Software Design 2

Project Assignment

Mnqobi Lisbon Jeza
230878369

Contents

1. Introduction

The creation of the "Jazzman Bank" application, an object-oriented bank account management system, is shown in this study. In order to develop a useful and intuitive banking solution, the project makes use of fundamental concepts of object-oriented software design (OOSD). This application's main objective is to enable users to do standard banking functions, including balance inquiries, deposits, withdrawals, and transaction histories, in a secure setting that necessitates user identification.

Realistic functions like managing transaction costs, keeping track of transactions, and enabling users to safely change their PIN are also built into the system. This application emphasizes the use of the Unified Modeling Language (UML) in system planning and design, in addition to showcasing the application of object-oriented concepts including encapsulation, inheritance, polymorphism, and composition.

2. Object Oriented Software Design (OOSD) Concepts

2.1. Objects and Classes

Classes and objects are the cornerstones of system design in object-oriented programming (OOP). A class can be compared to a template or blueprint that is used to construct objects. It outlines the methods (functions) and characteristics (data) that the objects may possess, as well as their structure and behaviour. A class only specifies the data and behaviour's that objects derived from the class will have; it does not itself contain data.

A **BankAccount** class, for instance, can define methods like **deposit()** and **withdraw()** in addition to fields like **accountNumber** and amount.

An instance of a class with unique attribute values is called an object. For instance, John's bank account, which was made using the **BankAccount** class, would be distinct from Sarah's bank account in terms of **accountNumber** and amount.

OOP relies heavily on the link between classes and objects since it enables developers to model real-world items in a modular and structured manner. You can instantiate as many objects as

required by generating a class only once, guaranteeing that the objects' structure and behaviour are maintained across the system. Additionally, because classes encapsulate behaviour, debugging and maintenance are considerably easier because modifications to an object's structure may be made to the class rather than updating each individual object.

2.2. Encapsulation

One of the fundamental ideas of object-oriented programming (OOP) is encapsulation. It describes the combination of methods (functions) that manipulate data and data (attributes) into a single unit, or class. More significantly, by limiting access and ensuring that objects control their own state, encapsulation aids with access control.

The concept of data hiding lies at the core of encapsulation. Usually, an object's attributes are not directly accessible from the outside and can only be accessed or changed using particular getters and setters. Access specifiers like private, public, and protected are typically used to do this. For instance, designating an attribute as private guarantees that external programs cannot directly access it. Alternatively, the class may have a **deposit()** function to safely alter the balance and a **getBalance()** way to get it.

This degree of control makes sure that an object's internal operations are shielded from outside intervention and unintentional alteration, avoiding unexpected outcomes. Because modifications to an object's internal structure can be changed without impacting the exterior code that interacts with the object, it also makes code maintenance easier.

To prevent direct manipulation of the balance attribute from outside the class, encapsulation could be utilized in the case of a **BankAccount** class. This preserves the integrity of the account data by guaranteeing that operations such as deposits and withdrawals can only be carried out using methods that the class provides.

2.3. Abstraction

OOP abstraction makes complicated systems simpler by displaying only the pertinent functionality and concealing extraneous features. Instead of emphasizing how an object does something, it concentrates on what it does. This is accomplished through the use of abstract classes or interfaces, in which subclasses implement particular details.

For instance, the abstract method **calculateInterest()** in the **BankAccount** class might be implemented differently in **SavingsAccount** and **FixedDepositAccount**. By permitting implementation modifications without impacting the object's functionality, abstraction preserves flexibility and facilitates the design and upkeep of systems.

2.4. Inheritance

In object-oriented programming, inheritance enables a class—referred to as the child or subclass—to inherit the properties and functions of another class—referred to as the parent or superclass. This encourages efficiency and minimizes duplication by allowing the child class to reuse the code defined in the parent class. By altering inherited methods and attributes or adding new ones, a child class can also increase the functionality of the parent class.

For instance, **a BankAccount** class in a banking system might define methods like **deposit()** and **withdraw()** in addition to standard properties like **accountNumber** and amount. You might make specialized subclasses from this class that inherit these common behaviors, such as **SavingsAccount** or **CurrentAccount**. Nonetheless, distinct features can be added by each subclass; for example, SavingsAccount might have an **interestRate** attribute and an interest calculation method.

Inheritance enables a hierarchical design in addition to code reuse. It makes systems more adaptable, structured, and manageable by enabling the development of specialized classes that retain the parent class's common functionality.

2.5. Polymorphism

A key idea in object-oriented programming (OOP) is polymorphism, which permits objects of multiple classes to be regarded as belonging to the same superclass. The phrase "polymorphism" refers to the ability of various objects to assume several forms or behaviors while sharing a common interface. It is derived from the Greek terms "poly," which means many, and "morph," which means shapes.

The two primary methods for achieving polymorphism in OOP are method overloading and overriding.

Method Overriding

This happens when a subclass offers a method that is already defined in its parent class with a customized implementation. The subclass can modify the behaviour to suit its requirements since its method "overrides" the parent class's method. **SavingsAccount** and **CurrentAccount**, for instance, can each override the **calculateInterest()** function with unique implementations tailored to their respective account types if they both inherit from **BankAccount**. The user can still interact with the **BankAccount** interface.

Method Overloading

This makes it possible for several methods to coexist in the same class with the same name but differing parameter kinds or numbers. This allows for technique flexibility depending on various input conditions.

The dynamic binding (also known as late binding) principle, which determines the precise method to be called at runtime rather than compile time, is made possible by polymorphism. For example, the system detects at runtime if a **BankAccount** object is a **SavingsAccount** or **CurrentAccount** and runs the appropriate version of **calculateInterest()** when you call it on the object. More adaptable and scalable code is made possible as a result.

Polymorphism's primary benefit is that it encourages adaptability and extension. Systems may expand and alter without requiring significant rewrites because to the ability for developers to build universal code that can deal with a variety of object kinds. For instance, new account types could be added to a banking system without changing the main code that communicates with **BankAccount** objects.

2.6. Composition

In object-oriented programming, composition is the process of merging smaller, more specialized objects to create larger, more complicated objects by establishing a "has-a" relationship. Composition focuses on creating an entity by assembling other objects as its components, as opposed to inheritance, which is the process by which a subclass extends a parent class.

A **BankAccount** class, for instance, would consist of client and Transaction objects, each of which would be responsible for particular functions such as managing transactions and client

data. Composition increases reusability because these items can be used in multiple modules and encourages loose coupling, which allows components to be modified without affecting other system components. Compared to inheritance, this method makes designs more adaptable and simpler to maintain.

3. Unified Modeling Language (UML) in OOSD

As a standardized method for visualizing a system's architecture and design, the Unified Modeling Language (UML) is essential to Object-Oriented Software Design (OOSD). Developers, architects, and stakeholders can better convey the structure, behavior, and relationships of objects in an object-oriented system by using the collection of diagrams and symbols that UML offers.

Before real coding starts, UML helps define and record how the system will work during the design phase, which is one of the many stages of the software development lifecycle. Its diagrams make complex designs easier to comprehend by providing a clear, visual representation of a system's dynamic (behavioral) and static (structural) components.

The main categories of UML diagrams utilized in OOSD are as follows:

1. Diagram of the class:
   A class diagram, which is the most often used diagram in OOSD, illustrates the system's structure by displaying its classes together with their properties, methods, and connections. It highlights important object-oriented ideas like inheritance, composition, and connections while graphically defining the relationships between classes.

   A class diagram might, for instance, show the relationships between classes like BankAccount, Customer, and Transaction in a financial system. This aids developers in comprehending the general architecture of the system and the interactions between its many parts.

2. Diagram of the Object:

Class instances (objects) at a certain point in time are depicted in object diagrams. They are helpful for illustrating how classes and their relationships are instantiated in the actual world. Object diagrams depict particular instances of the abstract structure, whereas class diagrams display the abstract structure.

3. Diagram of the Use Case:
   The interactions between users (actors) and the system are modeled by a use case diagram. It draws attention to the different features the system provides and the people that utilize them. Use case diagrams could illustrate how users interact with a financial system to carry out operations like checking their balance or making withdrawals.

4. Diagram of Sequence:
   Sequence diagrams illustrate the order of messages sent back and forth between objects, modeling the flow of interactions throughout time. This is helpful for comprehending a system's dynamics, including the sequence in which methods in various objects are invoked. A sequence diagram could illustrate how the Customer, BankAccount, and Transaction objects interact during a deposit in a bank account system.

5. Diagram of Activities:
   The system's control flow is shown in an activity diagram. It shows the order of actions, choices, and concurrent procedures. These diagrams are helpful for simulating system workflows, like the procedures in a bank transaction.

6. Diagram of the State:
   State diagrams are used to show the various states that an item might be in and how events can cause those states to change. This might be used to display a BankAccount object's different states—such as active, frozen, or closed—as well as the actions that cause them to change between them in the context of OOSD.

   UML is essential for OOSD Visualization and Communication because it offers a visual language that facilitates effective and clear communication of complex system designs among stakeholders, including developers, project managers, and clients.

Standardization: Because UML is a standardized modeling language, developers may express designs using common symbols and notations, which facilitates team collaboration and understanding.

Problem-Solving and Design: By providing a thorough overview of the system prior to coding, UML diagrams help detect possible design issues early on, such as incorrect object associations or ambiguous system actions.

Documentation: UML is a type of documentation that explains the system's architecture and design. Because it offers a blueprint that can be consulted at any point during the project's lifecycle, this documentation is helpful for updating and maintaining the system.

4. Key Stages of OOSD

Gathering requirements, creating the system, and testing it are all important steps in the structured Object-Oriented Software Design (OOSD) process. The ultimate objective is to create an object-oriented software system that is scalable, maintainable, and functional. Each step builds upon the one before it.

The following are the main phases of OOSD:

1. Analysis of Requirements
   Goal: Comprehending and obtaining specific requirements from the stakeholders (clients, users, and business analysts) is the initial step. Finding the system's functionality, performance needs, and limitations is the main goal of this stage.

   Important Tasks
   Use Case Development: In order to capture the primary functionalities that the system must offer, use cases are developed to simulate the interactions between users (actors) and the system.
   Conditions Records: a detailed description of the system's capabilities, including both non-functional (performance, security) and functional (particular tasks) needs.

Output: A thorough requirements paper or use case model outlining the objectives and capabilities of the system.

2. System Design
   Goal: The high-level architecture and structure of the system are created at this stage. This entails specifying the elements of the system, their connections, and its general design.

   Important Tasks
   High-Level Design: Class diagrams are frequently used to illustrate the structure of the system and map out its design.
   Subsystem Identification: Using object-oriented concepts like abstraction and encapsulation, the system is separated into manageable subsystems or modules.
   Interface Design: Specifies how the system will communicate with users, databases, and other systems. There are identified interfaces between modules or classes.
   System design diagrams, containing component and UML class diagrams as well as a high-level design document, are the output.

3. Object Design
   Goal: In object design, the emphasis switches to precisely defining the objects, classes, and relationships. Every class has comprehensive specifications for its properties, methods, and interactions with other objects.

   Important Tasks
   Diagrams of classes: Each class's properties, methods, visibility (private, public), and relationships (inheritance, composition, etc.) are all specified in comprehensive UML class diagrams.
   Interactions with Objects: Sequence diagrams or cooperation diagrams model how items interact. These demonstrate the invocation of methods and the data flow between objects in different contexts.
   Design Patterns: To address persistent design issues and improve the structure, common object-oriented design patterns—such as Factory, Singleton, and Observer—are used.
   Output: Comprehensive design documents, class diagrams, and sequence diagrams that explain every object and how it interacts with the others.

4. Implementation (Coding)
   Goal: The design is now converted into code. The programming language used for the project—Java, C++, Python, or another—is used to implement the object-oriented design.

   Important Tasks
   Class Implementation: The design criteria and class diagrams serve as the basis for the coding of each class. The implementation complies with polymorphism, inheritance, and encapsulation.
   Modular Development: Code is written in modules that match the previously designed classes and subsystems. This guarantees the system's continued scalability and maintainability.
   Integration: To create a functional system and guarantee that object interactions function as intended, separate modules or classes are combined.
   Output: The system's source code, arranged into packages, classes, and modules.

5. Testing
   Goal: Guarantees system functionality and compliance with specifications. Unit testing, system testing, and other testing layers are commonly used to object-oriented systems.

   Important Tasks
   Unit Testing: To make sure that its methods and properties operate as intended, each class or object is tested separately. Unit tests confirm that objects operate correctly when left alone.
   Integration Testing: Following the testing of individual classes, integration testing verifies that the interactions between modules and objects are operating as intended.
   System Testing: To make sure the system satisfies the functional and non-functional requirements specified in the analysis phase, the complete system is tested collectively. This covers user acceptance testing (UAT), security testing, and performance testing.
   Debugging: After addressing any defects or problems found during testing, the system is retested to confirm the corrections.
   Output: A thoroughly tested, error-free system that is prepared for implementation.

Additionally offered are test results and documentation of any problems that were discovered.

6. Implementation and Upkeep

Goal: The system is put into the live environment following testing. Updates, bug fixes, and other enhancements are made to the system during the maintenance phase after it is put into use.

Important Tasks

System deployment is the distribution of the software to users and the completion of the required setup (e.g., database setup, server installation).
Instruction for Users: To properly use the system, end users receive training.
Continuous Maintenance: Following deployment, the system might need to be updated, fixed, and improved on a regular basis. This step deals with user feedback and system performance problems.
Output: A deployed, functional system with ongoing maintenance and enhancements as required.

5. Jazzman Bank Application

- *User Verification and Login*

Users are asked to enter their account number and PIN when they launch the application. Only authorized users can access their accounts since the system validates the information entered.
A welcome message with the user's complete name appears after successful verification, providing an extra degree of personal validation.

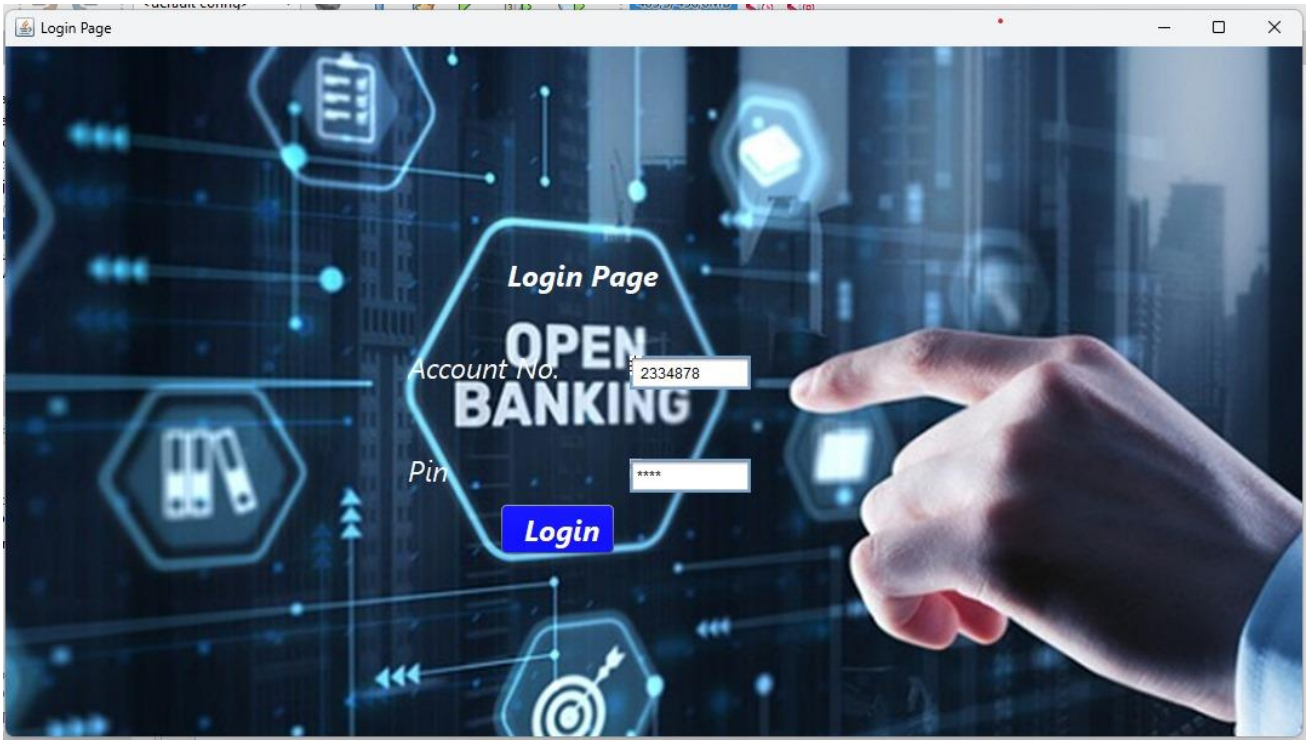

Figure 1: Login page for Jazzman Bank.

- *Features of Account Management*

Through a menu, the application offers customers a number of crucial banking features:

Users can check their current account balance with the **Balance Inquiry** feature. There is no cost for this service.
Deposit: Money can be deposited into an account by users. A fee of R15 is charged for each deposit transaction.
Withdrawal: Users are able to take money out of their accounts up to the available balance. The cost for each withdrawal is R10.
Transaction History: Recent transactions are available for users to examine. To access the transaction history, a nominal cost of R5 is charged for each entry.

PIN Change: For increased security, users have the option to modify their PIN. Before enabling the update to the new PIN, the system asks for the current PIN.

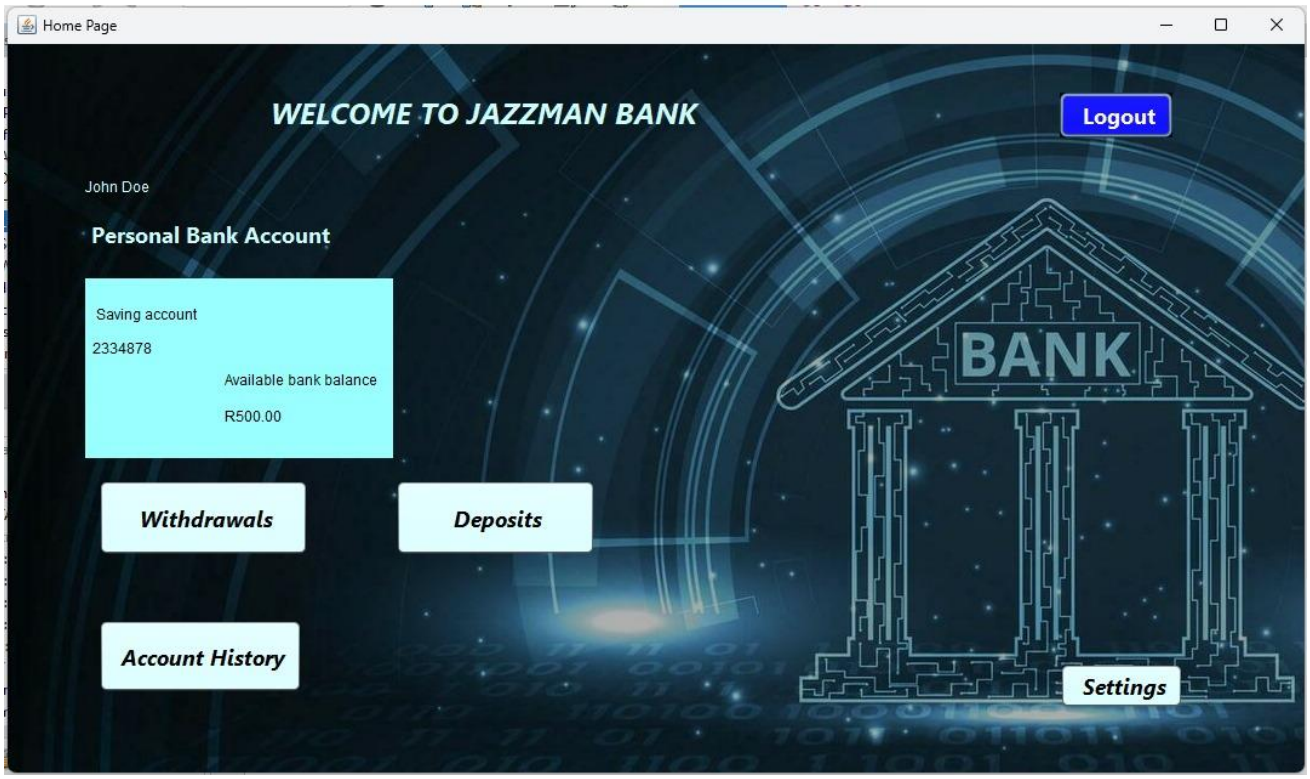

Figure 2: All features for Jazzman Bank.

- *Fees and Transaction Logging*

For future reference, every transaction is documented in the transaction history. Transparency is ensured via the application's notification system, which alerts users of any bank fees incurred following each transaction.
Fees for deposits, withdrawals, and transaction history viewing are all examples of bank costs. Inquiries about balances are free.

- *Configuration and Security Functionalities*

Users can examine their transaction history and change their account PIN through the "Settings" option.
Users must input their current PIN to verify their identification when changing their PIN for security reasons.

```
Settings Menu:
1. Change PIN
2. View Transaction History (R5 per entry)
1
Enter current PIN: 2204
Enter new PIN: 2011
PIN updated successfully.
```

Figure 3: Compiled demo of a successful change of pin.

To guarantee a seamless user experience, each transaction prompts the customer to choose depart or return to the main menu.
Users have the option to either exit the application or go back to the main menu to carry out other tasks after finishing each transaction.

- *Storage and Data Structure*

While the **BankAccount** class manages the account balance and transaction activities, User objects hold user data in memory, including name, surname, account number, PIN, and transaction history.
In order to facilitate effective access and maintenance of account data, the system employs a map to store and retrieve users according to their account number.

```java
package bankaccount;
import bankaccount.User;
import java.util.HashMap;
import java.util.Map;
import java.util.Scanner;

public class BankSystem {
    private Map<String, User> users = new HashMap<>();

    public BankSystem() {
        // Bank Users
        users.put(key:"1234567", new User(name:"John", surname:"Doe", accountNumber: "2334878", pin:"2204", initialBalance: 500.0));
        users.put(key:"2345678", new User(name:"Jane", surname:"Smith", accountNumber: "2345678", pin:"2345", initialBalance: 1000.0));
        users.put(key:"3456789", new User(name:"Mike", surname:"Brown", accountNumber: "3456789", pin:"3456", initialBalance: 750.0));
        users.put(key:"4567890", new User(name:"Alice", surname:"Green", accountNumber: "4567890", pin:"4567", initialBalance: 300.0));
        users.put(key:"5678901", new User(name:"Bob", surname:"White", accountNumber: "5678901", pin:"5678", initialBalance: 1200.0));
    }
```

Figure 4: Class that manages user information and used Maps.

6. Flowchart of the Bank

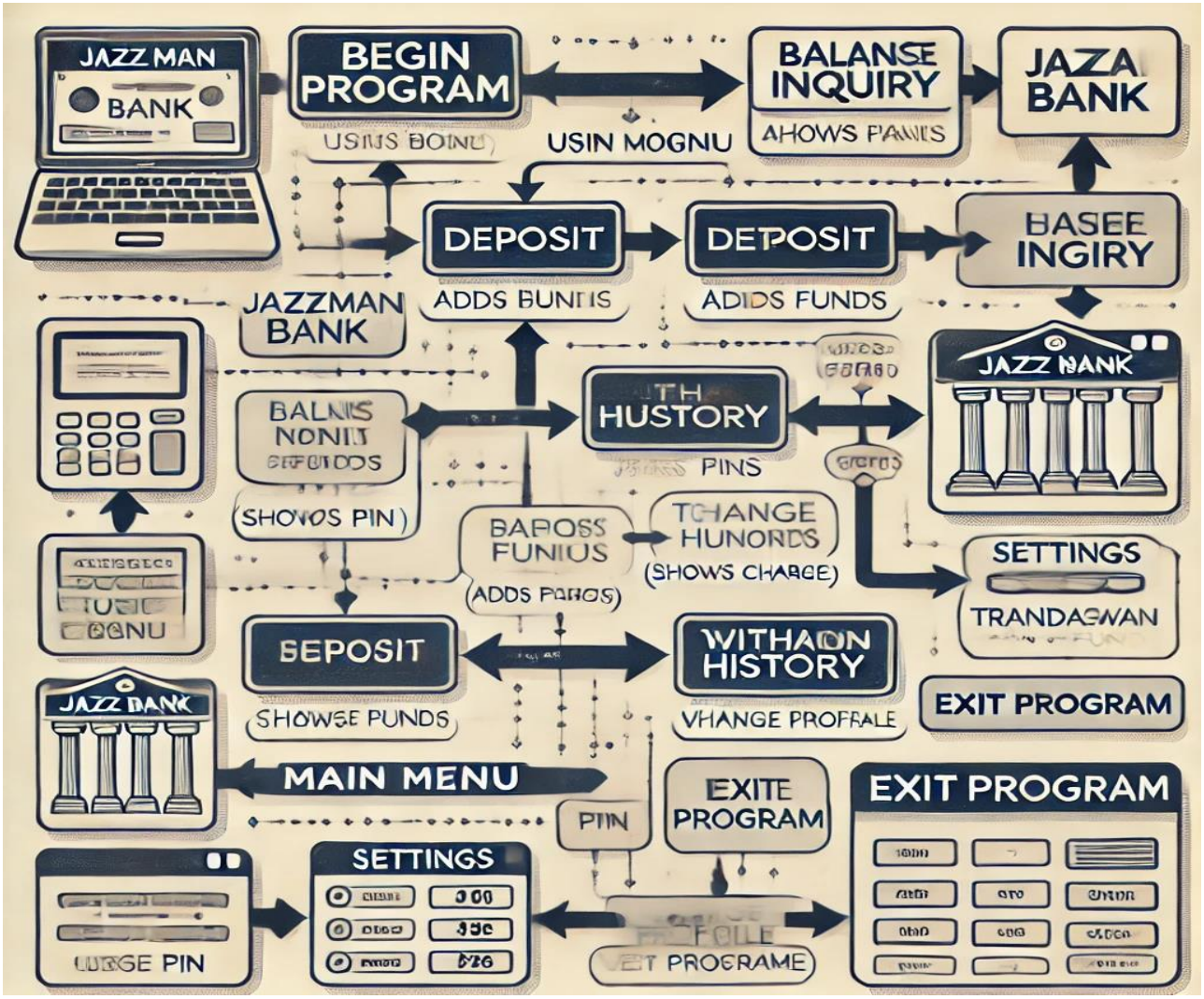

Figure 5: Jazzman Bank Flowchart.

7. Conclusion

The project "Jazzman Bank" effectively illustrates how to use object-oriented software design to produce an interactive, useful financial system. The application offers a realistic simulation of a banking environment by incorporating elements like transaction history, bank charges, and user authentication. The project demonstrates how object-oriented principles help make the code more modular, reusable, and maintainable, which facilitates feature additions and system modifications as needed. All things considered, this project is a good illustration of how object-oriented approaches may be used to address real-world issues and create useful software applications. We have learned how to manage user data securely, improve the user experience, and build and implement complicated systems as a result of this experience.

8. Reference

[1] P. Dietel and H. D. Dietel, Java How to program Early Objects, 9 ed., Cape Town: Prentice Hall, 2012.