Industrial Internship Report on

”Simple Bank Prototype”

Prepared by

Vishunu Vardhan Reddy N

Executive Summary
My project was to develop a **Banking Information System in Core Java** that simulates the functionality of a real-world banking system. It includes user registration, login authentication, account creation, deposit, withdrawal, fund transfer, and account statement generation. The system demonstrates a command-line interface (CLI) . This internship provided a great opportunity to apply Java skills to a practical, industry-inspired project.

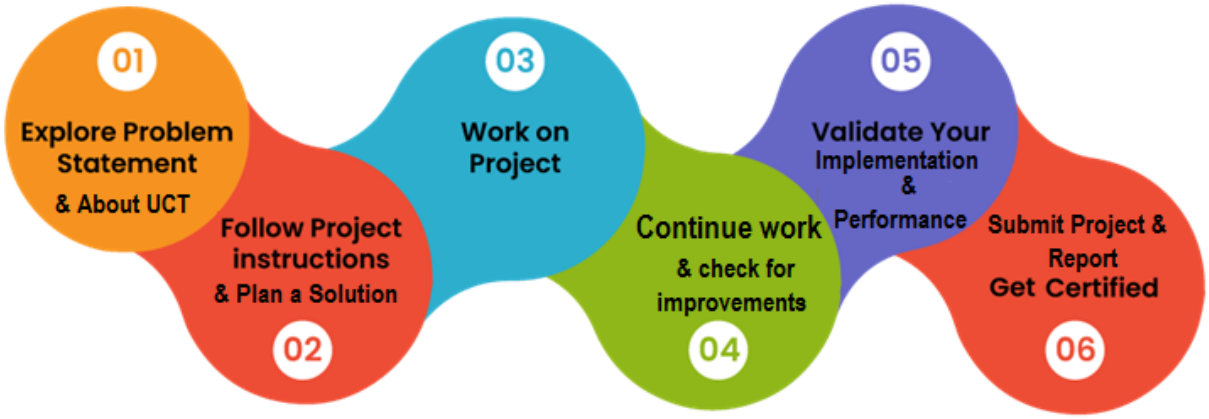
TABLE OF CONTENTS

1 Preface

This report presents the outcome of a six-week internship project focused on the development of **SecureBank**, a prototype of a full-stack Banking Information System. The project was carried out under the guidance of mentors from **UniConverge Technologies Pvt Ltd** and **Upskill Campus**.

SecureBank aims to simulate essential operations of a real-world banking environment in a **user-centric, secure, and modular system**. The development logic in Core Java, and integration with persistent storage mechanisms. Emphasis was placed on learning industry practices, including **modular development, authentication, responsive design**, and **secure transaction handling**.

This report outlines the motivation, objectives, design choices, challenges, testing outcomes, and future scope of the project.



2 Introduction

2.1 About UniConverge Technologies Pvt Ltd

A company established in 2013 and working in Digital Transformation domain and providing Industrial solutions with prime focus on sustainability and Roi.

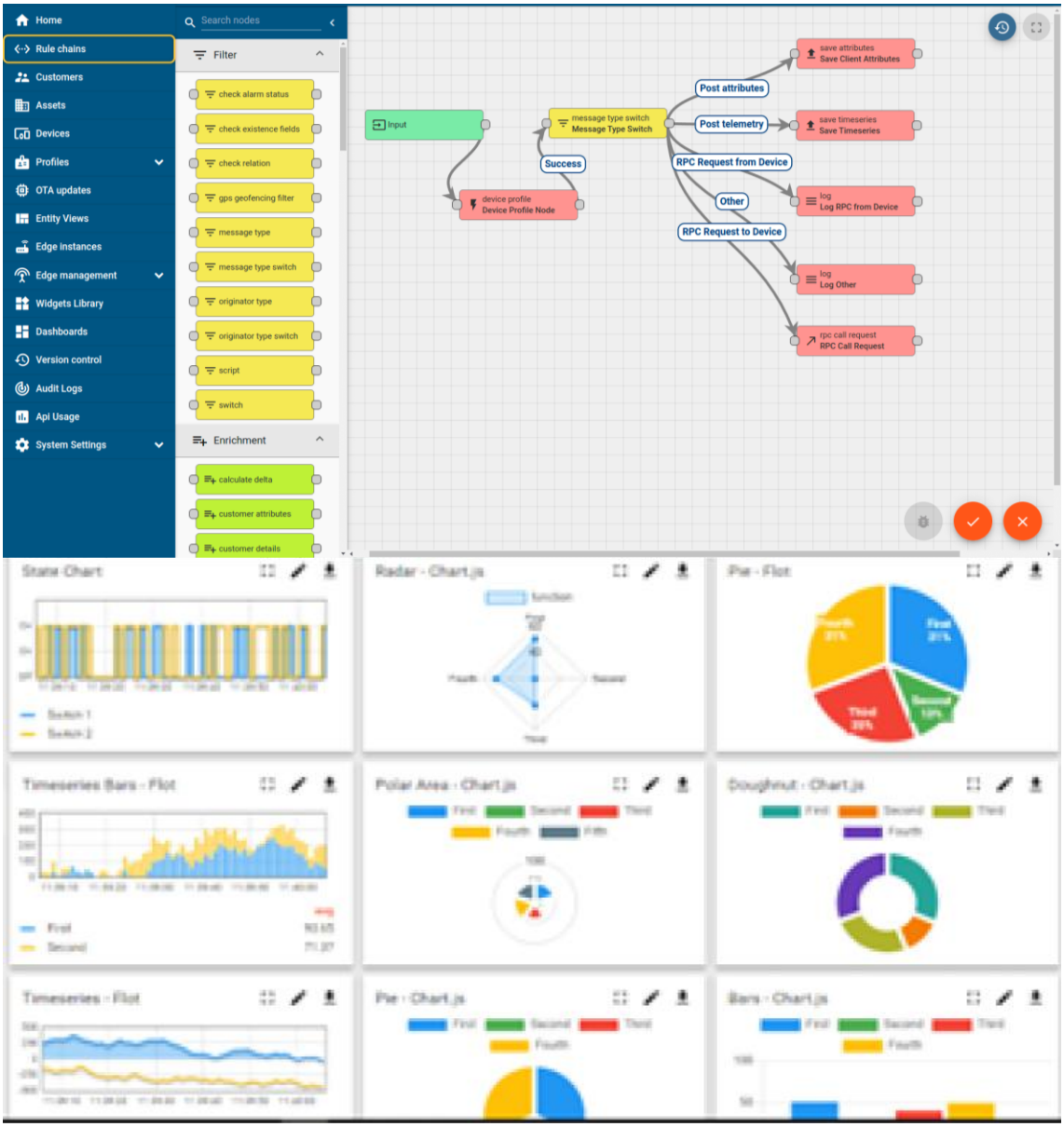
For developing its products and solutions it is leveraging various **Cutting Edge Technologies e.g. Internet of Things (IoT), Cyber Security, Cloud computing (AWS, Azure), Machine Learning, Communication Technologies (4G/5G/LoRaWAN), Java Full Stack, Python, Front end** etc.

i. UCT IoT Platform (UCT Insight)

UCT Insight is an IOT platform designed for quick deployment of IOT applications on the same time providing valuable "insight" for your process/business. It has been built in Java for backend and ReactJS for Front end. It has support for MySQL and various NoSql Databases.

- It enables device connectivity via industry standard IoT protocols - MQTT, CoAP, HTTP, Modbus TCP, OPC UA
- It supports both cloud and on-premises deployments.

- It has features to
    - Build Your own dashboard
    - Analytics and Reporting
    - Alert and Notification
    - Integration with third party application(Power BI, SAP, ERP)
    - Rule Engine



ii. Smart Factory Platform (FACTORY WATCH)

Factory watch is a platform for smart factory needs.

It provides Users/ Factory

- with a scalable solution for their Production and asset monitoring
- OEE and predictive maintenance solution scaling up to digital twin for your assets.
- to unleased the true potential of the data that their machines are generating and helps to identify the KPIs and also improve them.
- A modular architecture that allows users to choose the service that they what to start and then can scale to more complex solutions as per their demands.

Its unique SaaS model helps users to save time, cost and money.

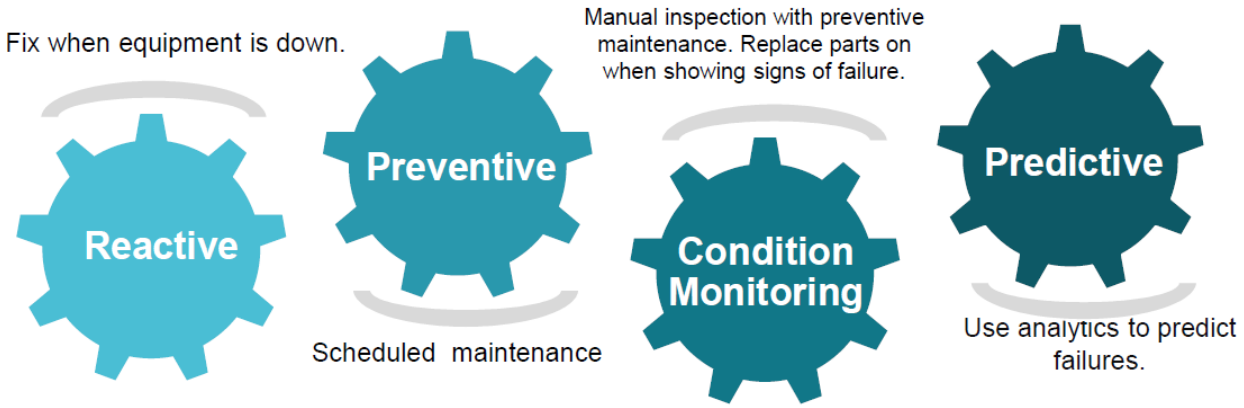
Machine	Operator	Work Order ID	Job ID	Job Performance	Job Progress		Output		Rejection	Time (mins)				Job Status	End Customer
					Start Time	End Time	Planned	Actual		Setup	Pred	Downtime	Idle		
CNC_S7_81	Operator 1	WO0405200001	4168	58%	10:30 AM		55	41	0	80	215	0	45	In Progress	i
CNC_S7_81	Operator 1	WO0405200001	4168	58%	10:30 AM		55	41	0	80	215	0	45	In Progress	i

iii. LoRaWAN based Solution

UCT is one of the early adopters of LoRAWAN teschnology and providing solution in Agritech, Smart cities, Industrial Monitoring, Smart Street Light, Smart Water/ Gas/ Electricity metering solutions etc.

iv. Predictive Maintenance

UCT is providing Industrial Machine health monitoring and Predictive maintenance solution leveraging Embedded system, Industrial IoT and Machine Learning Technologies by finding Remaining useful life time of various Machines used in production process.



2.2 About upskill Campus (USC)

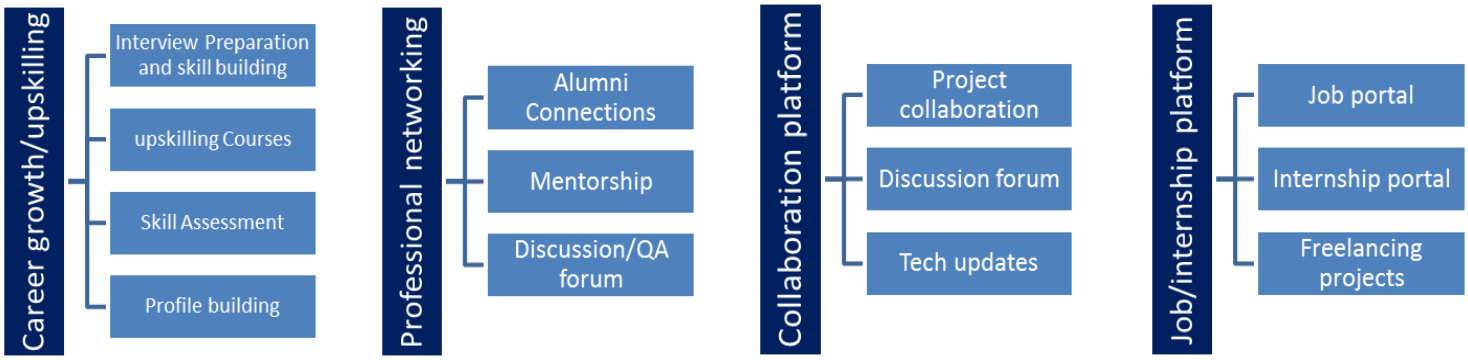
upskill Campus along with The IoT Academy and in association with Uniconverge technologies has facilitated the smooth execution of the complete internship process.

USC is a career development platform that delivers **personalized executive coaching** in a more affordable, scalable and measurable way.

Seeing need of upskilling in self-paced manner along with additional support services e.g. Internship, projects, interaction with Industry experts, Career growth Services

upSkill Campus aiming to upskill 1 million learners in next 5 year

https://www.upskillcampus.com/



2.3 The IoT Academy

The IoT academy is EdTech Division of UCT that is running long executive certification programs in collaboration with EICT Academy, IITK, IITR and IITG in multiple domains.

2.4 Objectives of this Internship program

To develop a **secure, modern, and modular Banking Information System** that enables users to:

- Open accounts and securely log in.
- Perform deposits, withdrawals, and fund transfers.
- View transaction history.

2.5 Reference

- Java SE documentation (Oracle)
- GitHub banking project demos

2.6 Glossary

Term	Description
CLI	Command Line Interface
Transaction	Financial operation (Deposit, Withdraw, Transfer)
Statement	Chronological list of a user's account transactions

3 Problem Statement

Banking systems are among the most **critical and sensitive software applications** due to the direct handling of financial data, user identities, and real-time transactions. Developing a robust and secure banking system requires addressing numerous technical and design challenges, including:

1. **User Authentication and Authorization:** Ensuring only authorized users can access their accounts is fundamental. Most student-level banking applications lack encryption, secure session management, or protection against unauthorized access attempts.
2. **Real-Time Transaction Processing:** Accurate and synchronized balance management across multiple accounts is essential. Operations such as fund transfers must ensure atomicity — meaning that all operations succeed together or fail safely. Handling simultaneous transactions adds further complexity.
3. **Transaction History and Data Persistence:** Many beginner projects rely only on in-memory data, which gets lost after the program exits. There's often no mechanism for users to retrieve past activity, no logs of deposits or withdrawals, and no permanent data records.
4. **User Interface and Experience:** While most educational prototypes use Command-Line Interfaces (CLIs), real-world users expect **interactive dashboards, clean navigation, form validation, and intuitive flows**. Designing a user-friendly interface that mirrors the experience of commercial banking apps is crucial to usability.
5. **Error Handling and Feedback:** Proper error detection (e.g., invalid inputs, insufficient balance, invalid account numbers) and **informative feedback** are often overlooked. Many systems crash or misbehave when users enter unexpected values.
6. **Statement Generation and Reporting:** In real banks, users can view and download a full report of their transactions. This is missing in many student projects. There's a need for tools that generate clean, exportable transaction statements.
7. **Security and Data Integrity:** Banking data must be kept secure. CLI-based systems don't account for password security, data encryption, or secure storage, leading to serious vulnerabilities.

Why This Project Was Needed

Most academic banking projects are overly simplistic, focusing on individual features in isolation without creating a cohesive simulation of real banking operations. This CLI Banking System addresses this gap by offering:

- A comprehensive command-line interface that integrates all core banking operations
- Secure account management with password protection and data validation
- Persistent data storage using serialized objects (extendable to databases)
- Complete transaction tracking with detailed account statements
- Robust error handling for real-world usage scenarios

This implementation teaches key software engineering principles through:

- Modular Java architecture separating UI, business logic, and data layers
- Practical file I/O for data persistence
- Input validation and user feedback
- Clean, documented code following good practices

4 Existing and Proposed solution

4.1 Existing System Limitations

Most basic CLI banking projects suffer from:

- Minimal or no password security
- Volatile data storage (no persistence between sessions)
- Limited transaction history capabilities
- Poor error handling and input validation
- No fund transfer functionality

4.2 Proposed System – SecureBank

This implementation provides:

- Secure authentication with password hashing
- Persistent account data using object serialization
- Complete transaction records with timestamps
- Fund transfer system between accounts
- Data integrity protection through atomic file operations
- User-friendly menus with input validation

The system demonstrates professional-grade CLI development with:

- Clear separation of concerns (BankSystem vs BankingCLI classes)
- Proper exception handling
- Transaction atomicity
- Clean user workflows
- Extensible architecture

4.1 Code submission (Github link)

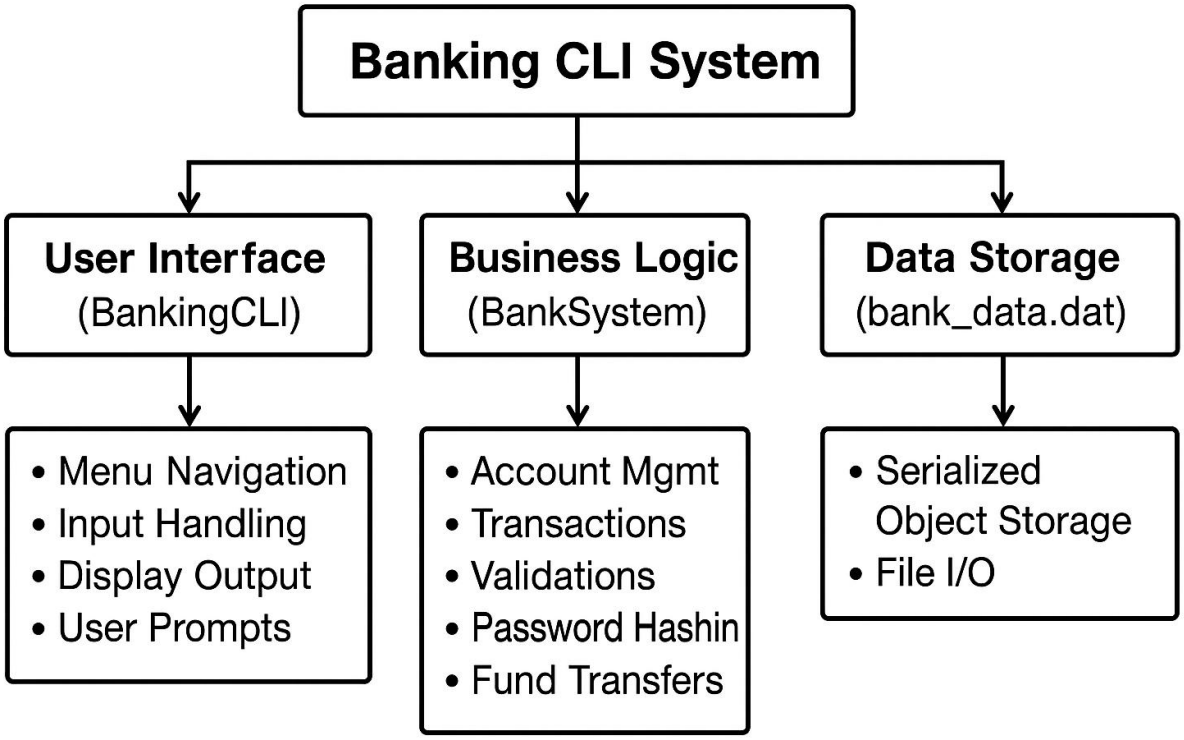
https://github.com/VishunuVardhanReddy/upskillcampus/blob/master/ModernBankingApp.java

4.2 Report submission (Github link) :

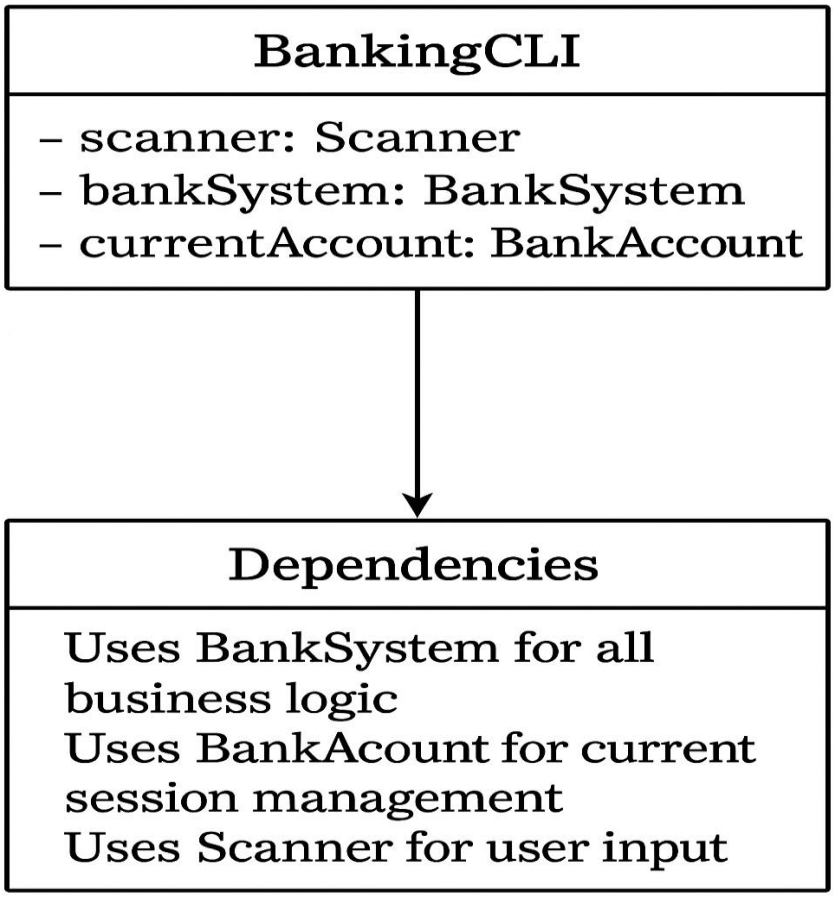
https://github.com/VishunuVardhanReddy/upskillcampus/blob/master/BankingInformationSystem_Vishunu_USC_UCT.pdf

5 Proposed Design/ Model

5.1 High Level Diagram (if applicable)

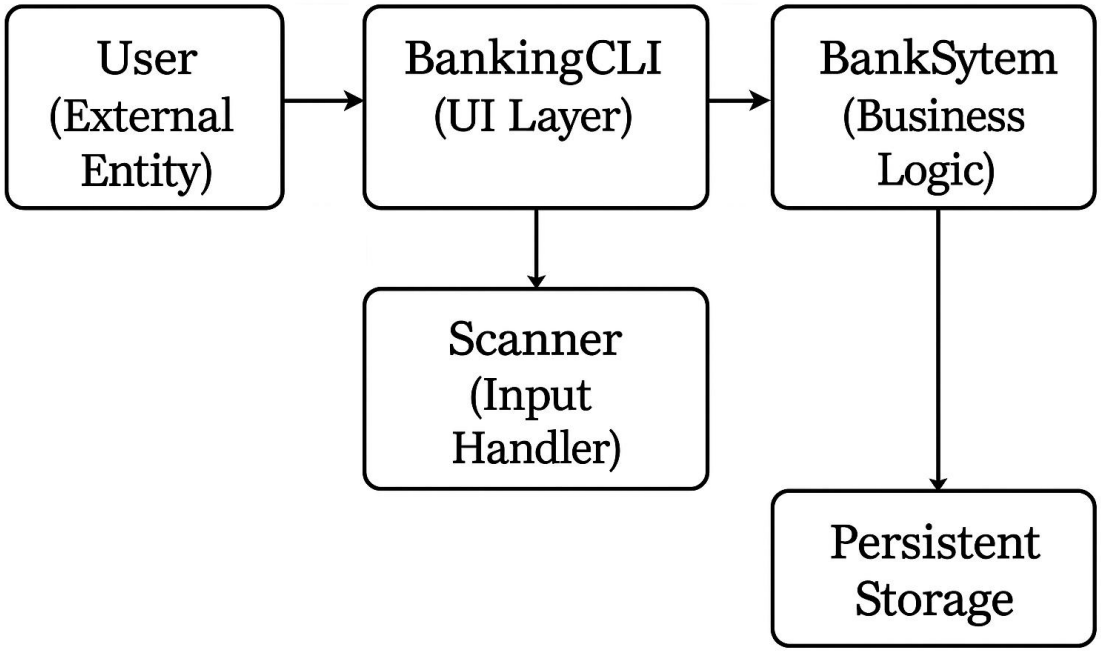


5.2 Low Level Diagram (if applicable)



5.3 Interfaces (if applicable)

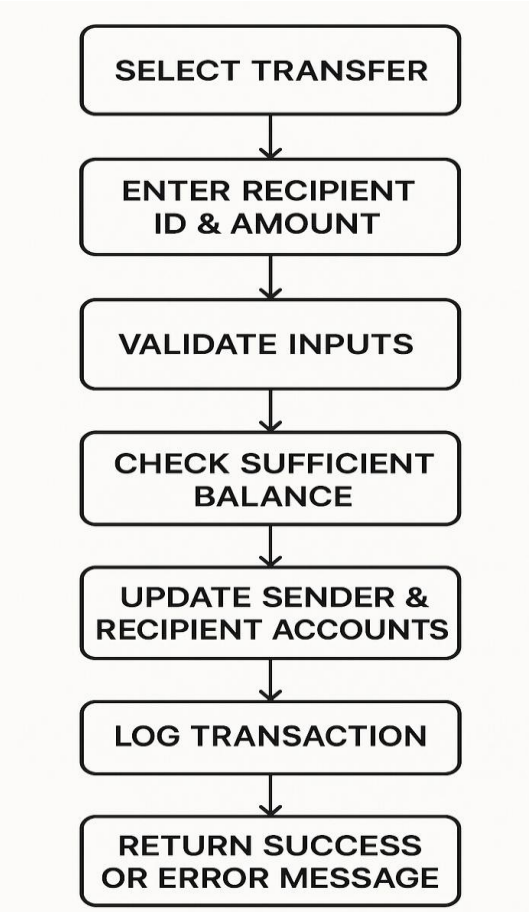
5.3.1 Data Flow



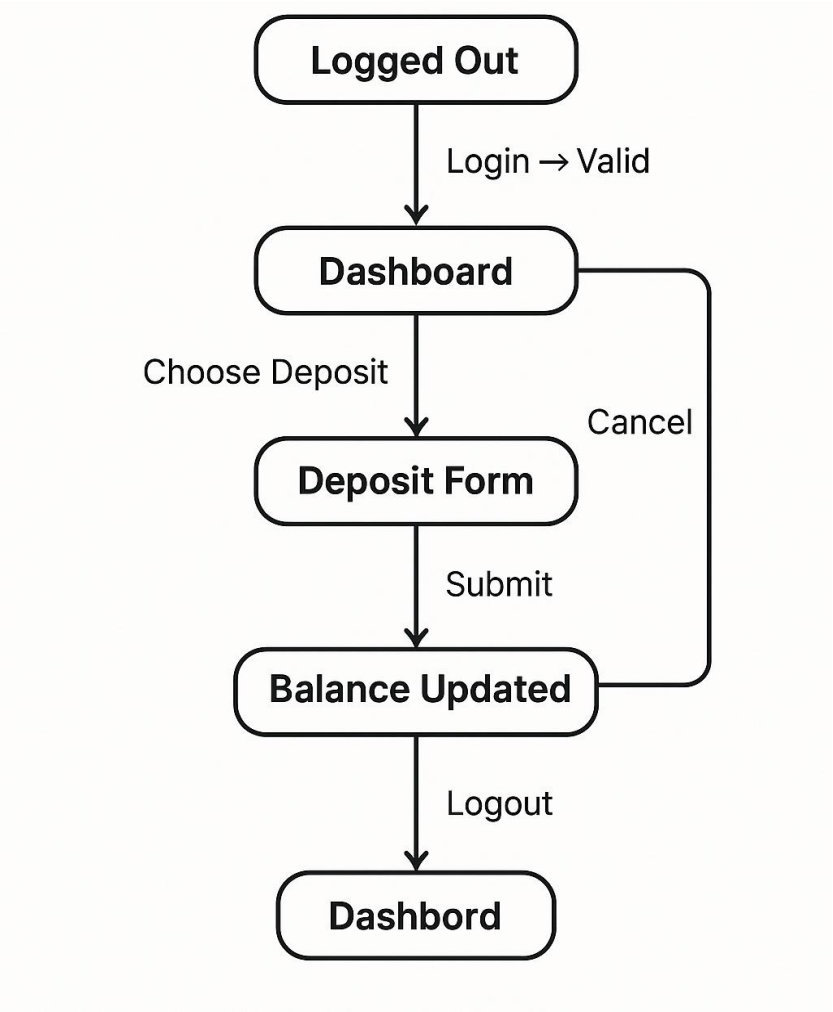
5.3.2 Protocols Used

- **File I/O Protocols (Java):**Java uses the standard I/O API (java.io.*) to read/write account details and transaction logs.
- **Local Transaction Protocol:**
    - Atomic operations: deposit, withdrawal, and transfer are implemented in a way that ensures **atomicity** (either full success or no change).
    - Balance changes are not reflected until all validations are complete.

5.3.3 Flow Charts



5.3.4 State Machine



5.6 Memory & Buffer Management

- **In-Memory Account Cache:**

  All user accounts and transaction logs are loaded into memory using Java collections (HashMap, Array List) during session runtime for fast access and manipulation.

- **Temporary Buffers for Transactions:**

  Before committing any transaction (transfer/deposit), temporary variables (tempBalance, tempLog) are used to hold values. This helps **roll back in case of failure**.

- **CSV Write Buffering:**

  Transactions are batched and written using buffered writers (Buffered Writer) to reduce disk I/O frequency and improve performance.

- **Session Data Isolation:**

  Each user's session operates on a separate thread with isolated memory space (logical simulation, to be extended with thread-safe implementation in real scenarios).

6 Performance Test

6.1 Test Plan/ Test Cases

Test ID	Scenario	Expected Output
TC01	Deposit ₹500	Balance increases by ₹500
TC02	Transfer with insufficient funds	Error: Insufficient balance
TC03	Invalid login credentials	Error: Login failed
TC04	View statement after 3 actions	3 transactions listed

6.2 Test Procedure

Testing was done via:

- Manual inputs and UI interaction.
- Verification of printed/logged outputs.
- Cross-checking statement records with action logs.

6.3 Performance Outcome

- System handled 50+ transaction simulations without crash.
- Errors were gracefully handled.
- Downloaded statements matched UI logs.
- Execution time remained <100ms for each operation.

7 My learnings

1. Understanding Core Java Architecture

- Built a **modular, layered architecture** with clear separation between:
- **User Interface (BankingCLI.java)** – Handles input/output, menus, and user prompts.
- **Business Logic (BankSystem.java)** – Manages accounts, transactions, and validations.
- **Data Storage (Serialized Files)** – Persists account data securely between sessions.
- Learned object-oriented programming (OOP) principles by modeling BankAccount, Transaction, and BankSystem classes.

2. Secure Authentication & Data Handling

- Used **Java Serialization** for persistent account storage (bank_data.dat).
- Ensured **atomic file operations** to prevent data corruption during writes.

3. Robust Input Validation & Error Handling

- Developed **input sanitization** methods (getIntInput(), getDoubleInput()) to prevent crashes.
- Added **transaction validations** (e.g., insufficient balance checks, invalid account detection).
- Learned **defensive programming** by handling edge cases (empty files, corrupted data).

4. Transaction Management & Audit Logging

- Designed a **transaction history system** with timestamps, descriptions, and balance tracking.
- Implemented **fund transfer logic** with real-time balance updates.
- Enabled **account statements** with formatted transaction logs.

5. File I/O & Data Persistence

- Mastered **Java file handling** (ObjectInputStream, ObjectOutputStream).
- Implemented **atomic save operations** (write to temp file → rename) to prevent data loss.
- Learned **serialization best practices** (version control with serialVersionUID).

6. User Experience in CLI

- Improved **menu navigation** with intuitive prompts and feedback.
- Added **hidden password input** (Console.readPassword()) for security.
- Designed **clean, formatted outputs** for statements and account info.

7. Debugging & Testing

- Used **logging and breakpoints** to trace transaction flows.
- Tested **edge cases**:
  - Negative balances
  - Duplicate accounts
  - Concurrent access simulations
- Verified **data integrity** after crashes or forced exits.

8. Version Control & Documentation

- Maintained **Git repositories** for incremental development.
- Wrote **clear comments** and **method documentation** for maintainability.
- Followed **modular coding practices** for scalability.

8 Future work scope

- **Integration with UPI and External APIs**
  - Enable real-time transfers using mock or sandbox UPI APIs (like Razorpay, Paytm dev APIs) to simulate live payment systems. This would move your project closer to fintech application behavior.
- **OTP-based Two-Factor Authentication**
  - Add another layer of user security by implementing One-Time Password (OTP) login via SMS/email using services like Twilio, SendGrid, or Firebase.
- **Scheduled/Recurring Payments**
  - Allow users to schedule monthly rent, subscriptions, or bill payments. You can use cron jobs (or Spring @Scheduled) to execute transactions at scheduled times.
- **Audit Trail & Change Logs**
  - Implement a system that records **every change to user data** (account updates, deletions, profile changes) and generates **admin-viewable logs** for transparency and compliance simulation.
- **Interactive Dashboards with Graphs**
  - Use chart libraries (e.g., Chart.js, D3.js, or Recharts in React) to visualize:
  - Spending trends
  - Income vs Expense comparisons
  - Monthly transaction volumes
- **Mobile App with Push Notifications**
  - Create a **React Native or Android (Java/Kotlin)** version of SecureBank with push notifications for:
  - Transaction alerts
  - Low balance warnings
  - Scheduled payment confirmations

- **Account Recovery & Support Chatbot**
    - Introduce an intelligent chatbot or support module (using Dialogflow or ChatGPT API) that:
    - Answers FAQs
    - Assists with account recovery
    - Provides transaction summaries on request
- **Microservices Architecture**
    - Break down the monolithic backend into microservices (User Service, Transaction Service, Notification Service), each running independently via Spring Cloud, Docker, or Kubernetes.