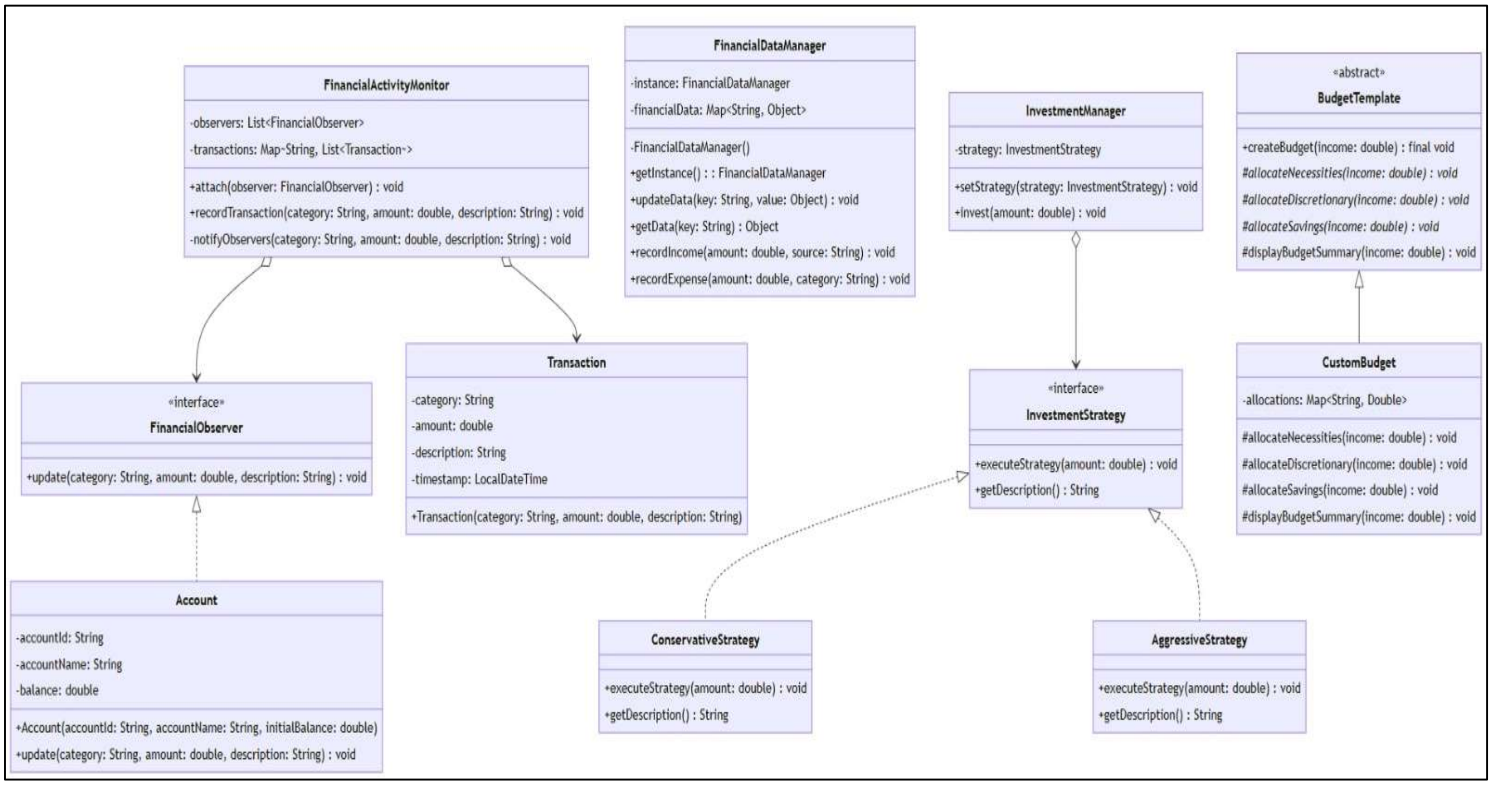

FinancialActivityMonitor
-observers: List<FinancialObserver>
-transactions: Map~String, List~Transaction~~
+attach(observer: FinancialObserver) : void
+recordTransaction(category: String, amount: double, description: String) : void
-notifyObservers(category: String, amount: double, description: String) : void
FinancialDataManager
-instance: FinancialDataManager
-financialData: Map<String, Object>
-FinancialDataManager()
+getInstance() : : FinancialDataManager
+updateData(key: String, value: Object) : void
+getData(key: String) : Object
+recordIncome(amount: double, source: String) : void
+recordExpense(amount: double, category: String) : void
InvestmentManager
-strategy: InvestmentStrategy
+setStrategy(strategy: InvestmentStrategy) : void
+invest(amount: double) : void
«abstract»
BudgetTemplate
+createBudget(income: double) : final void
#allocateNecessities(income: double) : void
#allocateDiscretionary(income: double) : void
#allocateSavings(income: double) : void
#displayBudgetSummary(income: double) : void
«interface»
FinancialObserver
+update(category: String, amount: double, description: String) : void
Transaction
-category: String
-amount: double
-description: String
-timestamp: LocalDateTime
+Transaction(category: String, amount: double, description: String)
«interface»
InvestmentStrategy
+executeStrategy(amount: double) : void
+getDescription() : String
CustomBudget
-allocations: Map<String, Double>
#allocateNecessities(income: double) : void
#allocateDiscretionary(income: double) : void
#allocateSavings(income: double) : void
#displayBudgetSummary(income: double) : void
Account
-accountId: String
-accountName: String
-balance: double
+Account(accountId: String, accountName: String, initialBalance: double)
+update(category: String, amount: double, description: String) : void
ConservativeStrategy
+executeStrategy(amount: double) : void
+getDescription() : String
AggressiveStrategy
+executeStrategy(amount: double) : void
+getDescription() : String

Figure 1. Overall class diagram of the application

- **Design pattern abstracts & Code:**

Singleton Pattern

Purpose:
The Singleton Pattern is a design strategy aimed at ensuring a class has only one instance throughout the application, providing a unique access point to that instance. This is essential in scenarios where multiple instances of a class could lead to conflicting data or operations, especially when managing shared resources.

Application in FinancialDataManager:
In this application, the Singleton Pattern is central to the FinancialDataManager class, as depicted in the class diagram (fig. 2). The class is responsible for managing all financial data, serving as a unified source for data related to income, expenses, and other financial records. By implementing this pattern, the FinancialDataManager becomes the sole authority on financial data across the entire application, eliminating potential data duplication and conflicts.

Rationale:
The need for a Singleton Pattern in financial systems arises from the importance of data accuracy and consistency. In applications where multiple components need to access or modify shared data, having multiple instances of a class could lead to issues such as data redundancy, misalignment, or concurrency problems. By utilizing a Singleton, FinancialDataManager ensures that all parts of the application interact with a single, reliable instance, thereby promoting data integrity and operational consistency. This design choice is common in systems that require centralized control, like configuration managers, logging frameworks, or, as in this case, financial data managers.

Implementation Details:
The Singleton pattern in FinancialDataManager is realized through the getInstance() method, which enforces lazy instantiation, synchronized access, and static control, as depicted in the provided structure. Here's a breakdown of each aspect:

- Lazy Instantiation: The instance of FinancialDataManager is created only when it's first needed, optimizing resource usage by avoiding premature object creation.
- Synchronized Access: To ensure thread safety, getInstance() is synchronized. This prevents multiple threads from creating separate instances of FinancialDataManager when accessed concurrently, maintaining a single, consistent instance.
- Static Control: The instance is managed through a static attribute (instance), making it accessible without requiring an object of FinancialDataManager. This approach centralizes control and simplifies access to the single instance.

Benefits of This Design Choice:
Using the Singleton Pattern for FinancialDataManager is especially fitting for financial applications, where data integrity and consistency are paramount. By preventing the creation of multiple instances, this pattern guarantees that financial data remains accurate and free from conflicts. Additionally, the centralized access point simplifies the application's architecture, enhancing maintainability and reducing the potential for errors. This approach minimizes redundancy and ensures that all financial operations reference a unified, authoritative source, which is crucial in financial systems.

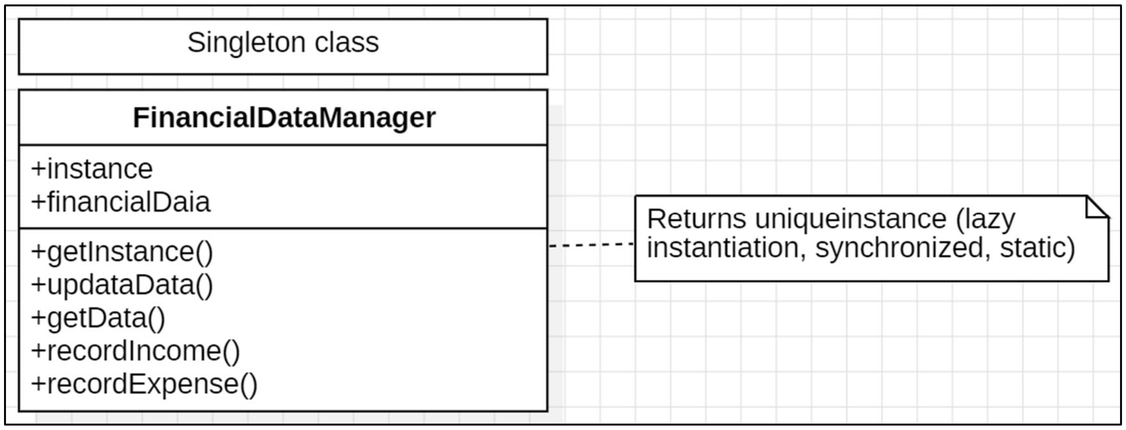
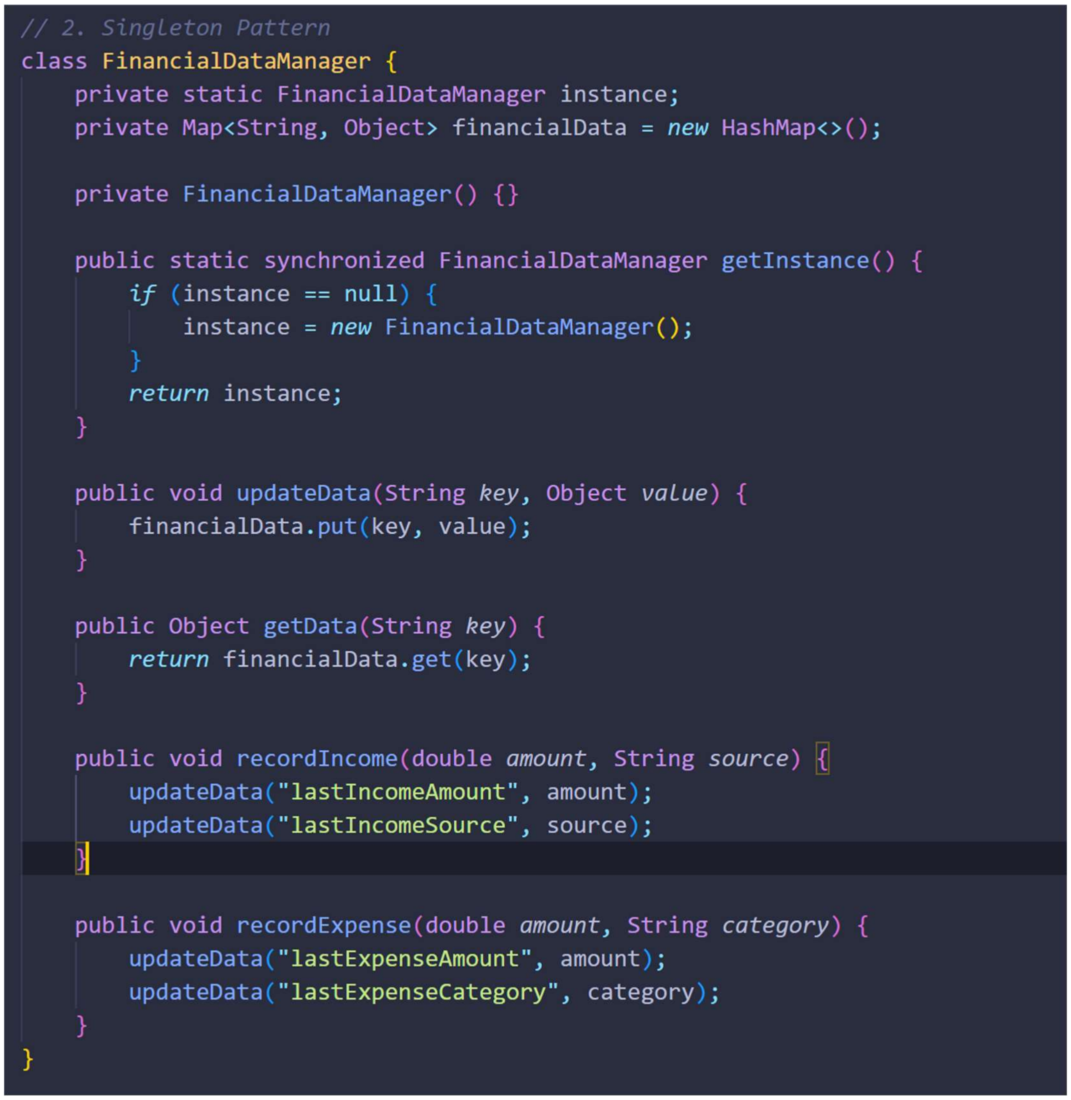


Figure 2. Singleton Design pattern (in app)

Code Snippet:



```
// 2. Singleton Pattern
class FinancialDataManager {
    private static FinancialDataManager instance;
    private Map<String, Object> financialData = new HashMap<>();

    private FinancialDataManager() {}

    public static synchronized FinancialDataManager getInstance() {
        if (instance == null) {
            instance = new FinancialDataManager();
        }
        return instance;
    }

    public void updateData(String key, Object value) {
        financialData.put(key, value);
    }

    public Object getData(String key) {
        return financialData.get(key);
    }

    public void recordIncome(double amount, String source) {
        updateData("lastIncomeAmount", amount);
        updateData("lastIncomeSource", source);
    }

    public void recordExpense(double amount, String category) {
        updateData("lastExpenseAmount", amount);
        updateData("lastExpenseCategory", category);
    }
}
```

Observer Pattern

Purpose:

The Observer Pattern establishes a one-to-many relationship, ensuring that when an object's state changes, all dependent objects are automatically notified and updated. This pattern is essential in scenarios where updates in one component must be communicated across multiple dependent components in real-time.

Application in FinancialActivityMonitor:

The Observer Pattern is implemented in the FinancialActivityMonitor class, which is responsible for managing transactions and notifying its observers—such as Account instances—whenever a transaction update occurs. As shown in the class diagram, FinancialActivityMonitor keeps a list of observers (in this case, Account instances) and uses a set of methods—attach(), detach(), and notifyObservers()—to manage and inform these observers of any state changes in transactions.

Rationale:

The Observer Pattern is ideal in situations where multiple components must respond to changes in a central component, making it highly applicable in financial applications. Here, each Account needs immediate updates on transactions to reflect accurate balances and transaction histories. This setup is analogous to systems such as real-time network monitoring, where nodes must be aware of new devices. In this financial application, the Observer Pattern ensures that all accounts stay synchronized with transaction changes, supporting the accuracy and consistency of financial data across the application.

Implementation Details:

In FinancialActivityMonitor, the Observer Pattern is applied by designating it as the subject (the entity that changes). The class maintains a list of Account observers, which are registered using the attach() method and removed using detach(). When a transaction update occurs, notifyObservers() iterates through the list of observers and calls update() on each observer to relay the new transaction details.

- Observer Management: Observers can be dynamically added or removed, offering flexibility and scalability as the application grows.
- Real-Time Updates: By invoking update() on each Account observer, FinancialActivityMonitor ensures each account has real-time access to the latest transaction information.
- Loose Coupling: Observers only rely on the subject's state via getState(), minimizing dependencies and simplifying maintenance.

Benefits of This Design Choice:
The Observer Pattern offers a scalable and flexible approach, allowing new Account instances to be easily added as observers without changing the core implementation of FinancialActivityMonitor. This loose coupling ensures that FinancialActivityMonitor and Account remain independent, enhancing modularity. Furthermore, the pattern's ability to handle a one-to-many relationship aligns well with financial applications, where numerous accounts must subscribe to transaction updates. This design improves application flexibility and robustness, as more accounts can seamlessly receive updates, maintaining data consistency and integrity.

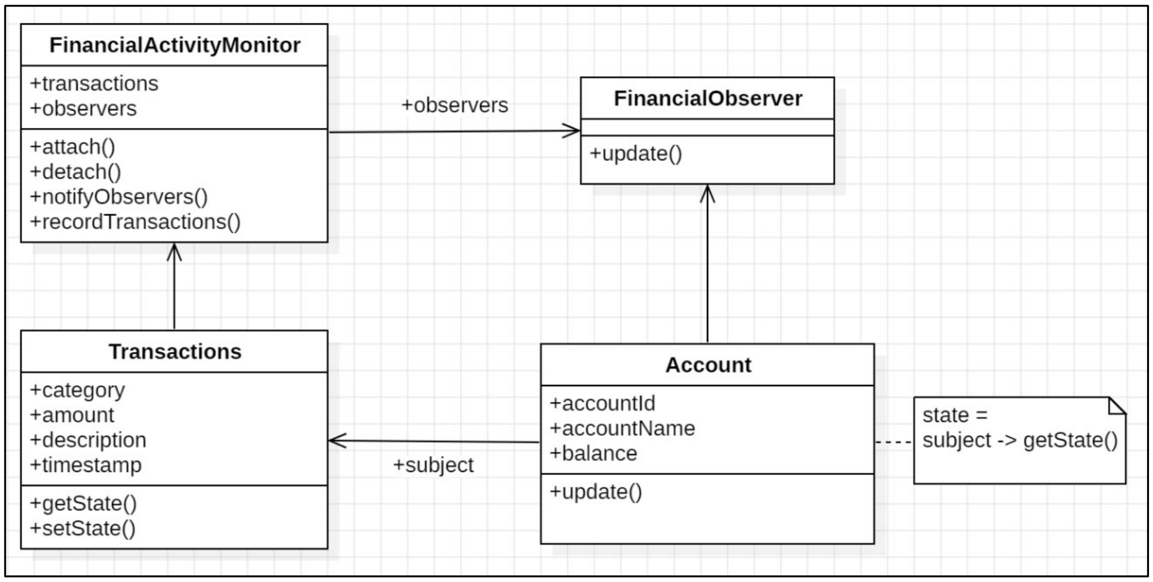
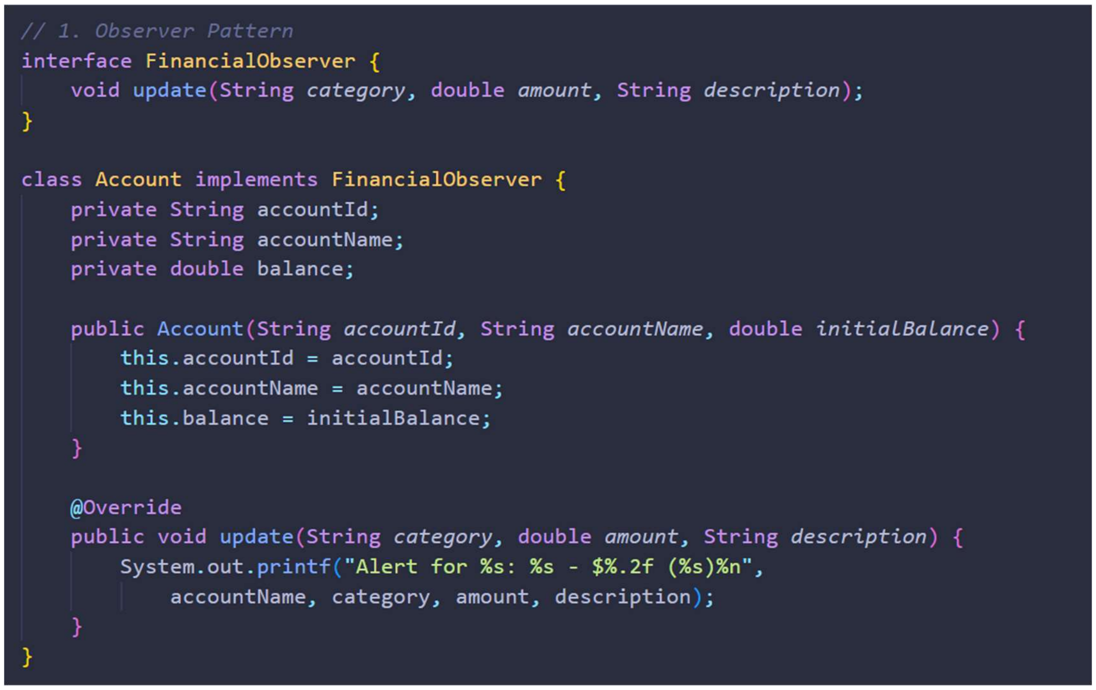


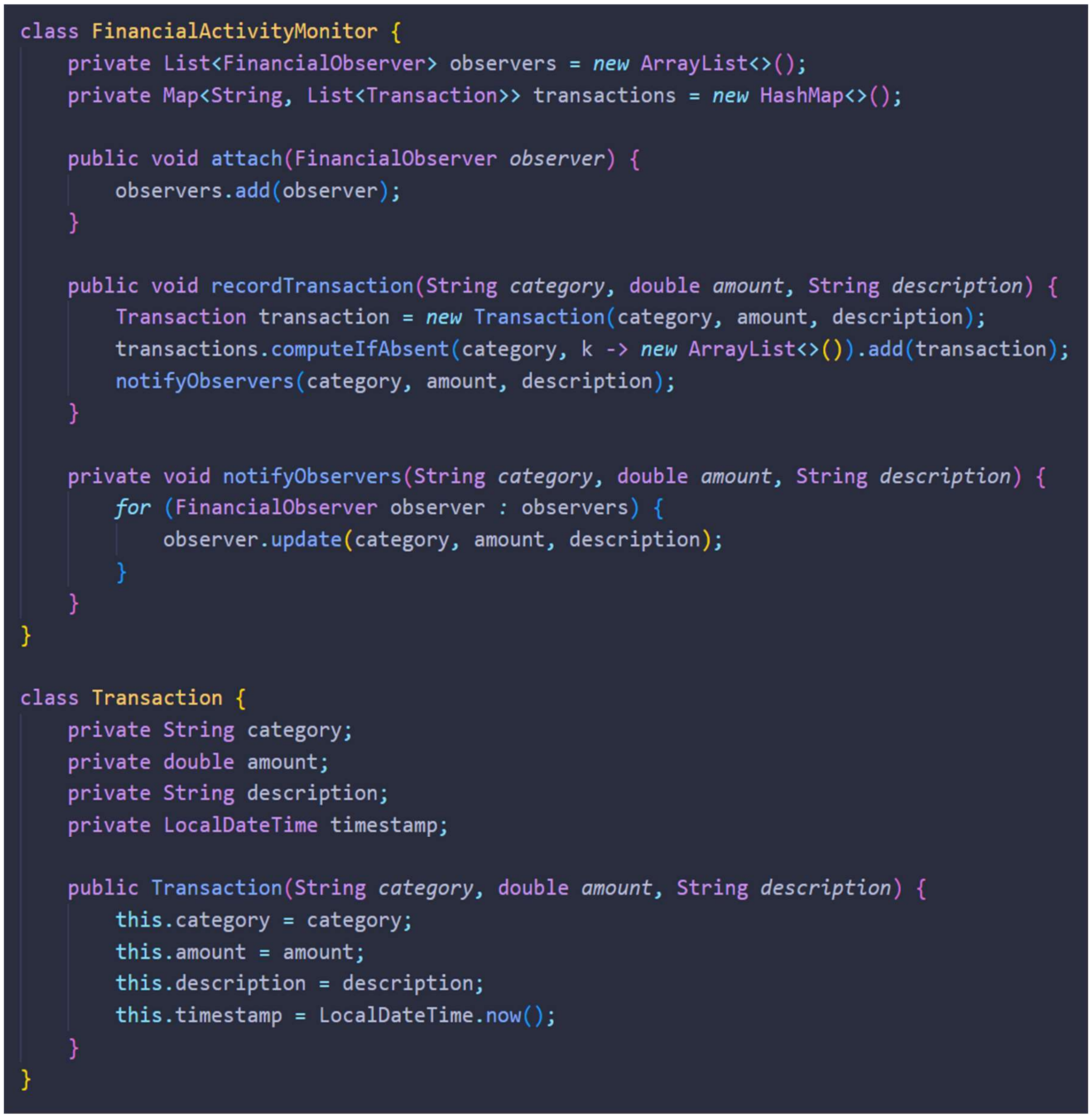
Figure 3: Observer pattern implementation (in app)

Code Snippet: (Observer interface & concrete observer)



```
// 1. Observer Pattern
interface FinancialObserver {
    void update(String category, double amount, String description);
}

class Account implements FinancialObserver {
    private String accountId;
    private String accountName;
    private double balance;

    public Account(String accountId, String accountName, double initialBalance) {
        this.accountId = accountId;
        this.accountName = accountName;
        this.balance = initialBalance;
    }

    @Override
    public void update(String category, double amount, String description) {
        System.out.printf("Alert for %s: %s - $%.2f (%s)%n",
            accountName, category, amount, description);
    }
}
```

Code Snippet (Subject & individual transactions)



```
class FinancialActivityMonitor {
    private List<FinancialObserver> observers = new ArrayList<>();
    private Map<String, List<Transaction>> transactions = new HashMap<>();

    public void attach(FinancialObserver observer) {
        observers.add(observer);
    }

    public void recordTransaction(String category, double amount, String description) {
        Transaction transaction = new Transaction(category, amount, description);
        transactions.computeIfAbsent(category, k -> new ArrayList<>()).add(transaction);
        notifyObservers(category, amount, description);
    }

    private void notifyObservers(String category, double amount, String description) {
        for (FinancialObserver observer : observers) {
            observer.update(category, amount, description);
        }
    }
}

class Transaction {
    private String category;
    private double amount;
    private String description;
    private LocalDateTime timestamp;

    public Transaction(String category, double amount, String description) {
        this.category = category;
        this.amount = amount;
        this.description = description;
        this.timestamp = LocalDateTime.now();
    }
}
```

Strategy Pattern

Purpose:
The Strategy Pattern allows the definition of a family of algorithms, encapsulating each one so they can be selected interchangeably at runtime. This approach is valuable when an application requires flexibility in choosing among multiple behaviors, enabling dynamic adaptation based on specific conditions or preferences.

Application in InvestmentManager:
The Strategy Pattern is utilized in the InvestmentManager class, which dynamically adapts various investment strategies based on user preferences, such as risk tolerance and financial goals. As shown in the design, the InvestmentManager takes an InvestmentStrategy implementation, which defines specific investing behaviors. This setup enables InvestmentManager to apply different strategies—like ConservativeStrategy or AggressiveStrategy—tailored to each user's profile.

Rationale:
In systems where different algorithms or behaviors may be suitable under varying circumstances, the Strategy Pattern provides a flexible means of selecting the appropriate behavior at runtime. For

instance, in sorting applications, different comparison strategies might be chosen based on data type or sorting requirements. Here, InvestmentManager leverages this pattern to provide a range of investment strategies that suit diverse user needs. By allowing interchangeable strategies, the application can meet the unique risk tolerances and financial objectives of individual users, enhancing its adaptability and responsiveness.

Implementation Details:
The Strategy Pattern in InvestmentManager is implemented by accepting an InvestmentStrategy object, which encapsulates specific investment algorithms. Through the invest() method, InvestmentManager invokes the selected strategy, determining the investment behavior based on the user's profile.

- Encapsulation of Strategies: Each investment strategy is encapsulated in its own class (e.g., ConservativeStrategy, AggressiveStrategy), ensuring that the investment logic is isolated from the core InvestmentManager.
- Interchangeability: InvestmentManager can switch strategies easily by swapping the InvestmentStrategy implementation, allowing seamless transitions between strategies based on the user's evolving goals and risk level.
- Flexibility and Modularity: New strategies can be added without modifying the existing code in InvestmentManager, supporting scalability and ease of maintenance.

Benefits of This Design Choice:
The Strategy Pattern decouples investment behaviors from the main application logic, promoting modularity and customization. This separation allows for easy expansion, as new investment strategies can be introduced without altering the existing InvestmentManager structure. This flexibility aligns with the application's need to offer a wide range of customizable investment options, catering to diverse user preferences while improving both code maintainability and readability. The Strategy Pattern's design thus enables InvestmentManager to accommodate and dynamically adapt to various investment approaches, ensuring that users receive a tailored experience based on their individual risk tolerance and investment goals.

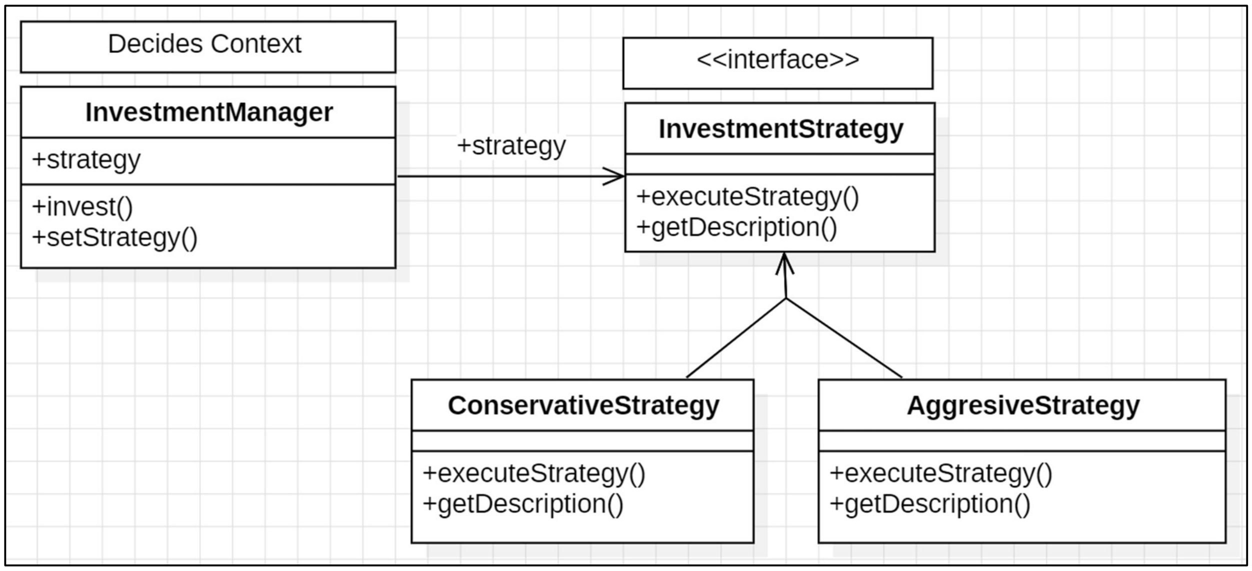
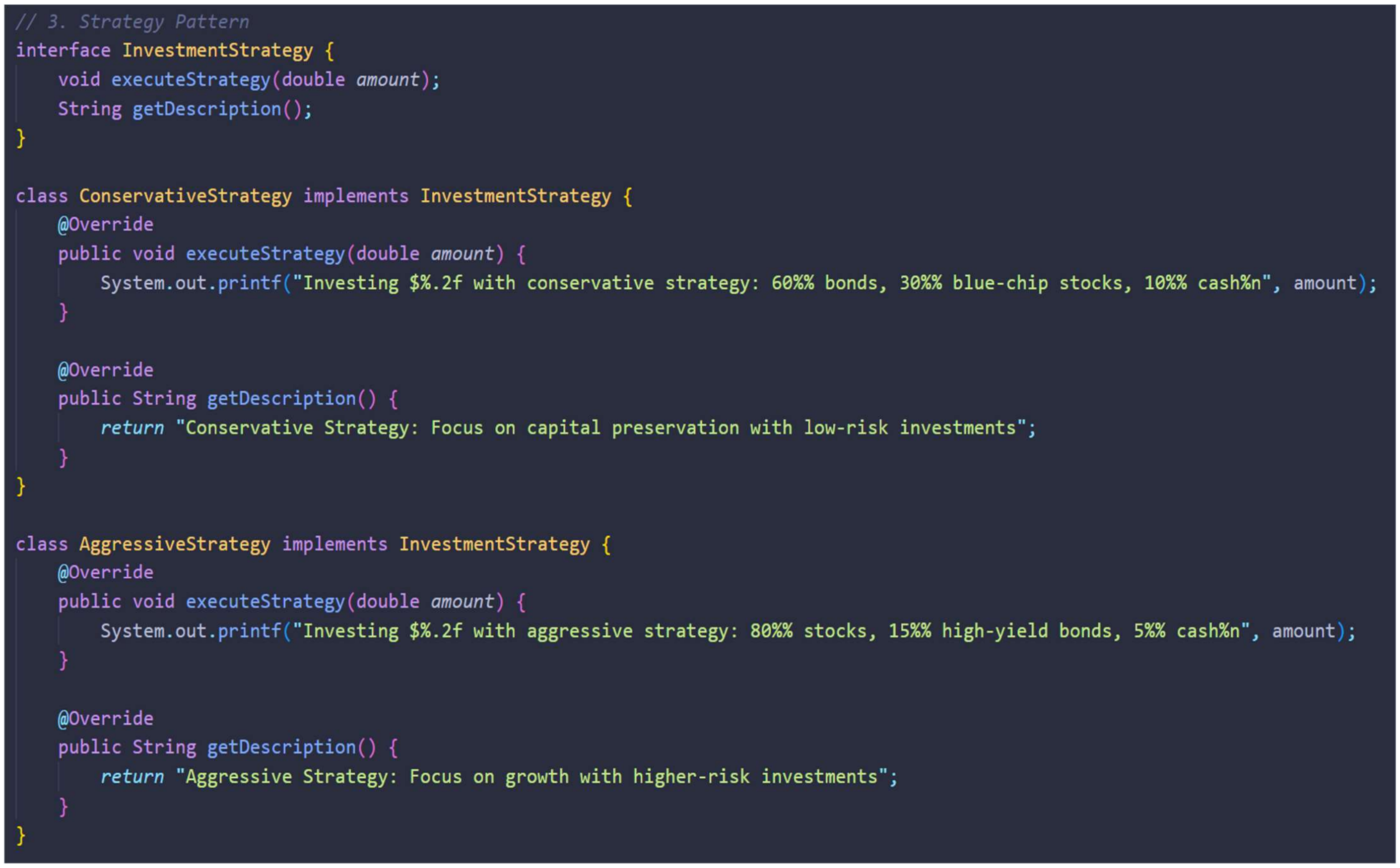


Figure 4: Strategy pattern implementation (in app)

Code Snippet:



```java
// 3. Strategy Pattern
interface InvestmentStrategy {
    void executeStrategy(double amount);
    String getDescription();
}

class ConservativeStrategy implements InvestmentStrategy {
    @Override
    public void executeStrategy(double amount) {
        System.out.printf("Investing $%.2f with conservative strategy: 60%% bonds, 30%% blue-chip stocks, 10%% cash%n", amount);
    }

    @Override
    public String getDescription() {
        return "Conservative Strategy: Focus on capital preservation with low-risk investments";
    }
}

class AggressiveStrategy implements InvestmentStrategy {
    @Override
    public void executeStrategy(double amount) {
        System.out.printf("Investing $%.2f with aggressive strategy: 80%% stocks, 15%% high-yield bonds, 5%% cash%n", amount);
    }

    @Override
    public String getDescription() {
        return "Aggressive Strategy: Focus on growth with higher-risk investments";
    }
}
```

Template Method Pattern

Purpose:

The Template Method Pattern is a behavioral design pattern that defines the skeleton of an algorithm in a method, allowing subclasses to override specific steps without changing the algorithm's overall structure. This approach is effective when a consistent sequence of steps is needed, but specific actions within the sequence require flexibility.

Application in BudgetTemplate:

The Template Method Pattern is implemented in the BudgetTemplate abstract class, which serves as the foundation for creating budgets. The BudgetTemplate class provides a standardized method, createBudget(), that outlines the structured process for setting up a budget. The CustomBudget subclass, derived from BudgetTemplate, customizes specific steps of this process, such as allocation methods for various categories (e.g., necessities, discretionary spending, savings), adapting the budget creation to meet user-specific requirements.

Rationale:

The Template Method Pattern is beneficial in scenarios where an application requires a stable core algorithm with the flexibility to customize certain steps. This pattern is comparable to file processing, where the general process (opening, reading, and closing a file) remains the same, but specific behaviors differ based on file type. Similarly, BudgetTemplate provides a standardized budget creation structure, while subclasses like CustomBudget adapt portions of this process to suit different budgeting needs, aligning with user preferences and financial goals.

Implementation Details:

In BudgetTemplate, the Template Method Pattern is applied through the createBudget() method, which defines the overall process for budget creation. Specific steps within this process, such as category allocations, are left abstract, allowing subclasses to implement these details.

- Core Structure Defined in Template: The createBudget() method in BudgetTemplate outlines the sequence for setting up a budget, providing a framework that all budgets follow.
- Customizable Steps in Subclasses: The CustomBudget subclass provides concrete implementations for specific budget allocations, tailoring the budget to meet distinct user preferences in areas like necessities, discretionary spending, and savings.
- Reuse and Consistency: By encapsulating the general process in BudgetTemplate, the pattern promotes consistency across different budget types, minimizing redundant code and enhancing maintainability.

Benefits of This Design Choice:
The Template Method Pattern enforces a structured approach to budget creation, ensuring a consistent foundation while allowing specific steps to vary in subclasses. This design enhances adaptability by enabling CustomBudget and other subclasses to implement unique budget allocations without altering the overall structure. Additionally, the pattern improves code reuse, as the core logic in BudgetTemplate can be applied across various budget types. This structured flexibility is essential for aligning budgeting processes with diverse user needs, enabling a balance between consistency and customization.

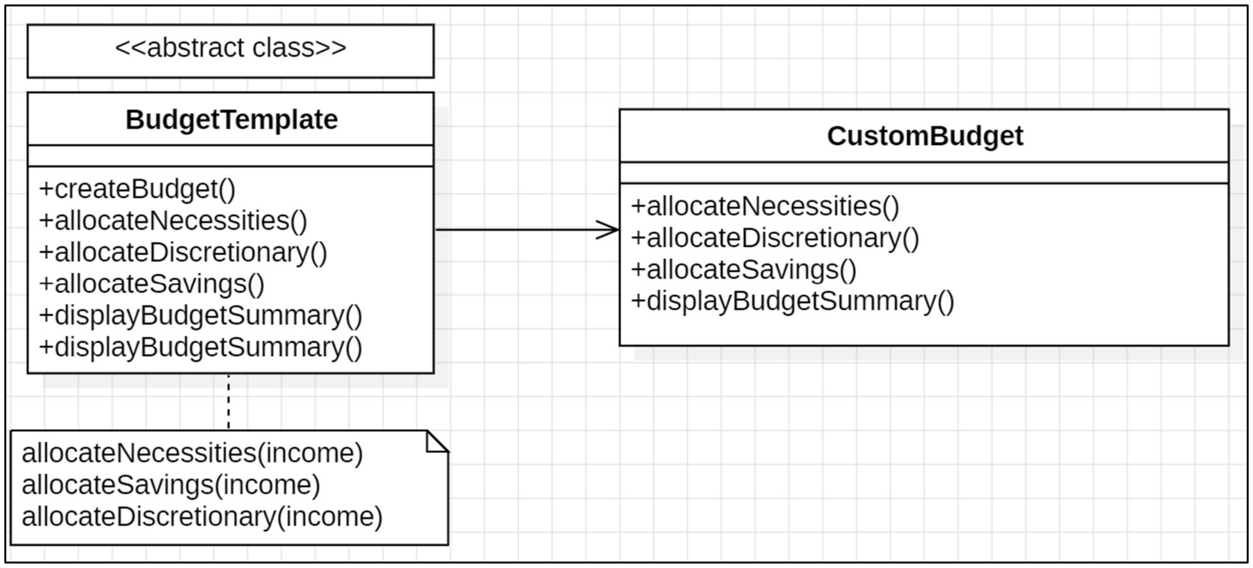
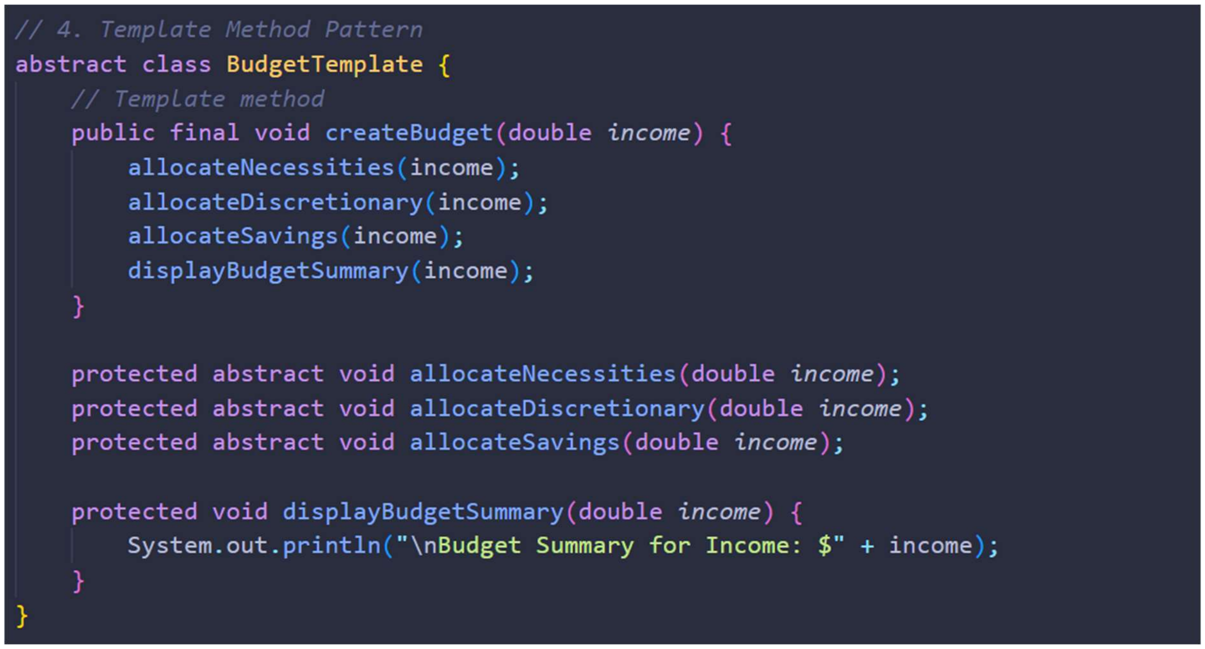


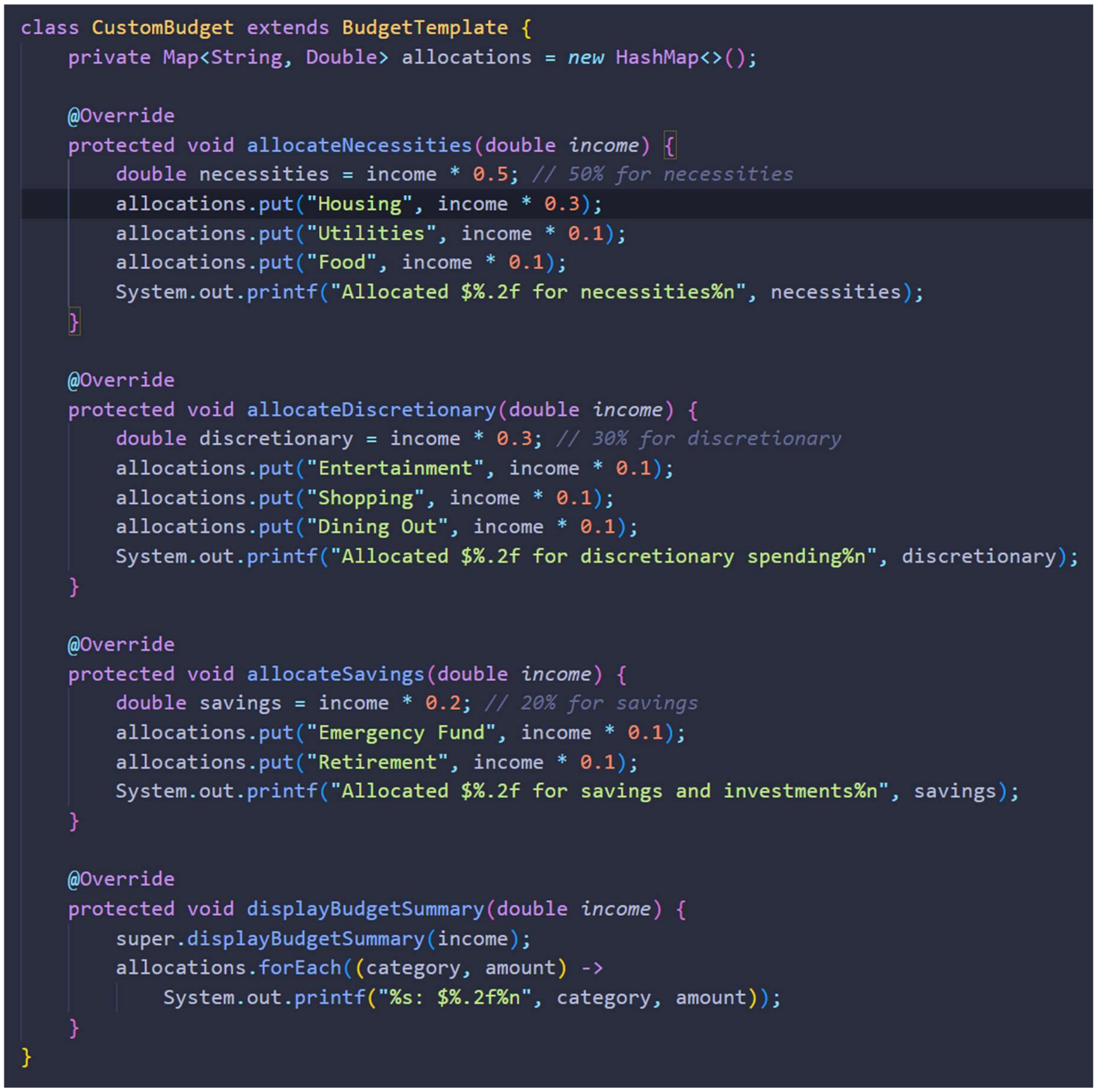
Figure 5: Template pattern implementation (in app)

Code Snippet:



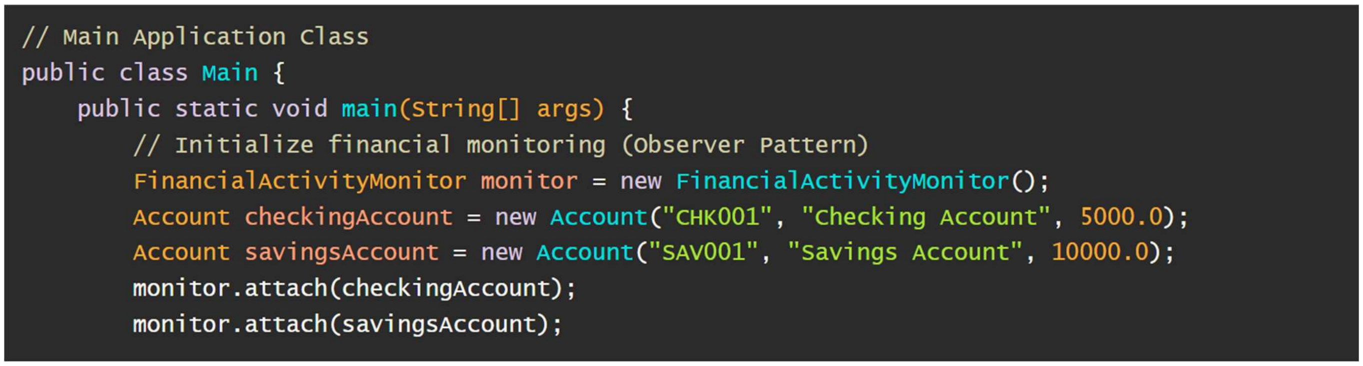
```
// 4. Template Method Pattern
abstract class BudgetTemplate {
    // Template method
    public final void createBudget(double income) {
        allocateNecessities(income);
        allocateDiscretionary(income);
        allocateSavings(income);
        displayBudgetSummary(income);
    }

    protected abstract void allocateNecessities(double income);
    protected abstract void allocateDiscretionary(double income);
    protected abstract void allocateSavings(double income);

    protected void displayBudgetSummary(double income) {
        System.out.println("\nBudget Summary for Income: $" + income);
    }
}
```

Code Snippet (Concrete Budget)

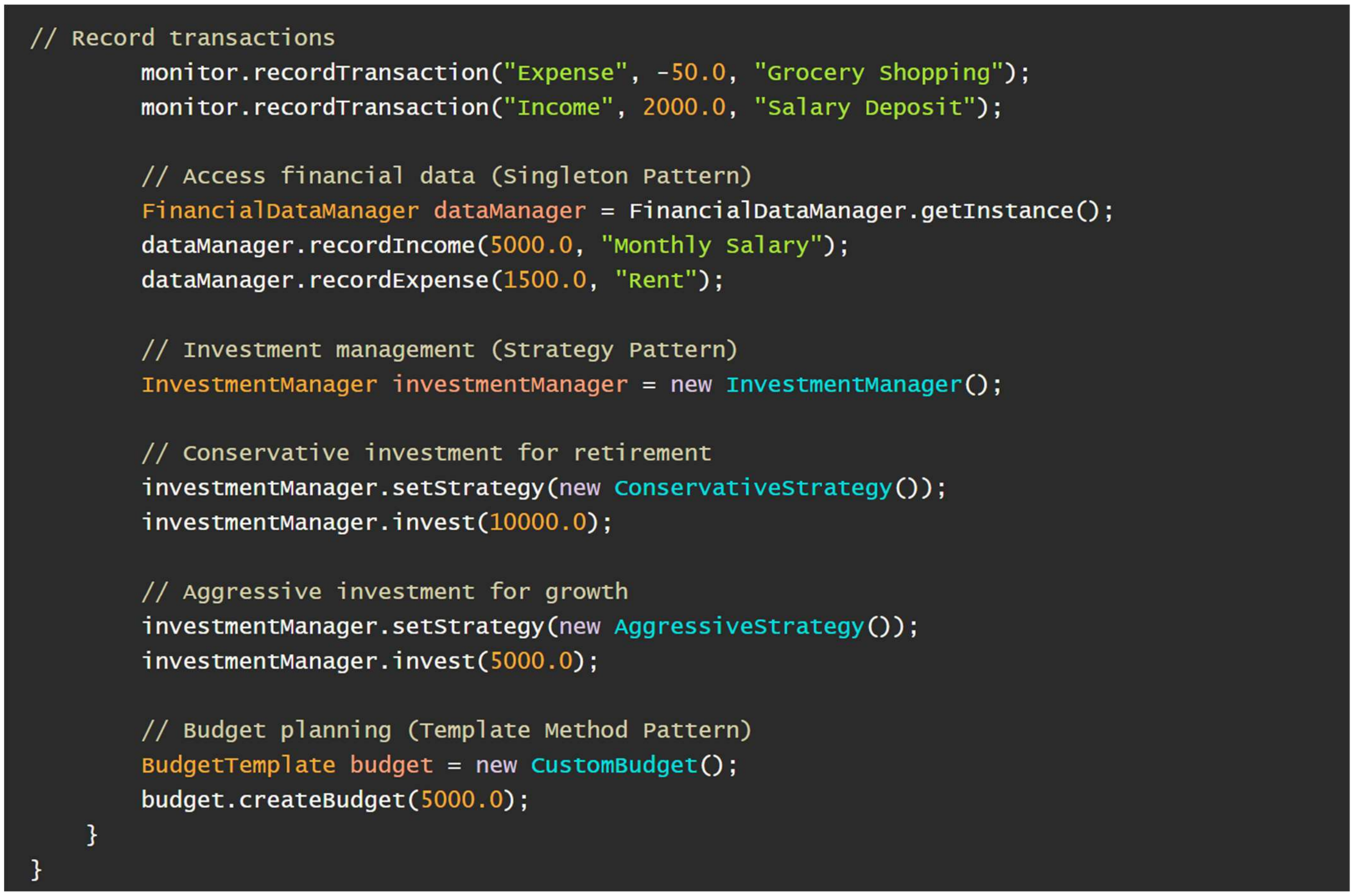


```java
class CustomBudget extends BudgetTemplate {
    private Map<String, Double> allocations = new HashMap<>();

    @Override
    protected void allocateNecessities(double income) {
        double necessities = income * 0.5; // 50% for necessities
        allocations.put("Housing", income * 0.3);
        allocations.put("Utilities", income * 0.1);
        allocations.put("Food", income * 0.1);
        System.out.printf("Allocated $%.2f for necessities%n", necessities);
    }

    @Override
    protected void allocateDiscretionary(double income) {
        double discretionary = income * 0.3; // 30% for discretionary
        allocations.put("Entertainment", income * 0.1);
        allocations.put("Shopping", income * 0.1);
        allocations.put("Dining Out", income * 0.1);
        System.out.printf("Allocated $%.2f for discretionary spending%n", discretionary);
    }

    @Override
    protected void allocateSavings(double income) {
        double savings = income * 0.2; // 20% for savings
        allocations.put("Emergency Fund", income * 0.1);
        allocations.put("Retirement", income * 0.1);
        System.out.printf("Allocated $%.2f for savings and investments%n", savings);
    }

    @Override
    protected void displayBudgetSummary(double income) {
        super.displayBudgetSummary(income);
        allocations.forEach((category, amount) ->
            System.out.printf("%s: $%.2f%n", category, amount));
    }
}
```

Main.java: (Sample execution):

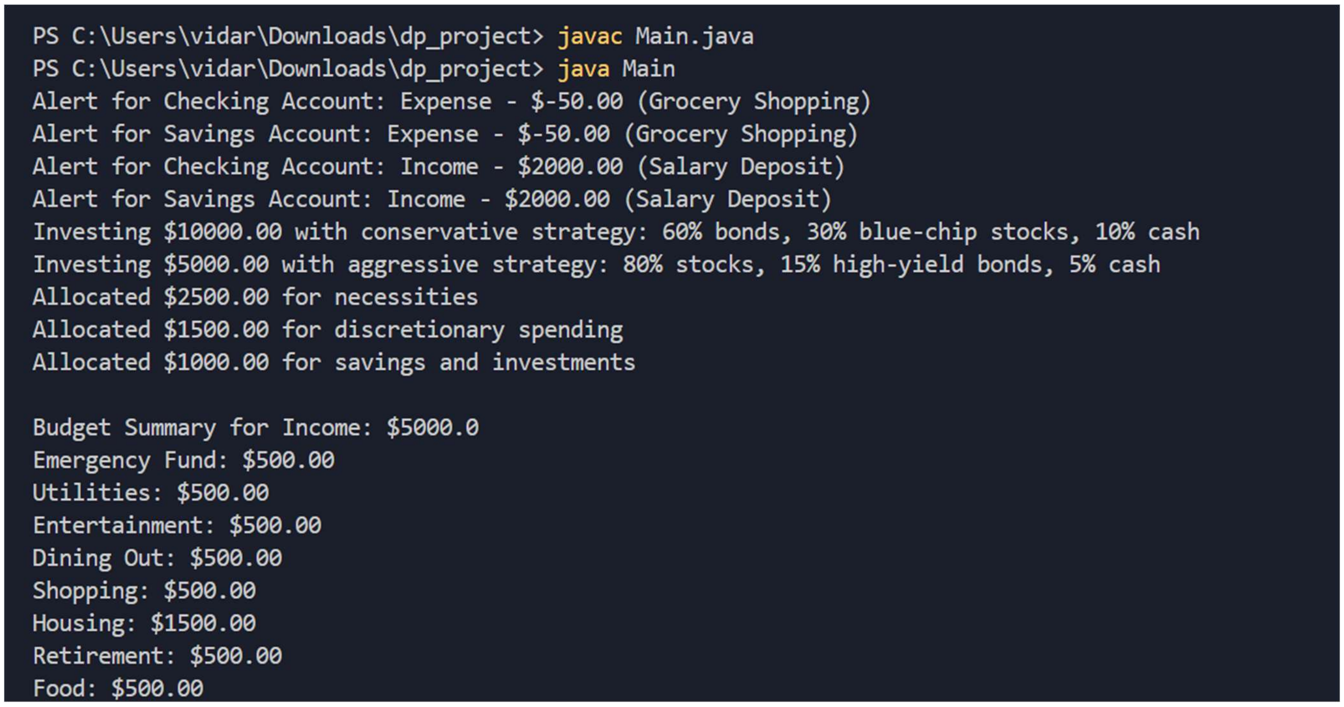


```java
// Main Application Class
public class Main {
    public static void main(String[] args) {
        // Initialize financial monitoring (Observer Pattern)
        FinancialActivityMonitor monitor = new FinancialActivityMonitor();
        Account checkingAccount = new Account("CHK001", "Checking Account", 5000.0);
        Account savingsAccount = new Account("SAV001", "Savings Account", 10000.0);
        monitor.attach(checkingAccount);
        monitor.attach(savingsAccount);
```



```java
// Record transactions
        monitor.recordTransaction("Expense", -50.0, "Grocery Shopping");
        monitor.recordTransaction("Income", 2000.0, "Salary Deposit");

        // Access financial data (Singleton Pattern)
        FinancialDataManager dataManager = FinancialDataManager.getInstance();
        dataManager.recordIncome(5000.0, "Monthly Salary");
        dataManager.recordExpense(1500.0, "Rent");

        // Investment management (Strategy Pattern)
        InvestmentManager investmentManager = new InvestmentManager();

        // Conservative investment for retirement
        investmentManager.setStrategy(new ConservativeStrategy());
        investmentManager.invest(10000.0);

        // Aggressive investment for growth
        investmentManager.setStrategy(new AggressiveStrategy());
        investmentManager.invest(5000.0);

        // Budget planning (Template Method Pattern)
        BudgetTemplate budget = new CustomBudget();
        budget.createBudget(5000.0);
    }
}
```

Screenshots of sample run:



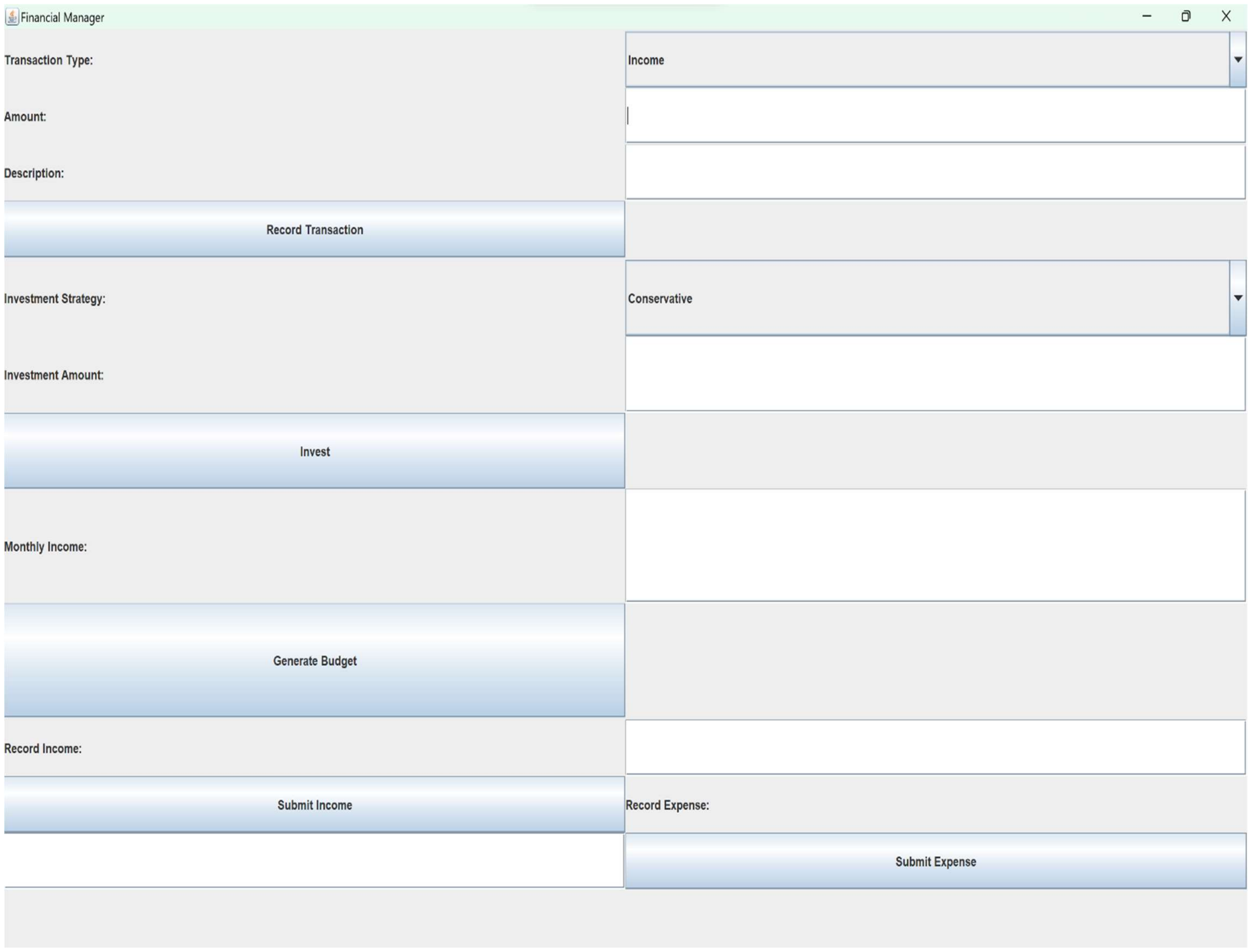
```
PS C:\Users\vidar\Downloads\dp_project> javac Main.java
PS C:\Users\vidar\Downloads\dp_project> java Main
Alert for Checking Account: Expense - $-50.00 (Grocery Shopping)
Alert for Savings Account: Expense - $-50.00 (Grocery Shopping)
Alert for Checking Account: Income - $2000.00 (Salary Deposit)
Alert for Savings Account: Income - $2000.00 (Salary Deposit)
Investing $10000.00 with conservative strategy: 60% bonds, 30% blue-chip stocks, 10% cash
Investing $5000.00 with aggressive strategy: 80% stocks, 15% high-yield bonds, 5% cash
Allocated $2500.00 for necessities
Allocated $1500.00 for discretionary spending
Allocated $1000.00 for savings and investments

Budget Summary for Income: $5000.0
Emergency Fund: $500.00
Utilities: $500.00
Entertainment: $500.00
Dining Out: $500.00
Shopping: $500.00
Housing: $1500.00
Retirement: $500.00
Food: $500.00
```

GUI (Code with Swing):

```java
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class FinancialAppGUI {
    private FinancialActivityMonitor monitor;
    private InvestmentManager investmentManager;
    private BudgetTemplate budget;
    private FinancialDataManager dataManager;

    public FinancialAppGUI() {
        monitor = new FinancialActivityMonitor();
        investmentManager = new InvestmentManager();
        budget = new CustomBudget();
        dataManager = FinancialDataManager.getInstance();

        // Setting up the main JFrame
        JFrame frame = new JFrame("Financial Manager");
        frame.setSize(600, 600);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setLayout(new GridLayout(4, 1));

        // Panel 1: Transaction Panel (Observer Pattern)
        JPanel transactionPanel = new JPanel();
        transactionPanel.setLayout(new GridLayout(4, 2));

        JLabel transactionTypeLabel = new JLabel("Transaction Type: ");
        String[] transactionTypes = {"Income", "Expense"};
       JComboBox<String> transactionTypeBox = new
       JComboBox<>(transactionTypes);

        JLabel amountLabel = new JLabel("Amount: ");
        JTextField amountField = new JTextField();

        JLabel descriptionLabel = new JLabel("Description: ");
        JTextField descriptionField = new JTextField();

        JButton recordTransactionBtn = new JButton("Record Transaction");

        transactionPanel.add(transactionTypeLabel);
        transactionPanel.add(transactionTypeBox);
        transactionPanel.add(amountLabel);
        transactionPanel.add(amountField);
        transactionPanel.add(descriptionLabel);
        transactionPanel.add(descriptionField);
        transactionPanel.add(recordTransactionBtn);

        // Transaction button action (Observer Pattern)
        recordTransactionBtn.addActionListener(e -> {
           double amount = Double.parseDouble(amountField.getText());
        String description = descriptionField.getText();
        String category = (String) transactionTypeBox.getSelectedItem();
        monitor.recordTransaction(category, amount, description);
        JOptionPane.showMessageDialog(frame, "Transaction Recorded!");
    });

    // Panel 2: Investment Strategy Panel (Strategy Pattern)
    JPanel investmentPanel = new JPanel();
    investmentPanel.setLayout(new GridLayout(3, 2));

    JLabel strategyLabel = new JLabel("Investment Strategy: ");
    String[] strategies = {"Conservative", "Aggressive"};
    JComboBox<String> strategyBox = new JComboBox<>(strategies);

    JLabel investAmountLabel = new JLabel("Investment Amount: ");
    JTextField investAmountField = new JTextField();

    JButton investBtn = new JButton("Invest");

    investmentPanel.add(strategyLabel);
    investmentPanel.add(strategyBox);
    investmentPanel.add(investAmountLabel);
    investmentPanel.add(investAmountField);
    investmentPanel.add(investBtn);

    // Investment button action (Strategy Pattern)
    investBtn.addActionListener(e -> {
    double investAmount =
    Double.parseDouble(investAmountField.getText());
        String selectedStrategy = (String) strategyBox.getSelectedItem();
        if (selectedStrategy.equals("Conservative")) {
            investmentManager.setStrategy(new ConservativeStrategy());
        } else {
            investmentManager.setStrategy(new AggressiveStrategy());
        }
        investmentManager.invest(investAmount);
    });

    // Panel 3: Budget Generation Panel (Template Method Pattern)
    JPanel budgetPanel = new JPanel();
    budgetPanel.setLayout(new GridLayout(2, 2));

    JLabel incomeLabel = new JLabel("Monthly Income: ");
    JTextField incomeField = new JTextField();

    JButton generateBudgetBtn = new JButton("Generate Budget");

    budgetPanel.add(incomeLabel);
    budgetPanel.add(incomeField);
    budgetPanel.add(generateBudgetBtn);

    // Budget button action (Template Method Pattern)


    generateBudgetBtn.addActionListener(e -> {
        double income = Double.parseDouble(incomeField.getText());
```

```
            budget.createBudget(income);
        });

        // Panel 4: Financial Data Management (Singleton Pattern)
        JPanel dataManagerPanel = new JPanel();
        dataManagerPanel.setLayout(new GridLayout(4, 2));

        JLabel recordIncomeLabel = new JLabel("Record Income: ");
        JTextField recordIncomeField = new JTextField();

        JLabel recordExpenseLabel = new JLabel("Record Expense: ");
        JTextField recordExpenseField = new JTextField();

        JButton recordIncomeBtn = new JButton("Submit Income");
        JButton recordExpenseBtn = new JButton("Submit Expense");

        dataManagerPanel.add(recordIncomeLabel);
        dataManagerPanel.add(recordIncomeField);
        dataManagerPanel.add(recordIncomeBtn);
        dataManagerPanel.add(recordExpenseLabel);
        dataManagerPanel.add(recordExpenseField);
        dataManagerPanel.add(recordExpenseBtn);

        // Income and Expense button actions (Singleton Pattern)
        recordIncomeBtn.addActionListener(e -> {
            double income = Double.parseDouble(recordIncomeField.getText());
            dataManager.recordIncome(income, "User Entry");
            JOptionPane.showMessageDialog(frame, "Income Recorded!");
        });

        recordExpenseBtn.addActionListener(e -> {
            double expense =
            Double.parseDouble(recordExpenseField.getText());
            dataManager.recordExpense(expense, "User Entry");
            JOptionPane.showMessageDialog(frame, "Expense Recorded!");
        });

        // Adding all panels to the frame
        frame.add(transactionPanel);
        frame.add(investmentPanel);
        frame.add(budgetPanel);
        frame.add(dataManagerPanel);

        // Display the frame
        frame.setVisible(true);
    }
}
```

GUI Output:



Prof. R. Damdoo

Course Coordinator