Project Report

CSE338

Software Testing,Validation,and Verification Project

By

Abdelrahman Barakat (21P0012, u2498523)
Maeen Mohamed (2101544, u2498566)
Tsneam Ahmed (21P0284, u2498621)
Fatma Ayman(2100327, u2679055)
Omar Alaa (21P0197, u2498597)

Submitted To: Dr Yasmine Afify
TA: Eng Mahmoud Wageeh
TA: Eng Amr Sayed

Contents

1

Introduction

1.1. Banking Systems Overview:

A banking system manages financial transactions, customer accounts, loans, and related services. It includes features like account management, fund transfers, loan processing, and online banking. Testing encompasses security measures, transaction accuracy, data integrity, and compliance with financial regulations.We divided our code into classes:

1.1.1. Bank Class:

```java
public class Bank {
    public static ArrayList<Account> accounts = new ArrayList<>();
    public static ArrayList<Transaction> transactions = new ArrayList<>();
    public static ArrayList<Loan> loans = new ArrayList<>();
    public void addAccount(Account account) {
        accounts.add(account);
    }

    public static Account getAccount(String accountId) {
        for (Account a:accounts){
            if (a.getAccountId().equals(accountId)){
                return a;
            }
        }
        return null;
    }

    public Transaction getTransaction(String transactionID) {
        for (Transaction t:transactions){
            if (t.getTransactionId().equals(transactionID)){
                return t;
            }
        }
        return null;
    }
}
```

Figure 1.1: Bank Class

- The Bank class represents a banking system. It contains three ArrayList fields: accounts, transactions, and loans.
  Methods:

- The addAccount method adds an account to the bank.
- The getAccount method retrieves an account based on the account ID.
- The getTransaction method retrieves a transaction based on the transaction ID.

1.1.2. Account Class:

```
class Account {
    private String accountId;
    private String accountOwner;
    private double balance;
    private String password;
    protected ArrayList<Transaction> transactionList = new ArrayList<>();
    protected ArrayList<Loan> takenLoans = new ArrayList<>();

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

    public String processTransaction(Account toAccount,double amount, String TransactionType) {
        if (this.getBalance() ⩾ amount && this ≠ toAccount) {
            if (amount > 0) {
                balance -= amount;
                toAccount.balance += amount;
                Bank.transactions.add(new Transaction(this, toAccount, amount, new SimpleDateFormat("dd/MM/yyyy HH:mm:ss").format(new Date()), TransactionType));
            } else {
                return "Amount Value Must be More Than 0";
            }
        } else {
            return "Couldn't Complete the Transfer. Insufficient Balance";
        }
        return null;
    }

    public String processTransaction(double amount, String transactionType) {
        if (transactionType.equals("D")){
            if (amount > 0) {
                balance += amount;
                Bank.transactions.add(new Transaction(this, amount, new SimpleDateFormat("dd/MM/yyyy HH:mm:ss").format(new Date()), transactionType));
            } else {
                return "Amount Value Must be More Than 0";
            }
        }
        else if (transactionType.equals("DL")){
            if (amount > 0) {
                balance += amount;
                Bank.transactions.add(new Transaction(this, amount, new SimpleDateFormat("dd/MM/yyyy HH:mm:ss").format(new Date()), transactionType));
            } else {
                return "Amount Value Must be More Than 0";
            }
        } else if (transactionType.equals("PL")){
            if (amount ⩽ balance) {
                balance -= amount;
                Bank.transactions.add(new Transaction(this, amount, new SimpleDateFormat("dd/MM/yyyy HH:mm:ss").format(new Date()), transactionType));
            } else {
                return "Couldn't Pay Loan. Insufficient Balance";
            }
        } else{
            if (amount ⩽ balance) {
                balance -= amount;
                Bank.transactions.add(new Transaction(this, amount, new SimpleDateFormat("dd/MM/yyyy HH:mm:ss").format(new Date()), transactionType));
            } else {
                return "Couldn't Complete the Withdraw. Insufficient Balance";
            }
        }
        return null;
    }
```

Figure 1.2: Account Class Part 1

```
    public String takeLoan(String loanId, double loanAmount) {
            for (Loan l : Bank.loans) {
                if (l.getLoanId().equals(loanId)) {
                    l.setLoanAccount(this);
                    l.setLoanAmount(loanAmount);
                    String msg = l.disburseLoan();
                    if (msg == null)
                        takenLoans.add(l);
                    return msg;
                }
            }
            return "Loan Not Found"; // unreachable
        }

        public double getBalance() {
            return balance;
        }

        public String getAccountId() {
            return accountId;
        }

        public String getAccountOwner() {
            return accountOwner;
        }

        public void setBalance(double balance) {
            this.balance=balance;
        }

        public void setAccountId(String accountId) {
            this.accountId=accountId;
        }

        public void setAccountOwner(String accountOwner) {
            this.accountOwner=accountOwner;
        }
        public void setPassword(String password){
            this.password=password;
        }
        public String getPassword(){
            return password;
        }
        public ArrayList<Transaction> getTransactionList() {
            return transactionList;
        }
        public ArrayList<Loan> getTakenLoans() {
            return takenLoans;
        }
    }
```

Figure 1.3: Account Class Part 2

- The Account class represents an individual bank account.

Attributes

- **accountId**: A unique identifier for the account.
- **accountOwner**: The name of the account holder.
- **balance**: The current monetary balance of the account.
- **password**: A password to secure access to the account.
- **transactionList**: A dynamic list that tracks all transactions associated with this account.
- **takenLoans**: A list of all loans taken out by the account holder.

Methods

- **Constructor**: Initializes a new account with an ID, owner's name, initial balance, and password.
- **processTransaction(Account toAccount, double amount, String transactionType)**: Processes a transaction between this account and another specified account. Validates the transaction based on the balance and the transaction amount. Adds the transaction to the bank's and the account's transaction list.
- **processTransaction(double amount, String transactionType)**: Overloaded method to process transactions that do not involve another account, such as deposits and loan disbursements. Validates transactions based on type and balance.
- **takeLoan(String loanId, double loanAmount)**: Attempts to take a loan from the bank. Checks for loan availability, sets up the loan, and handles disbursement.
- **getBalance()**: Returns the current balance of the account.
- **getAccountId()**: Retrieves the unique account ID.
- **getAccountOwner()**: Retrieves the name of the account owner.
- **setBalance(double balance)**: Sets or updates the account's balance.
- **setAccountId(String accountId)**: Sets or updates the account's ID.
- **setAccountOwner(String accountOwner)**: Sets or updates the name of the account owner.
- **setPassword(String password)**: Sets or updates the account's password.
- **getPassword()**: Retrieves the account's password.
- **getTransactionList()**: Retrieves the list of transactions associated with the account.
- **getTakenLoans()**: Retrieves the list of loans taken by the account holder.

1.1.3. Transaction Class:

```
public class Transaction {
    private String transactionId;
    private String transactionType;
    private Account fromAccount;
    private Account toAccount;
    private double amount;
    private String transactionDate;

    public Transaction(Account fromAccount, Account toAccount, double amount, String transactionDate, String transactionType) {
        String newTransactionID = String.valueOf(new Random().nextInt(1000) + 1);
        boolean found = true;
        while (found) {
            for (int i=0 ;i<Bank.transactions.size();i++) {
                if (Bank.transactions.get(i).equals(newTransactionID)) {
                    newTransactionID = String.valueOf(new Random().nextInt(1000) + 1);
                    i=0;
                }
            }
            found = false;
        }
        this.transactionId = newTransactionID;
        this.fromAccount = fromAccount;
        this.toAccount = toAccount;
        this.amount = amount;
        this.transactionDate = transactionDate;
        this.transactionType = transactionType;
        fromAccount.transactionList.add(this);
        toAccount.transactionList.add(this);
    }

    public Transaction(Account fromAccount, double amount, String transactionDate, String transactionType) {
        String newTransactionID = String.valueOf(new Random().nextInt(1000) + 1);
        boolean found = true;
        while (found) {
            for (int i=0 ;i<Bank.transactions.size();i++) {
                if (Bank.transactions.get(i).equals(newTransactionID)) {
                    newTransactionID = String.valueOf(new Random().nextInt(1000) + 1);
                    i=0;
                }
            }
            found = false;
        }
        this.transactionId = newTransactionID;
        this.fromAccount = fromAccount;
        this.toAccount = null;
        this.amount = amount;
        this.transactionDate = transactionDate;
        this.transactionType = transactionType;
        fromAccount.transactionList.add(this);
    }
    public String getTransactionDetails(String transactionType) {
        if (transactionType == "T")
            return "Transaction ID: " + transactionId + ", Date: " + transactionDate +
                    ", From: " + fromAccount.getAccountId() + ", To: " + toAccount.getAccountId() + ", Amount: " + amount;
        else
            return "Transaction ID: " + transactionId + ", Date: " + transactionDate +
                    ", From: " + fromAccount.getAccountId() + ", Amount: " + amount;
    }

    public String getTransactionId() {
        return this.transactionId;
    }
    public String getTransactionDate() {
        return this.transactionDate;
    }
    public String getFromAccountId() {
        return this.fromAccount.getAccountId();
    }
    public String getToAccountId() {
        return this.toAccount.getAccountId();
    }
    public double getAmount() {
        return this.amount;
    }
    public String getTransactionType() {
        return this.transactionType;
    }
}
```

Figure 1.4: Transaction Class

1.2. Transaction Class Overview

The `Transaction` class is designed to manage the details and operations associated with financial transactions in the banking system. Below is a detailed breakdown of its attributes and methods:

1.2.1. Attributes

- **transactionId**: A unique identifier for the transaction, constructed dynamically to ensure uniqueness within the system by generating random integers.
- **transactionType**: The type of transaction, which can be a transfer between accounts, a deposit, or a loan payment.
- **fromAccount**: The account from which the transaction originates. This attribute is always populated.
- **toAccount**: The account to which the transaction is directed. This attribute may be null for transactions that do not involve a recipient account, such as deposits or withdrawals.
- **amount**: The monetary value of the transaction.
- **transactionDate**: The date on which the transaction occurred, formatted appropriately.

1.2.2. Methods

- **Constructor (two versions)**: Initializes a new instance of the `Transaction` class. One constructor handles transfers between two accounts, while the other handles transactions involving only a single account.
- **getTransactionDetails(String transactionType)**: Returns a string detailing the transaction. The string includes the transaction ID, date, accounts involved, and the amount. The details vary based on the transaction type.
- **getTransactionId()**: Returns the unique transaction identifier.
- **getTransactionDate()**: Retrieves the date the transaction was recorded.
- **getFromAccountId()**: Retrieves the account ID of the sender.
- **getToAccountId()**: Retrieves the account ID of the recipient, if applicable.
- **getAmount()**: Returns the amount involved in the transaction.
- **getTransactionType()**: Identifies the type of transaction, providing context for the transaction operation.

1.2.3. Loan Class:

```
class Loan {
    private String loanId;
    private double loanAmount;
    private Account loanAccount;
    private int period;
    private int startYear;
    private double interestRate;

    public Loan(String loanId, double loanAmount, Account loanAccount, double intR, int p) {
        this.loanId = loanId;
        this.interestRate = intR;
        this.period = p;
        this.loanAmount = loanAmount;
        this.loanAccount = loanAccount;
    }

    public String getLoanId() {
        return loanId;
    }

    public double getInterestRate() {
        return interestRate;
    }

    public boolean disburseLoan() {
        boolean isSuccess = loanAccount.processTransaction(loanAmount,"DL"); //Change to disburse loan
        startYear = Year.now().getValue();
        return isSuccess;
    }

    public boolean makePayment() {
        if (Year.now().getValue() - startYear > period) {
            //System.out.println("you exceeded loan payment date");
            return false;
        }
        boolean temp = loanAccount.processTransaction(loanAmount + (loanAmount*(interestRate/100)),"PL");
        return temp;
    }

    public void setStartYear(int startYear) {
        this.startYear = startYear;
    }

    public int getStartYear() {
        return startYear;
    }

    public int getPeriod() {
        return period;
    }

    public double getLoanAmount(){
        return loanAmount;
    }

    public void setLoanAmount(double loanAmount){
        this.loanAmount=loanAmount;
    }
    public void setLoanAccount(Account loanAccount){
        this.loanAccount=loanAccount;
    }
}
```

Figure 1.5: Loan Class

Attributes

- **loanId**: A unique identifier for each loan, ensuring each loan can be distinctly managed and tracked.
- **loanAmount**: The principal amount of the loan.
- **loanAccount**: The account to which the loan is disbursed and from which repayments are made.
- **period**: The duration of the loan in years, over which the loan should be repaid.
- **startYear**: The year the loan was originally disbursed. This is set during the disbursement of the loan and helps in calculating the tenure and repayment schedule.

- **interestRate**: The annual interest rate applied to the loan, used to calculate the total repayment amount including interest.

Methods

- **getLoanId()**: Returns the unique identifier of the loan.
- **getInterestRate()**: Retrieves the interest rate of the loan.
- **disburseLoan()**: Handles the disbursement of the loan amount to the loan account. It processes a transaction of type "DL" to deposit the loan amount into the loan account.
- **makePayment()**: Manages the repayment of the loan. This method checks if the loan payment period has been exceeded. If within the period, it processes a repayment transaction, including interest, of type "PL".
- **setStartYear(int startYear)**: Sets the year when the loan was disbursed. This is crucial for managing the repayment schedule.
- **getStartYear()**: Retrieves the year the loan was disbursed.
- **getPeriod()**: Retrieves the duration of the loan.
- **getLoanAmount()**: Retrieves the amount of the loan.
- **setLoanAmount(double loanAmount)**: Sets the loan amount. This could be used when adjusting the principal before disbursement.
- **setLoanAccount(Account loanAccount)**: Associates a specific bank account with the loan, facilitating the disbursement and repayment transactions.

2

Requirement 1
Unit Testing

2.1. JUnit Test Classes

2.1.1. AccountTest Class

First, we defined **BeforeAll** to be executed before running the tests and **AfterAll** to be executed after running the tests. We initialized loans.

```
@BeforeAll  MaeenMo
public static void setUp() {
    System.out.println("\n-------------------------");
    System.out.println("Account Class Test Started");
    System.out.println("-------------------------\n");
    Bank.loans.add(new Loan( loanId: "L2",  loanAmount: 0,  loanAccount: null,  intR: 15,  p: 3));
    Bank.loans.add(new Loan( loanId: "L3",  loanAmount: 0,  loanAccount: null,  intR: 20,  p: 5));
}
@AfterAll  MaeenMo
public static void tearDown() {
    System.out.println("\n-----------------------");
    System.out.println("Account Class Test Ended");
    System.out.println("-----------------------\n");
}
```

Figure 2.1: BeforeAll and AfterAll in AccountTest

Then, we defined an order to our first test function, which has the order (1) and tests the deposit, we also added a name to it. Then, we assert that the final balance = the deposited amount + the old balance value.

```
@Order(1)  MaeenMo
@Test
@DisplayName("Test Deposit to User Account")
public void testDeposit() {
    double oldBalance = a1.getBalance();
    a1.processTransaction( amount: 1000,  transactionType: "D");
    assertEquals( expected: oldBalance + 1000, a1.getBalance());
}
```

Figure 2.2: Deposit Test

Then, we defined the second test function, which has the order (2) and tests the withdraw with an amount less than the balance, we also added a name to it. Then, we assert that the final balance = the old balance value - the withdrawn amount.

```
@Order(2)  MaeenMo
@Test
@DisplayName("Test Withdraw With Sufficient Amount from User Account")
public void testWithdrawWithSufficientAmount() {
    double oldBalance = a1.getBalance();
    a1.processTransaction( amount: 200,  transactionType: "W");
    assertEquals( expected: oldBalance - 200, a1.getBalance());
}
```

Figure 2.3: Withdraw Test (Sufficient Amount)

Then, we defined the third test function, which has the order (3) and tests the withdraw with an amount more than the balance, we also added a name to it. Then, we assert that the final balance doesn't equal the old balance value - the withdrawn amount because the withdraw function should return an error because it's not possible to withdraw an amount more than the balance.

```
@Order(3)  MaeenMo
@Test
@DisplayName("Test Withdraw Insufficient Amount from User Account")
public void testWithdrawWithInSufficientAmount() {
    double oldBalance = a1.getBalance();
    a1.processTransaction( amount: 30000,  transactionType: "W");
    assertFalse( condition: oldBalance - 30000 == a1.getBalance());
}
```

Figure 2.4: Withdraw Test (Insufficient Amount)

Then, we defined the fourth test function, which has the order (4) and tests the transfer in case of success, we also added a name to it. Then, we assert that the final balance of the user = the old balance value - the transferred amount and also assert that the destination account balance = the old balance value + the transferred amount.

```
@Order(4)  MaeenMo
@Test
@DisplayName("Test Valid Transfer Money")
public void testProcessTransferValid(){
    double oldBalanceA1 = a1.getBalance();
    double oldBalanceA2 = a2.getBalance();
    a1.processTransaction(a2,  amount: 1000,  TransactionType: "T");
    assertEquals( expected: oldBalanceA1 - 1000,a1.getBalance());
    assertEquals( expected: oldBalanceA2 + 1000,a2.getBalance());

}
```

Figure 2.5: Transfer Test (Valid Transfer)

Then, we defined the fifth test function, which has the order (5) and tests the transfer in case of fail, we also added a name to it. Then, we assert that the final balance of the user doesn't equal the old balance value - the transferred amount and also assert that the destination account balance doesn't equal the old balance value + the transferred amount, that's because the transfer will fail because the transfer amount is greater than the user's balance.

```
@Order(5)  MaeenMo
@Test
@DisplayName("Test Fail Transfer Money")
public void testProcessTransferFail(){
    double oldBalanceA1 = a1.getBalance();
    double oldBalanceA2 = a2.getBalance();
    a1.processTransaction(a2,  amount: 2000,  TransactionType: "T");
    assertFalse( condition: oldBalanceA1 - 1000 == a1.getBalance());
    assertFalse( condition: oldBalanceA2 + 1000 == a2.getBalance());

}
```

Figure 2.6: Transfer Test (Invalid Transfer)

Then, we defined the sixth test function, which has the order (6) and tests the take loan success, we also added a name to it. Then, we assert that the loan has been added to the taken loans arraylist and that the final balance = the old balance + the loan value.

```
@Order(6)  MaeenMo
@Test
@DisplayName("Test take 5 years loan successfully")
public void testTakeLoan5Years() {
    double oldBalance = a1.getBalance();
    a1.takeLoan( loanId: "L3",  loanAmount: 8000 );
    assertTrue( a1.takenLoans.getLast().getLoanId().equals("L3"));
    assertEquals(a1.getBalance(),  actual: 8000+oldBalance);
}
```

Figure 2.7: Take Loan 5 years Success

Then, we defined the seventh test function, which has the order (7) and tests the pay loan success, we also added a name to it. Then, we assert that the loan has been added to the paid loans arraylist and that the final balance = the old balance - (the loan value + loan value * interest).

```
@Order(7)  MaeenMo
@Test
@DisplayName("Test pay 5 years loan successfully")
public void testPayLoan5Years() {
    double oldBalance = a1.getBalance();
    assertTrue( condition: a1.takenLoans.get(0).makePayment()==null);
    assertEquals(a1.getBalance(),  actual: oldBalance - (8000 + (8000*0.2)));
}
```

Figure 2.8: Take Pay 5 years Success

Then, we defined the eighth test function, which has the order (8) and tests the take loan success, we also added a name to it. Then, we assert that the loan has been added to the taken loans arraylist and that the final balance = the old balance + the loan value.

```
@Order(8)  MaeenMo
@Test
@DisplayName("Test take 3 years loan successfully")
public void testTakeLoan3Years() {
    double oldBalance = a1.getBalance();
    a1.takeLoan( loanId: "L2",  loanAmount: 10000);
    a1.takenLoans.getLast().setStartYear(2020);
    assertTrue(a1.takenLoans.getLast().getLoanId().equals("L2"));
    assertEquals(a1.getBalance(),  actual: 10000+oldBalance);
}
```

Figure 2.9: Take Loan 3 years Success

Then, we defined the ninth test function, which has the order (9) and tests the pay loan fail, we also added a name to it. Then, we assert that the loan hasn't been added to the paid loans arraylist and that the final balance doesn't equal the old balance - (the loan value + loan value * interest) as it will fail because the loan payment date has been exceeded.

```
@Order(9)  MaeenMo
@Test
@DisplayName("Test pay 3 years loan fail")
public void testPayLoan3Years() {
    double oldBalance = a1.getBalance();
    assertFalse( condition: a1.takenLoans.get(0).makePayment()==null);
    assertFalse( condition: a1.getBalance() == oldBalance - (10000 + (10000*0.1)));
}
```

Figure 2.10: Take Pay 3 years Fail

2.1.2. Loan Test Class

First, we defined **BeforeAll** to be executed before running the tests and **AfterAll** to be executed after running the tests. We initialized an account and loans.

```
@BeforeAll  MaeenMo
static void setUp() {
    System.out.println("\n------------------------");
    System.out.println("Loan Class Test Started");
    System.out.println("------------------------\n");
    account = new Account( accountId: "1",  accountOwner: "Joe",  initialBalance: 1000,  password: "12345");
    loan = new Loan( loanId: "1",  loanAmount: 5000,account,  intR: 15,  p: 5);
    loanFail = new Loan( loanId: "2",  loanAmount: 10000,account, intR: 10, p: 3 );
}

@AfterAll  MaeenMo
public static void tearDown() {
    System.out.println("\n-----------------------------------");
    System.out.println("Loan Class Tested Successfully");
    System.out.println("-----------------------------------\n");
}
```

Figure 2.11: BeforeAll and AfterAll in LoanTest

Then, we tested that a loan can be taken successfully, then we assert that final balance = old balance + loan value.

```
@Test  MaeenMo
@Order(1)
void testDisburseSuccessLoan() {
    double oldBalance = account.getBalance();
    loan.disburseLoan();
    assertEquals(account.getBalance(),  actual: loan.getLoanAmount()+oldBalance);
}
```

Figure 2.12: Disburse Loan Success 1

Then, we tested that a loan can be paid successfully, then we assert that final balance = old balance - (loan value + loan value * interest rate).

```
@Test  MaeenMo *
@Order(2)
void testPaySuccessLoan() {
    double oldBalance = account.getBalance();
    loan.makePayment();
    assertEquals(account.getBalance(),  actual: oldBalance - (loan.getLoanAmount() +
            (loan.getLoanAmount()*(loan.getInterestRate()/100))));
}
```

Figure 2.13: Pay Loan Success

Then, we tested the take loan success, then we assert that final balance = old balance + loan value, but this will cause in a pay loan fail because the loan was disbursed in 2020 and after 4 years the loan payment date is exceeded.

```
@Test  MaeenMo
@Order(3)
void testDisburseFailLoan() {
    double oldBalance = account.getBalance();
    loanFail.disburseLoan();
    loanFail.setStartYear(2020);
    assertEquals(account.getBalance(),  actual: loanFail.getLoanAmount()+oldBalance);
}
```

Figure 2.14: Disburse Loan Success 2

Then, we tested the pay loan fail, then we assert that final balance doesn't equal old balance - (loan value + loan value * interest rate), it failed because the loan payment date has been exceeded.

```
@Test  MaeenMo
@Order(4)
void testPayFailLoan() {
    double oldBalance = account.getBalance();
    loanFail.makePayment();
    assertFalse( condition: account.getBalance() == oldBalance-loanFail.getLoanAmount());
}
```

Figure 2.15: Pay Loan Fail

2.1.3. Transaction Test Class

First, we defined **BeforeAll** to be executed before running the tests and **AfterAll** to be executed after running the tests. We initialized transactions.

```
public static Transaction t1= new Transaction(AccountTest.a2, AccountTest.a1,  amount: 3000,  7 usages
        transactionDate: "14/04/2024",  transactionType: "T");
public static Transaction t2= new Transaction(AccountTest.a1,  amount: 3000,  6 usages
        transactionDate: "14/04/2024",  transactionType: "W");

@BeforeAll  MaeenMo
@DisplayName("Setting Up Objects To Start Transaction Class Test")
public static void setUp() {
    System.out.println("\n---------------------------------");
    System.out.println("Transaction Class Test Started");
    System.out.println("---------------------------------\n");
}

@AfterAll  MaeenMo
@DisplayName("End of Transaction Class Test")
public static void tearDown() {
    System.out.println("\n-------------------------------------");
    System.out.println("Transaction Class Test Ended");
    System.out.println("-------------------------------------\n");
}
```

Figure 2.16: BeforeAll and AfterAll in TransactionTest

Then, we test the get transaction details to check that it has been saved in the system.

```
@Test  MaeenMo
@DisplayName("Test Get Transactions Details For t1")
public void testGetTransactionDetailsT1(){
    assertEquals( expected: "Transaction ID: " + t1.getTransactionId() + ", Date: " + t1.getTransactionDate() +
                        ", From: " + t1.getTransactionFromAccountId() + ", To: " + t1.getTransactionToAccountId() +
                        ", Amount: " + t1.getTransactionAmount()
                    ,t1.getTransactionDetails(t1.getTransactionType()));
}
```

Figure 2.17: Transaction Details 1

Then, we test the get another transaction details to check that it has been saved in the system.

```
@Test  MaeenMo
@DisplayName("Test Get Transactions Details For t2")
public void testGetTransactionDetailsT2(){
    assertEquals( expected: "Transaction ID: " + t2.getTransactionId() + ", Date: " + t2.getTransactionDate() +
                  ", From: " + t2.getTransactionFromAccountId() +
                  ", Amount: " + t2.getTransactionAmount()
            ,t2.getTransactionDetails(t2.getTransactionType()));
}
```

Figure 2.18: Transaction Details 2

2.1.4. Bank Test Class

First, we initialize some bank accounts and transactions in the BeforeAll, we also define AfterAll.

```
public static Account a1, a2, a3, a4, a5;  5 usages
public static Bank bank;  21 usages
public static Transaction t1, t2, t3, t4;  4 usages

@BeforeAll  MaeenMo
public static void setUp() {
    bank = new Bank();
    a1 = new Account( accountId: "1",  accountOwner: "Joe",  initialBalance: 0,  password: "12345");
    a2 = new Account( accountId: "2",  accountOwner: "Adam",  initialBalance: 10,  password: "67890");
    a3 = new Account( accountId: "3",  accountOwner: "Ben",  initialBalance: 100,  password: "13579");
    a4 = new Account( accountId: "4",  accountOwner: "Smith",  initialBalance: 1000,  password: "02468");
    a5 = new Account( accountId: "5",  accountOwner: "Tom",  initialBalance: 10000,  password: "87654");
    t1 = new Transaction(a2, a1,  amount: 3000, transactionDate: "13/03/2023",  transactionType: "T");
    t2 = new Transaction(a4,  amount: 1000, transactionDate: "14/04/2024", transactionType: "D");
    t3 = new Transaction(a3, a1,  amount: 90, transactionDate: "12/02/2022", transactionType: "T");
    t4 = new Transaction(a3,  amount: 7316, transactionDate: "18/05/2020", transactionType: "W");
    System.out.println("\n-----------------------");
    System.out.println("Bank Class Test Started");
    System.out.println("-----------------------\n");
}

@AfterAll  MaeenMo
static void tearDown() {
    System.out.println("\n----------------------");
    System.out.println("Bank Class Test Ended");
    System.out.println("----------------------\n");
}
```

Figure 2.19: BeforeAll and AfterAll in BankTest

Then, we test for creation of user accounts and we assert that the values we have entered while creating the instance of user is equal to the actual user data stored.

```
@Test  MaeenMo
@DisplayName("Test Create User Account")
public void testCreateUserAccount() {
    Account newUser1 = new Account( accountId: "100",  accountOwner: "Ahmed",  initialBalance: 500.0,  password: "A50004@x");
    Account newUser2 = new Account( accountId: "222",  accountOwner: "Mohamed",  initialBalance: 800.0,  password: "Axc43bkjhs");
    assertNotNull(newUser1);
    assertNotNull(newUser2);
    assertEquals( expected: "100", newUser1.getAccountId());
    assertEquals( expected: "222", newUser2.getAccountId());
    assertEquals( expected: "Ahmed", newUser1.getAccountOwner());
    assertEquals( expected: "Mohamed", newUser2.getAccountOwner());
    assertEquals( expected: 500.0, newUser1.getBalance());
    assertEquals( expected: 800.0, newUser2.getBalance());

}
```

Figure 2.20: Create Account Test

Then, we test for adding the created accounts to the accounts arraylist in the bank class.

```
@Test  MaeenMo
@DisplayName("Check added Accounts")
public void testAddedAccounts() {
    bank.addAccount(a1);
    bank.addAccount(a2);
    bank.addAccount(a3);
    bank.addAccount(a4);
    assertEquals(bank.getAccount( accountId: "1").getAccountId(),  actual: "1");
    assertEquals(bank.getAccount( accountId: "2").getAccountId(),  actual: "2");
    assertEquals(bank.getAccount( accountId: "3").getAccountId(),  actual: "3");
    assertEquals(bank.getAccount( accountId: "4").getAccountId(),  actual: "4");
    assertNotEquals(bank.getAccount( accountId: "4").getAccountId(),  actual: "3");
}
```

Figure 2.21: Add Account Test

Then, we test for retrieving each of the created accounts.

```
@Test  MaeenMo
@DisplayName("test Get Accounts")
public void testGetAccount(){
    assertEquals(a1, bank.getAccount( accountId: "1"));
    assertEquals(a2, bank.getAccount( accountId: "2"));
    assertEquals(a3, bank.getAccount( accountId: "3"));
    assertEquals(a4, bank.getAccount( accountId: "4"));
}
```

Figure 2.22: Get Account Test

Lastly, we assert for the transactions defined in the BeforeAll and check that the Balances have been updated accordingly with the transactions, the last one will fail because it withdraws from balance witha value more than the balance value.

```
@Test  MaeenMo *
@DisplayName("Check Transaction")
public void testTransaction() {
    bank.transactions.add(t1);
    bank.transactions.add(t2);
    bank.transactions.add(t3);
    assertEquals(bank.getTransaction(t1.getTransactionId()), t1);
    assertEquals(bank.getTransaction(t2.getTransactionId()), t2);
    assertEquals(bank.getTransaction(t3.getTransactionId()), t3);
    assertFalse( condition: bank.getTransaction(t4.getTransactionId()) == t4);
}
```

Figure 2.23: Transactions on Account Test

2.1.5. Test Suite

At the end, we defined a test suite to run all the previously explained tests at once.

```
@SuppressWarnings("deprecation")  MaeenMo
@RunWith(JUnitPlatform.class)
@SelectClasses({LoanTest.class, TransactionTest.class, BankTest.class, AccountTest.class})
public class AllTests {}
```

Figure 2.24: Test Suite

2.1.6. Tests Running

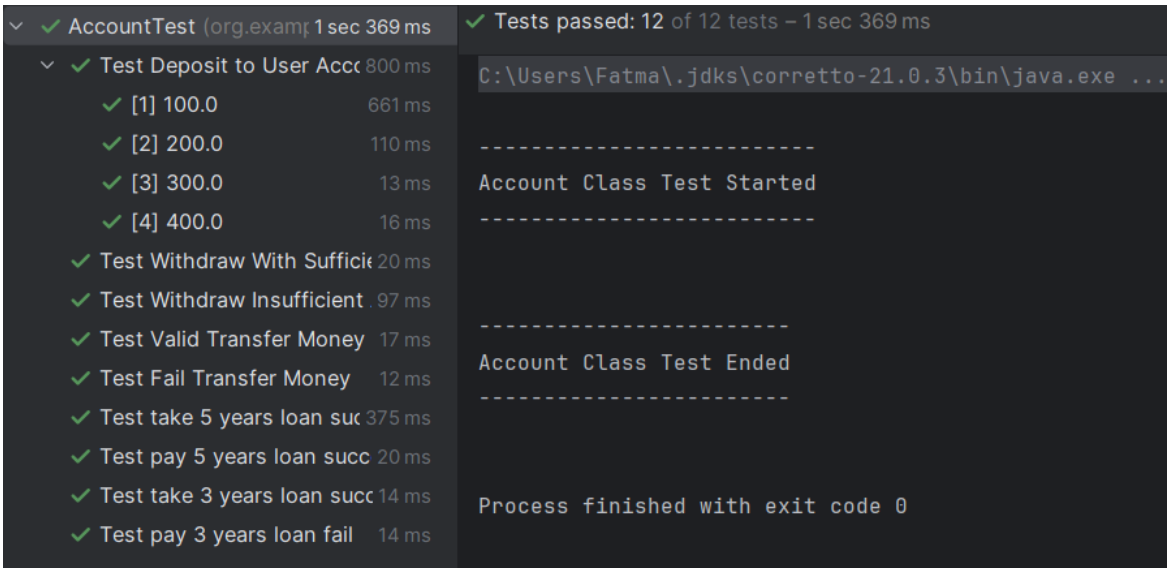


Figure 2.25: AccountTest Success

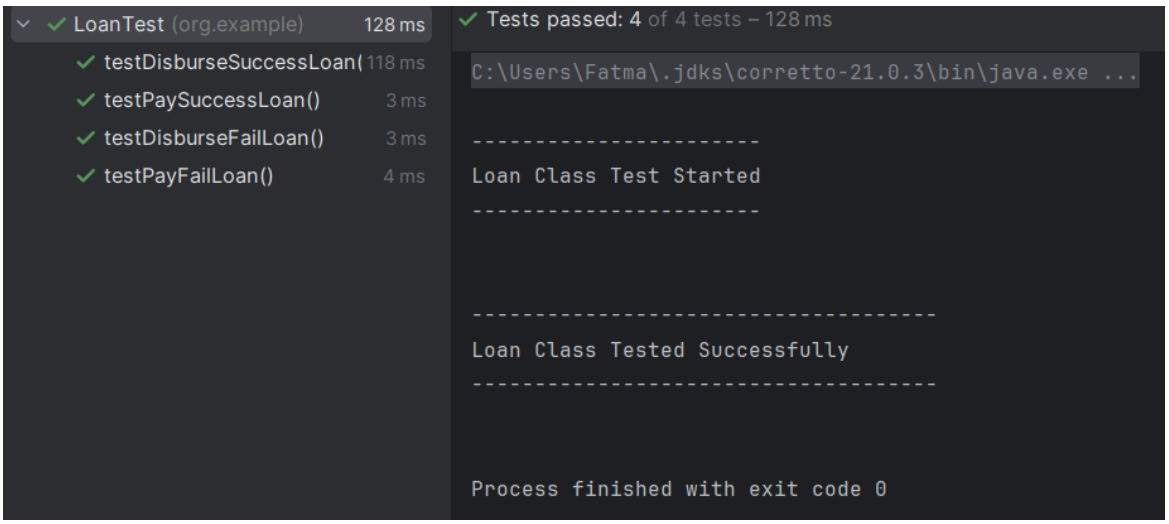


Figure 2.26: LoanTest Success

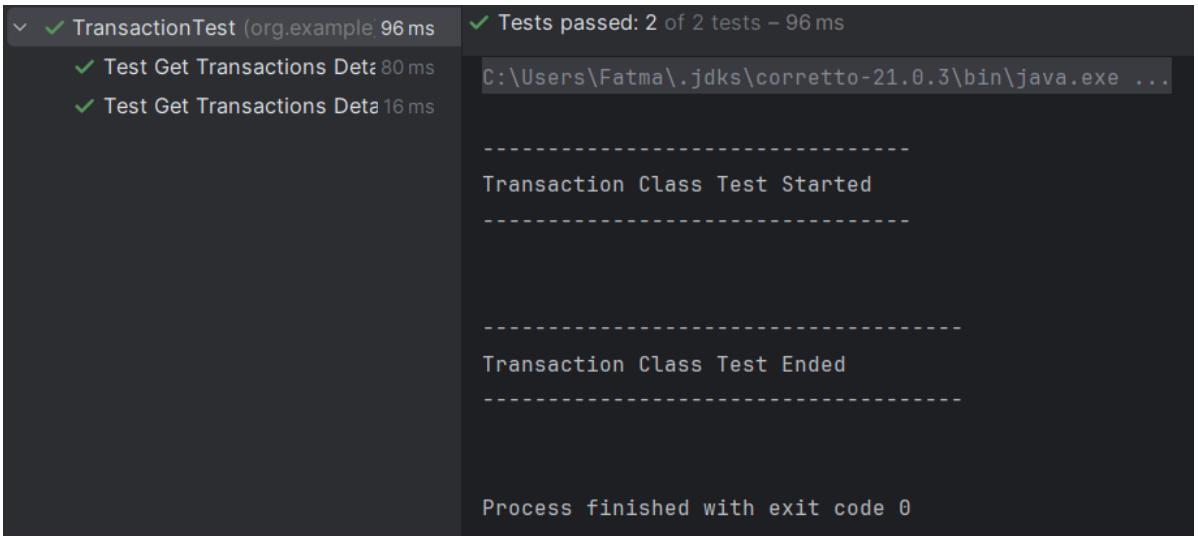


Figure 2.27: TransactionTest Success

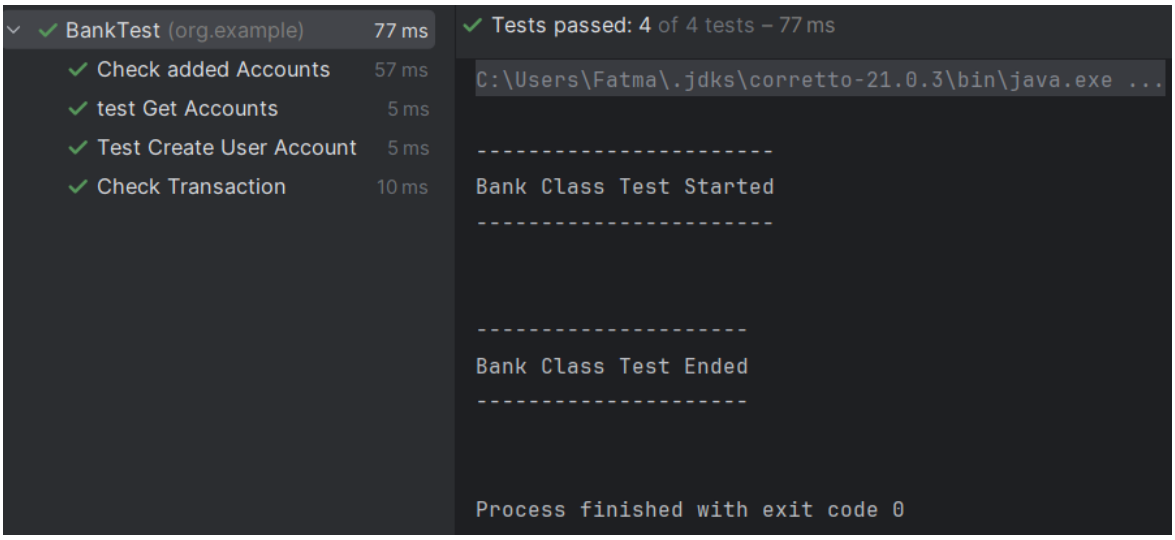


Figure 2.28: BankTest Success

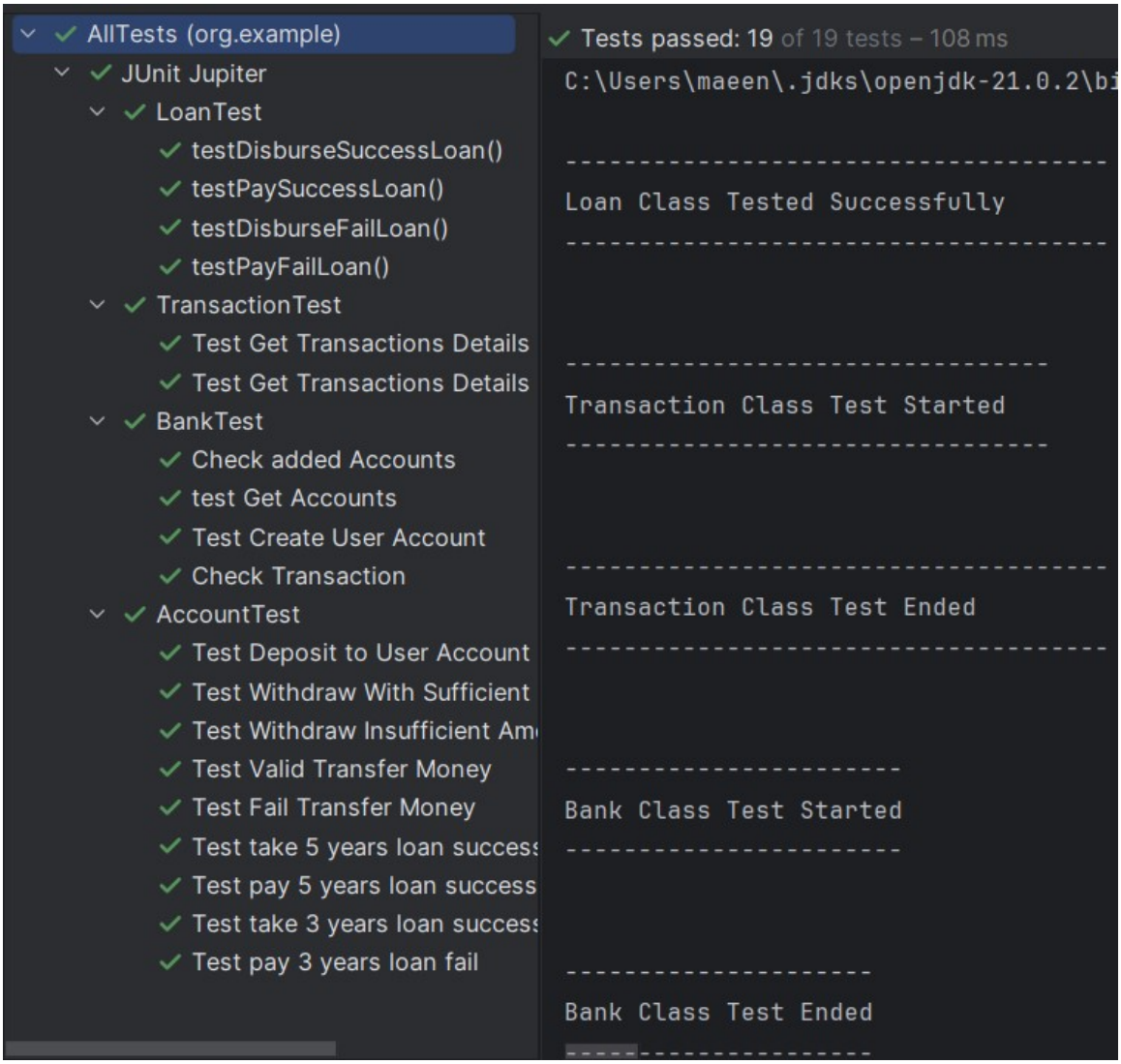
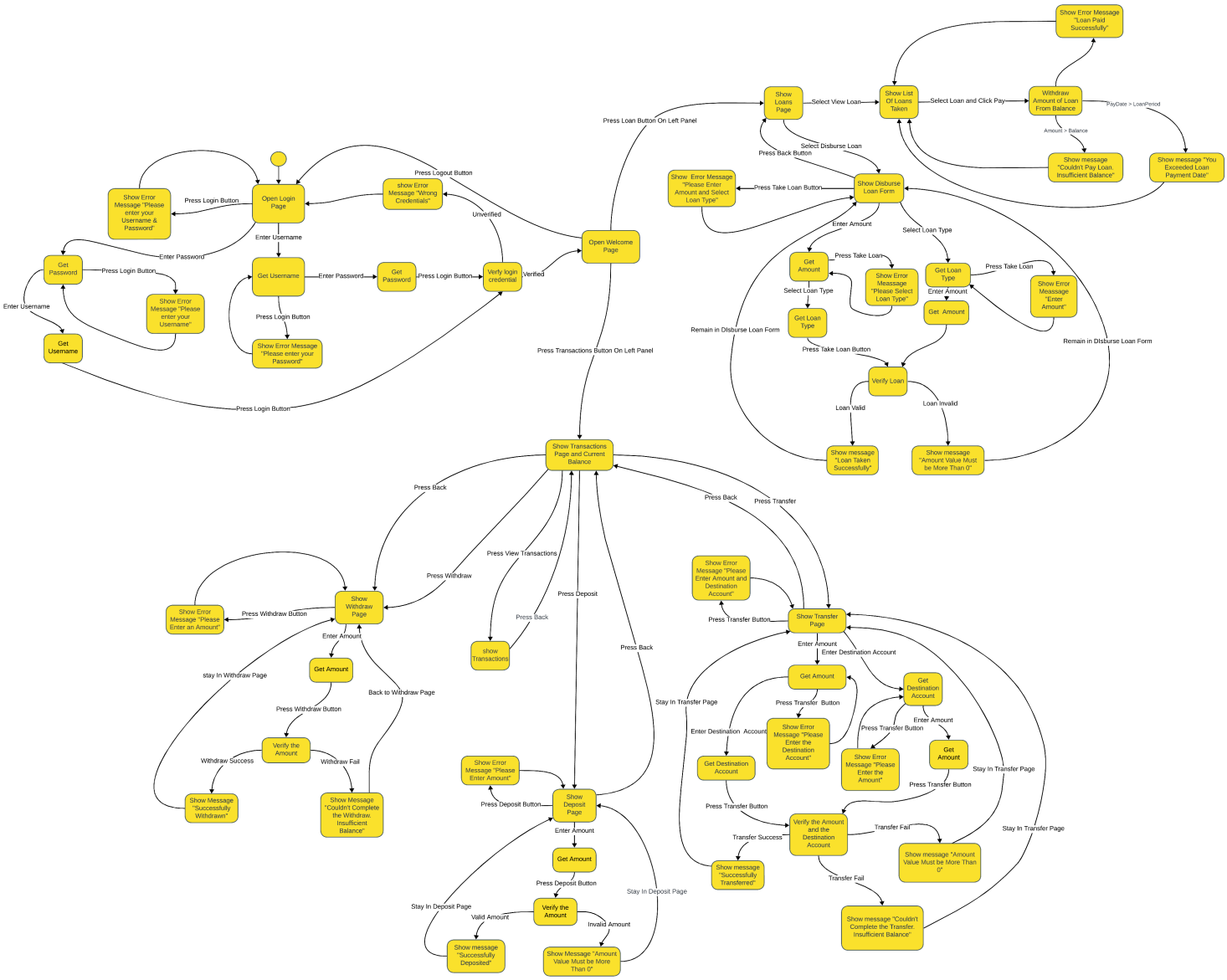


Figure 2.29: Test Suite Success

3

Requirement 2
GUI Testing

3.1. Finite State Machine(FSM):



3.2. Transitions and States Of The FSM:

3.2.1. The First Part Of the Finite State Machine Handles login Process:

- **Open Login Page:** The initial state where the user is prompted to enter login credentials.
- **Enter Username:** The user is expected to input their username.
- **Get Username:** The system retrieves the entered username.
- **Enter Password:** Following the username, the user must enter their password.
- **Get Password:** The system retrieves the entered password.
- **Verify login credentials:** The system checks the username and password combination.
- **Verified:** If the credentials are correct, access is granted.
- **Unverified:** If the credentials are incorrect, an error message is displayed.
- **Show Error Message:** Depending on the error, messages like “Please enter your Username & Password” or “Wrong Credentials!” are shown.
- **Open Welcome Page:** Upon successful login, the user is directed to the welcome page.
- **Press Logout Button:** The user can log out of the system.

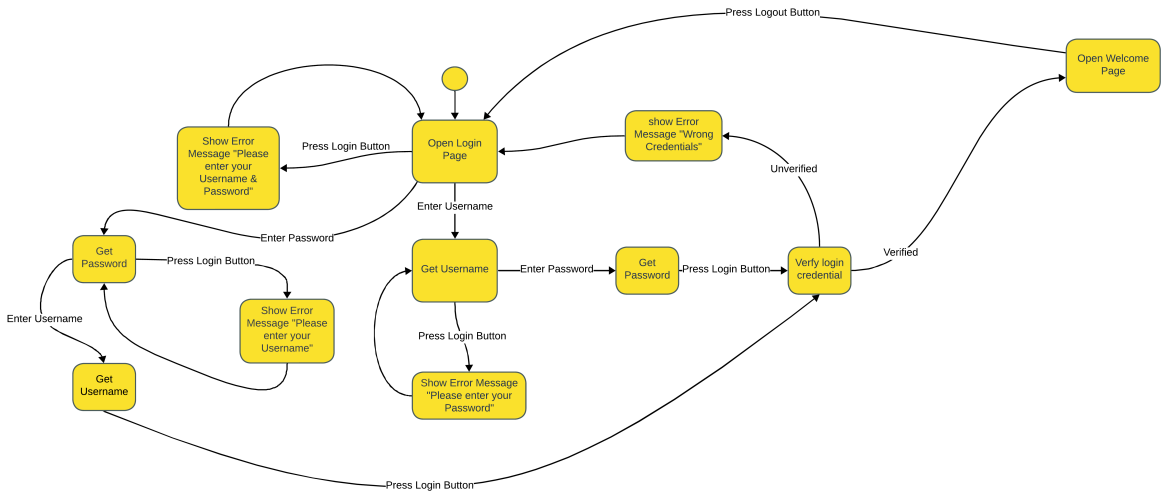


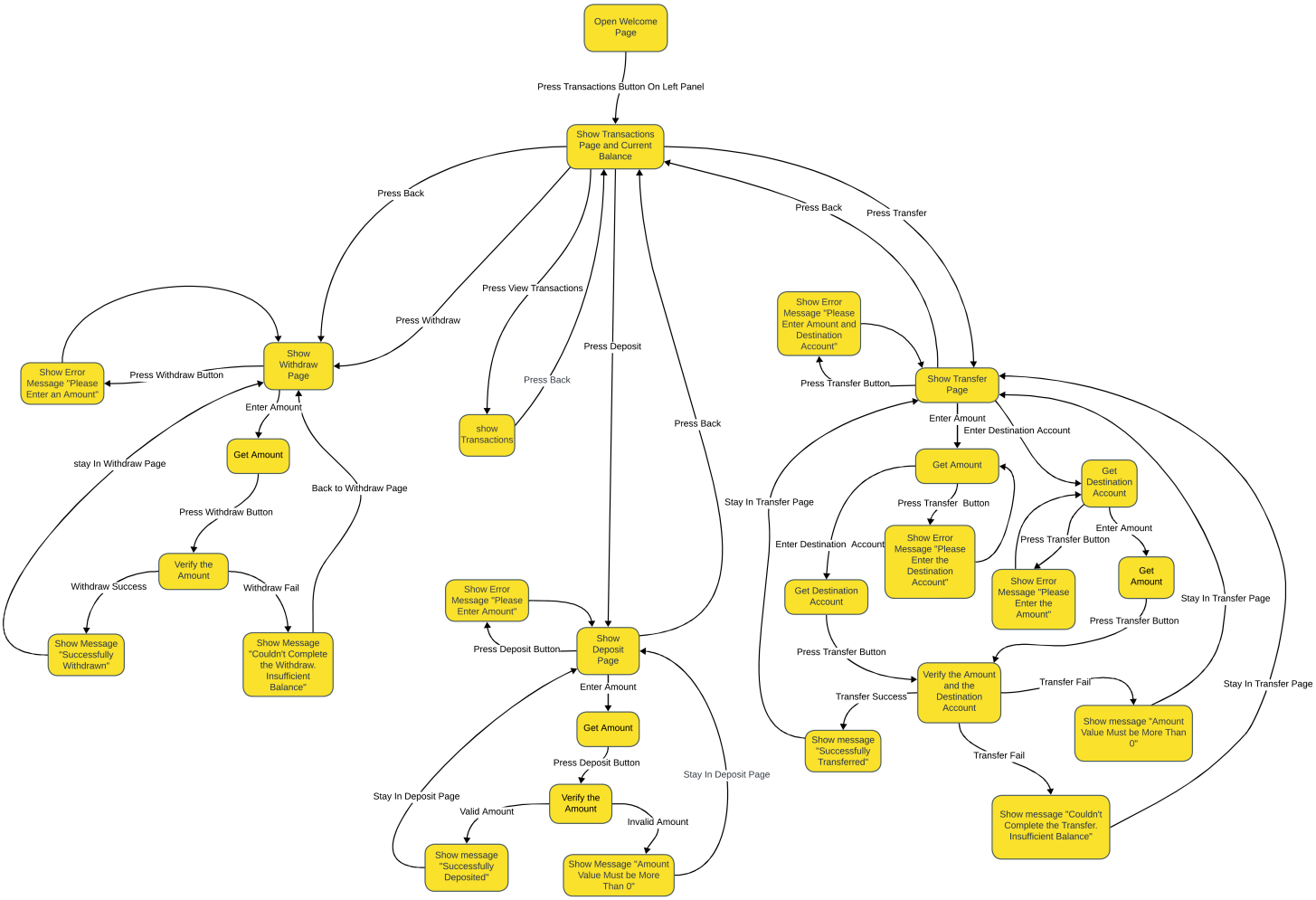
Figure 3.1: First Part of FSM

3.2.2. The Second Part Of the Finite State Machine Handles Transaction Process:

- **Press Transactions Button on Left Panel:** Display transactions menu and current balance.
- **Press Deposit Button:** Deposit window opens.
- **Enter Amount:** User is inputting the deposit amount.
- **Verify Amount:** System validates the entered amount.
- **Show Message:** The system checks the given amount and display a success message or an error message.
- **Press Withdraw Button:** Withdraw window opens.
- **Enter Amount:** User is inputting the withdraw amount.

- **Verify Amount:** System validates the entered amount.
- **Show Message:** The system checks the given amount and display a success message or an error message.
- **Press Transfer Button:** Transfer window opens.
- **Enter Destination Account:** User inputs the account for transferring funds.
- **Enter Transfer Amount:** User inputs the amount he want to transfer.
- **Verify the Amount and the Destination Account:** System validates the amount and destination account information.
- **Show Message:** The system checks the given account and the given amount and display a success message or an error message.
- **Press View Transactions:** View transactions window opens and all transactions related to this account are displayed.
- **Press Back Button:** Returns the user to the previous window of transactions menu.

Note: Transitions between these states are based on user actions, such as pressing the “Deposit”, "Withdraw" or "Transfer" buttons



3.2.3. The Third Part Of the Finite State Machine Handles Loan Process:

- **Press Loans Button on Left Panel:** Display loan menu and current balance.
- **Press Disburse Loan:** Display disburse loan screen.
- **Get Amount:** System retrieves the loan amount selected by the user.
- **Select Loan:**User selects a loan from the available options.
- **Verify Loan:** System checks the validity of the loan selection.
- **Loan Valid:** If the loan selection is valid, further processing continues.
- **Loan Invalid:** If the loan amount or loan type is invalid, an error message is displayed.
- **Show Error Message:** Error messages such as “Please Enter An Amount” or “Please Select Loan Type” are shown based on the error encountered.
- **Press Back Button:** Return to the previous menu.
- **Press View Loans Button:** Opens the Loans list for the current user.
- **Press Pay Loan Button:** If there are loans available user can pay them by pressing this button
- **Check loan start year:** If the user did not exceed the loan payment date, the loan is paid else error message appear.
- **Check available balance:** If the user have enough balance to pay the loan added to it its interest rate, the loan is paid else error message appear.
- **Press Back Button:** Returns the user to the previous window of loan menu.

Note: Transitions between these states are based on user actions, such as pressing the “Take Loan” or "Pay Loan" buttons

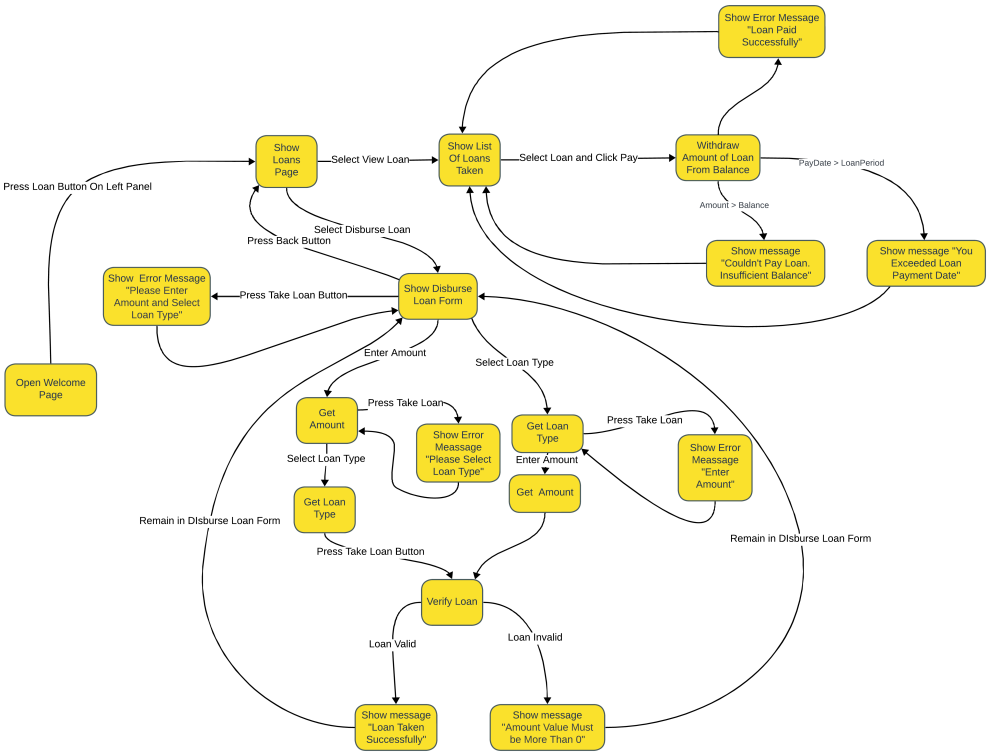


Figure 3.2: Second Part of FSM

3.3. GUI Screenshots With Validation:

3.3.1. Login Page:

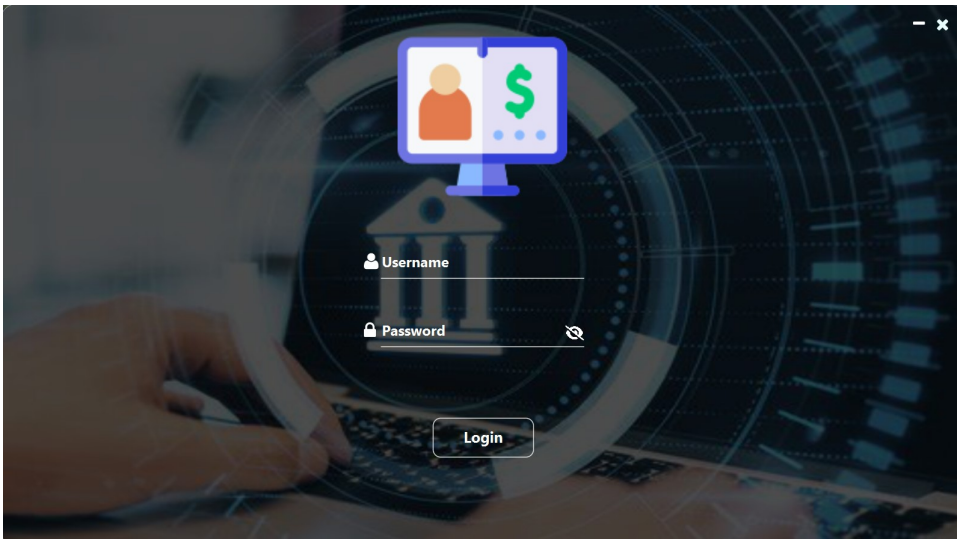


Figure 3.3: Login Page

If you press login without entering any data in username and password fields. Error Message appears.

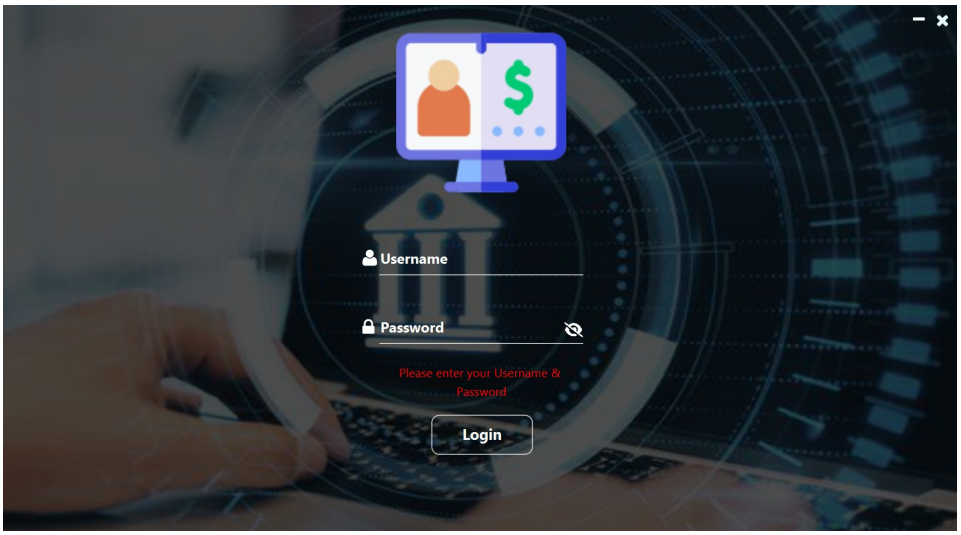


Figure 3.4: "Please enter your Username & Password"

If you press login without entering any data in password field. Error Message appears.

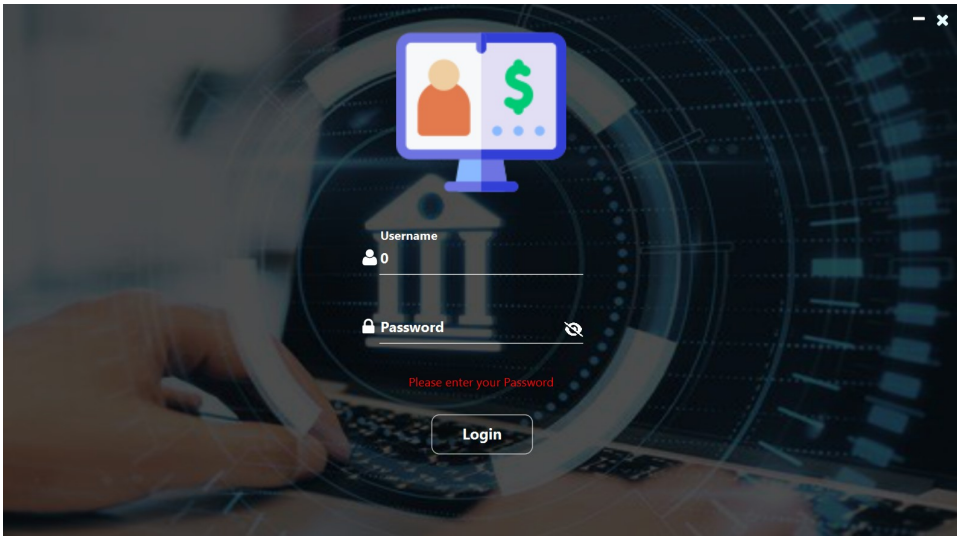


Figure 3.5: "Please enter your Password"

If you press login without entering any data in username field. Error Message appears.

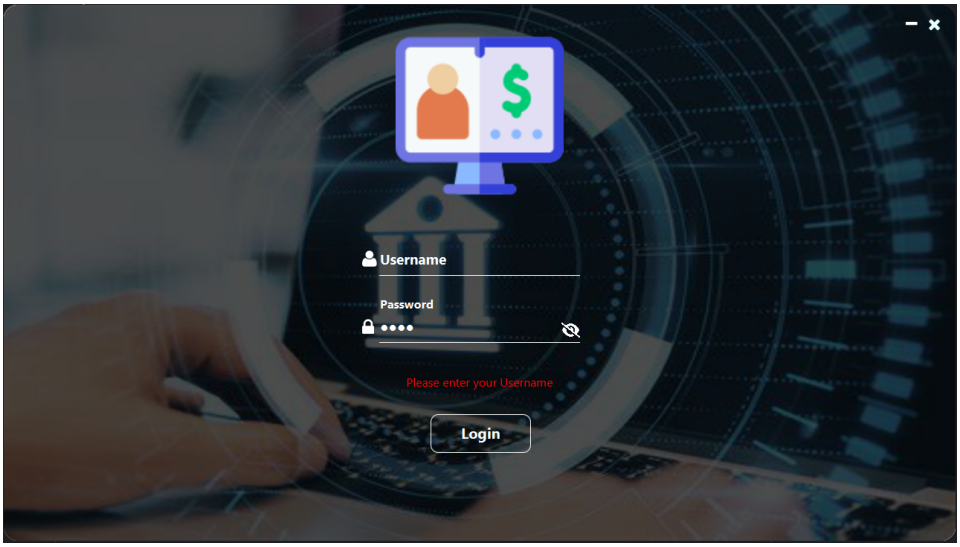


Figure 3.6: "Please enter your Username"

If you press login and username or password is wrong. Error Message appears.

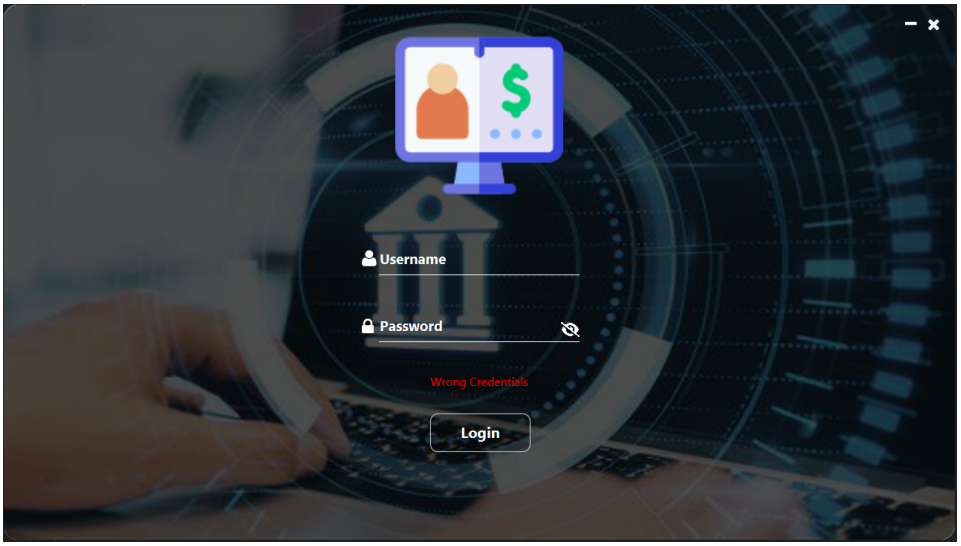


Figure 3.7: "Wrong Credentials"

After verified login welcome Page Appears.**SEE NEXT PAGE.**

3.3.2. Welcome Page:

Figure 3.8: Welcome Page

In welcome page you could choose to go to Transaction page or to Loan page by choosing them from the left side of the page. **See the Transaction page and the Loan Page below.**

3.3.3. Transaction Page:

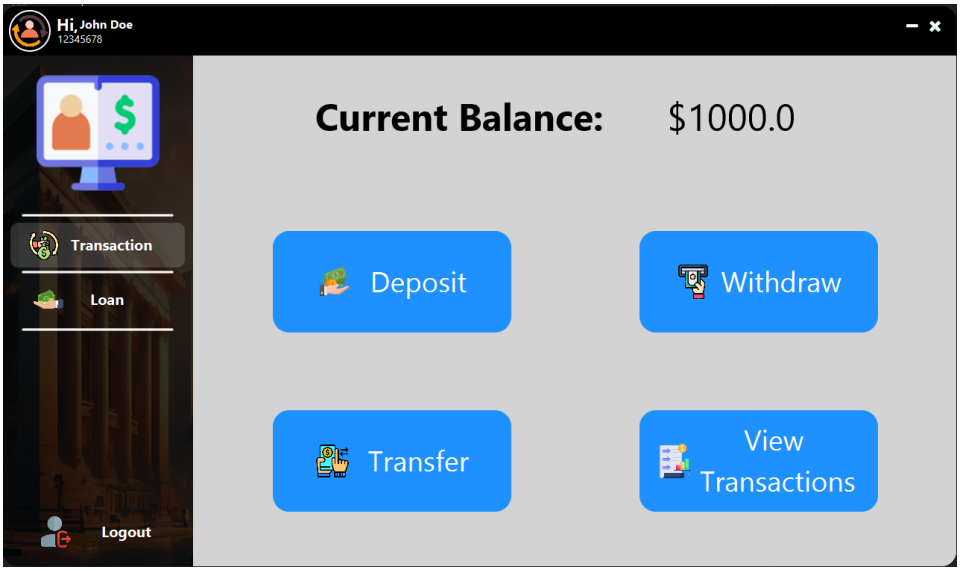


Figure 3.9: Transaction Page

Here you can choose to go to to Deposit page ,Withdraw Page , Transfer Page or View Transactions by pressing on the buttons containing their names. **See These Different Pages Below**

3.3.4. Deposit Page:

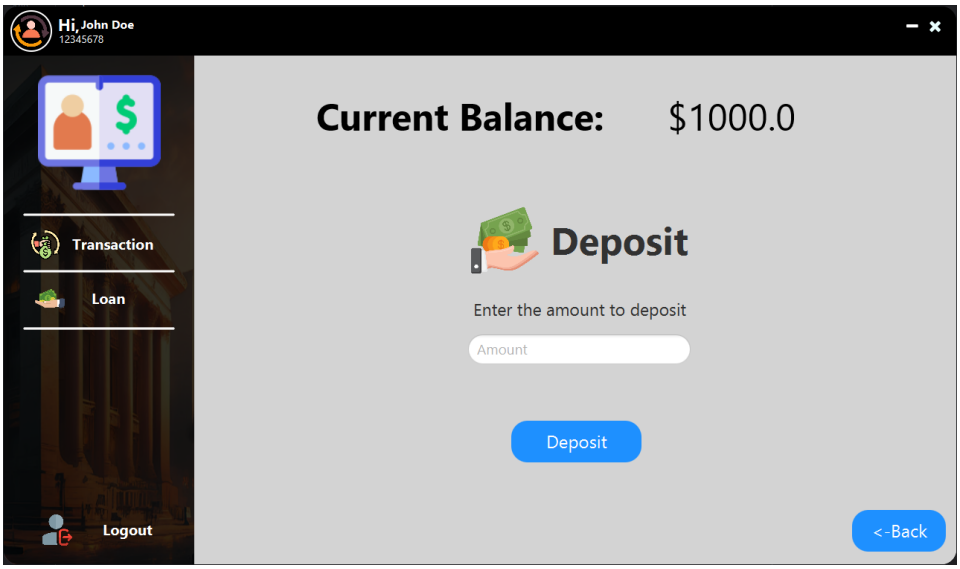


Figure 3.10: Deposit Page

If you press Deposit without entering an amount an Error message appears to tell you to enter an amount.

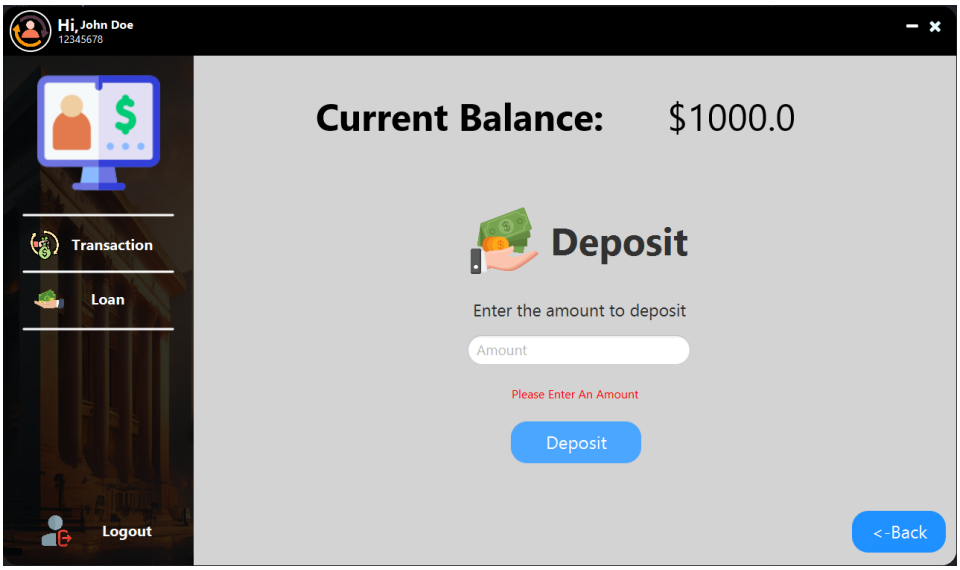


Figure 3.11: "Please Enter An Amount"

If you enter amount of 0 and you press Deposit an Error Message appears to tell you that the amount must be more than 0.

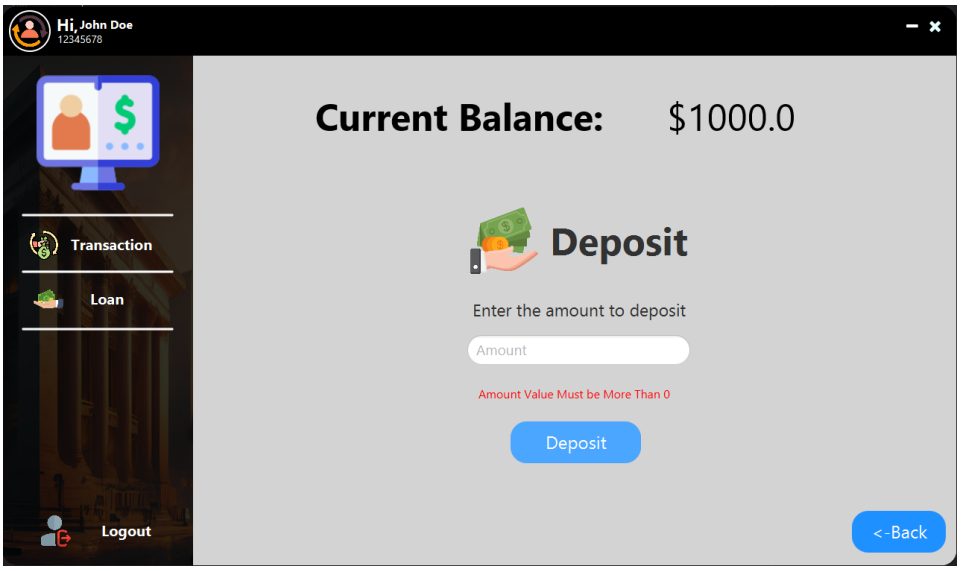


Figure 3.12: "Amount Value Must be More Than 0"

After Entering a valid Amount (E.g. 385) and press Deposit, Successfully Deposited message appears to you.

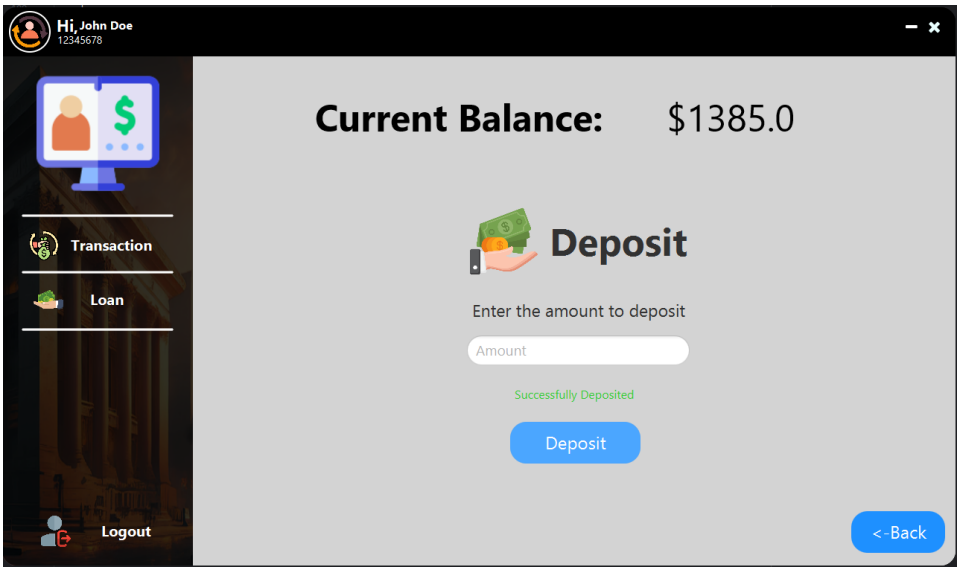


Figure 3.13: "Successfully Deposited"

3.3.5. Withdraw Page:

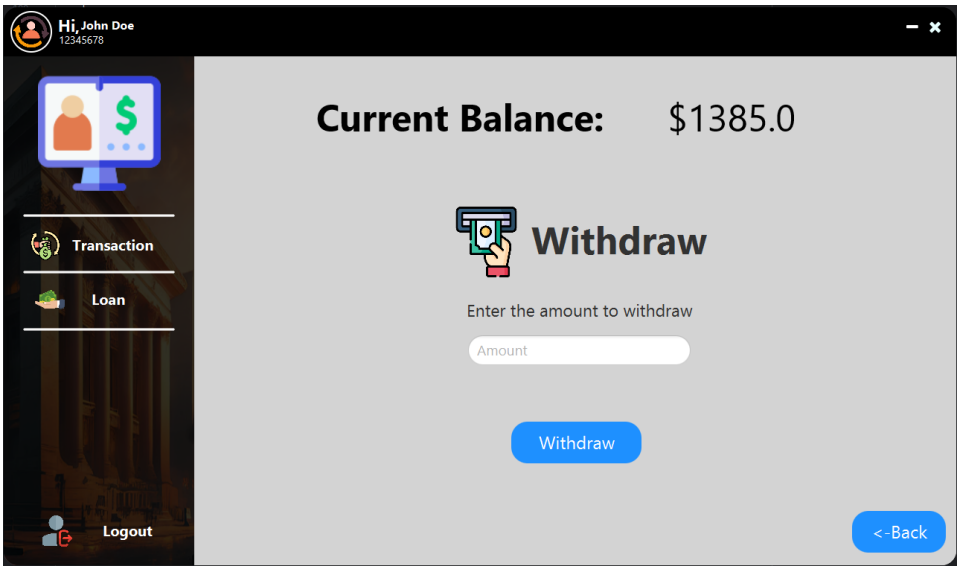


Figure 3.14: Withdraw Page

If you press Withdraw Button Without entering an Amount Error Message appears.

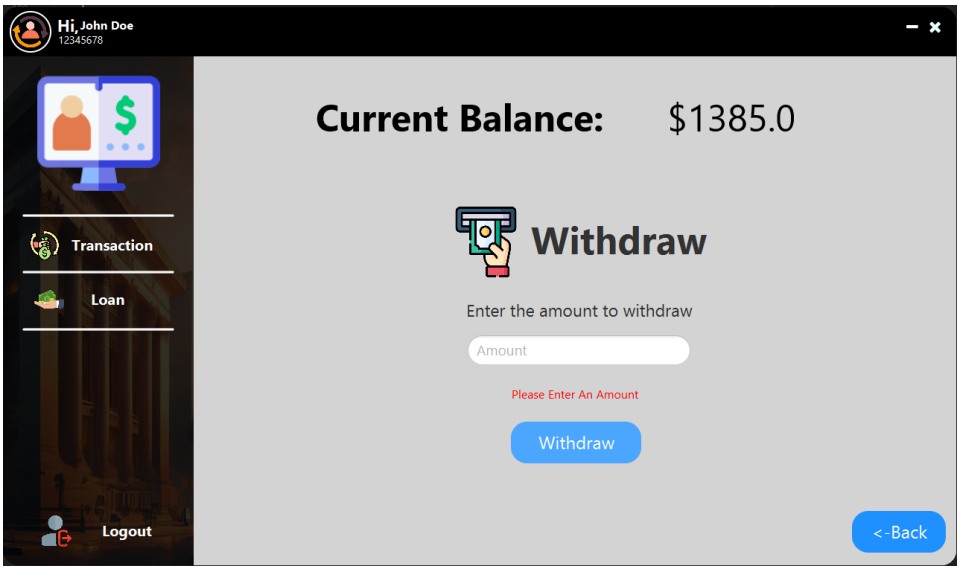


Figure 3.15: "Please Enter An Amount"

If you enter amount greater than your balance (E.g. 10000) then press Withdraw Button an Error Message appears.

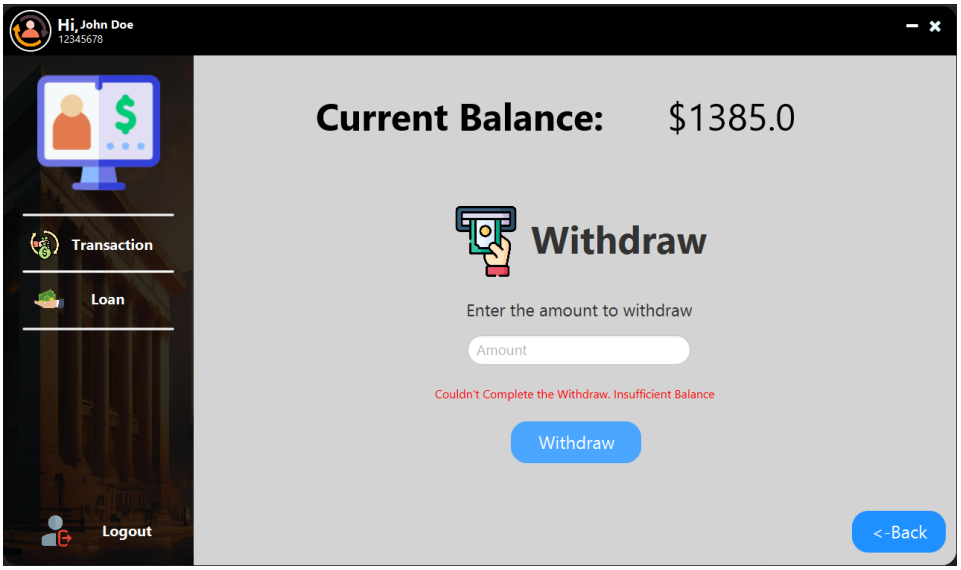


Figure 3.16: "Couldn't Complete the Withdraw. Insufficient Balance"

After Entering a valid Amount (E.g. 259.99) and press Withdraw Successful Message Appears.

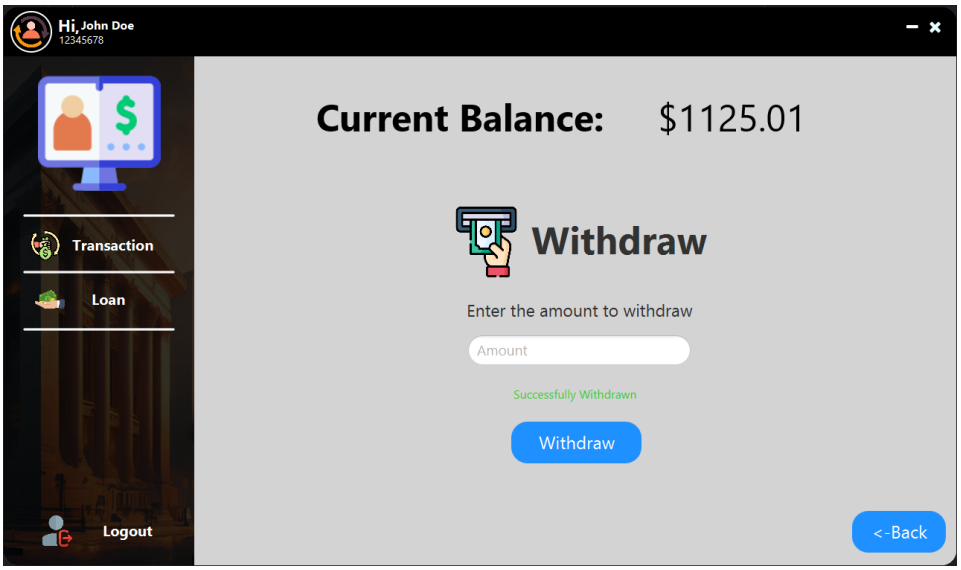


Figure 3.17: "Successfully Withdrawn"

3.3.6. Transfer Page:

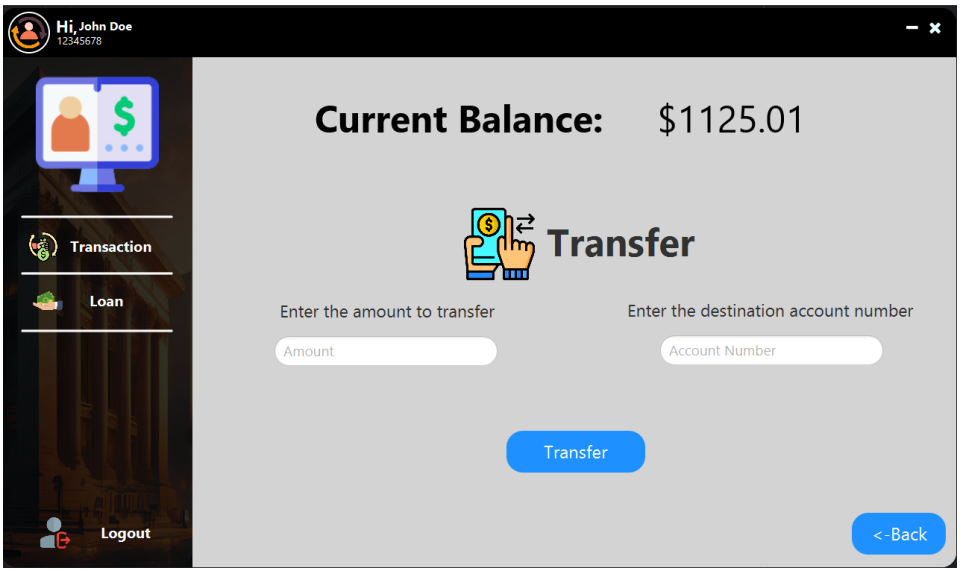


Figure 3.18: Transfer Page

If you press Transfer Button Without entering an Amount and Destination Account Error Message appears.

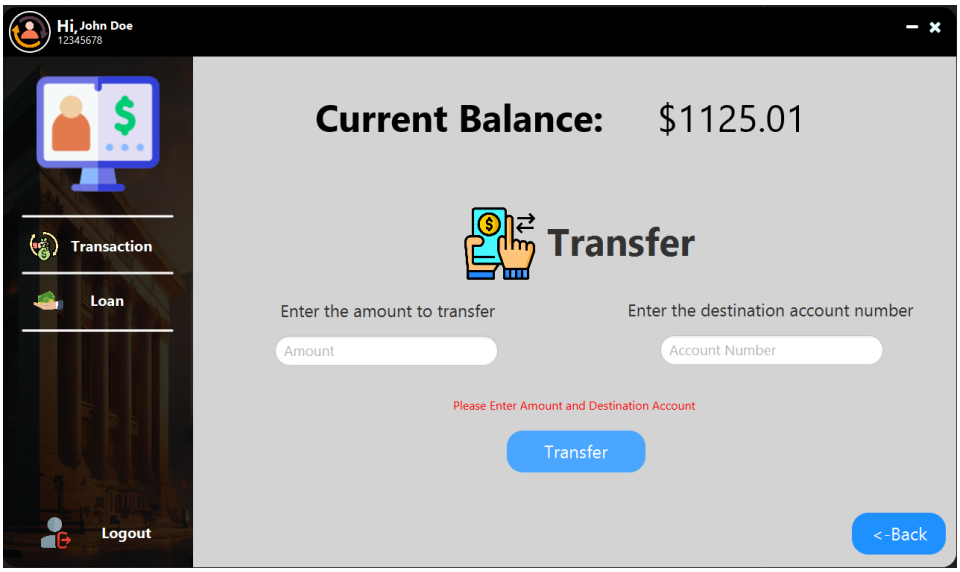


Figure 3.19: "Please Enter Amount and Destination Account"

If you press Transfer button after entering an Amount without entering the Destination Account an Error Message appears.

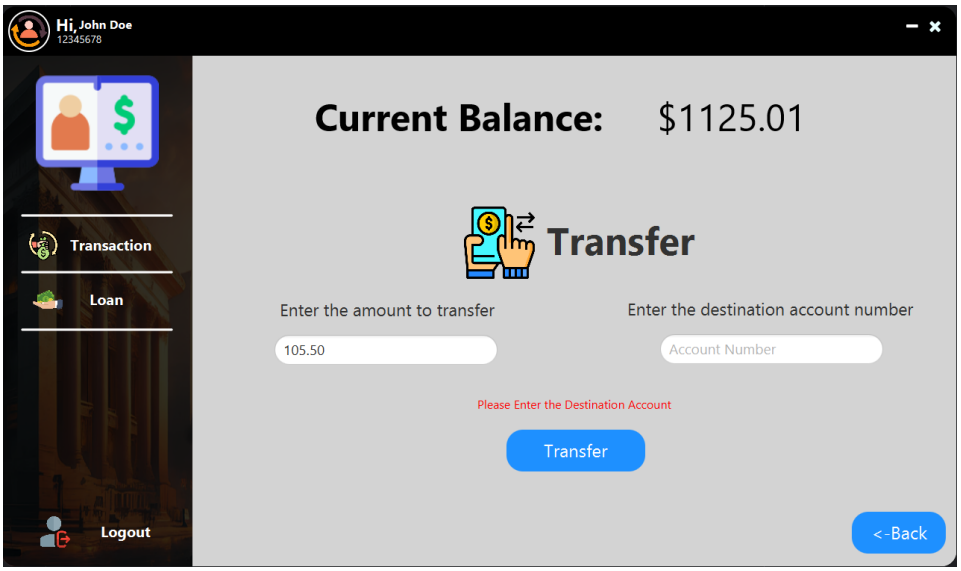


Figure 3.20: "Please Enter the Destination Account"

If you press Transfer button after writing Destination Account without entering an Amount an Error Message appears.

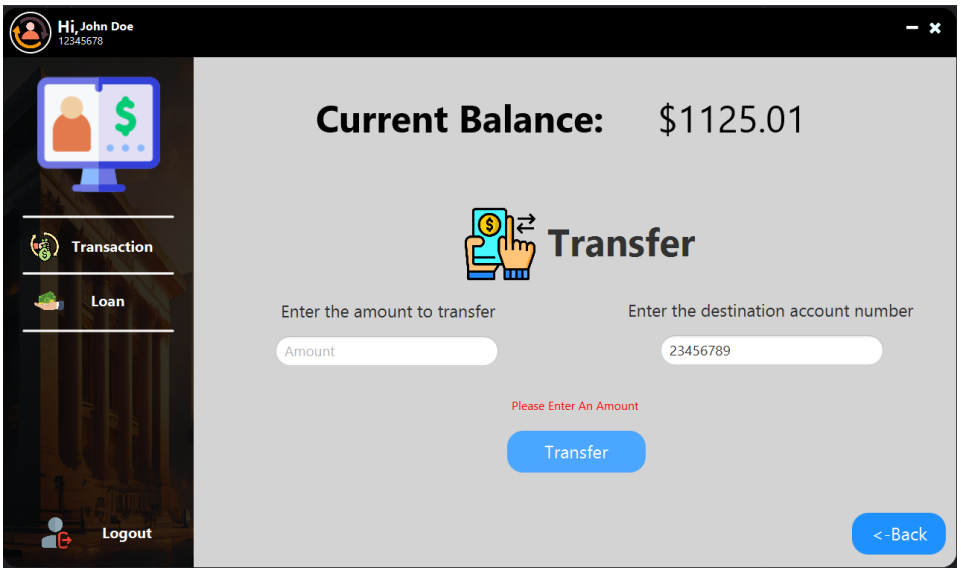


Figure 3.21: "Please Enter An Amount"

If you enter amount of 0 and you press Transfer an Error Message appears to tell you that the amount must be more than 0.

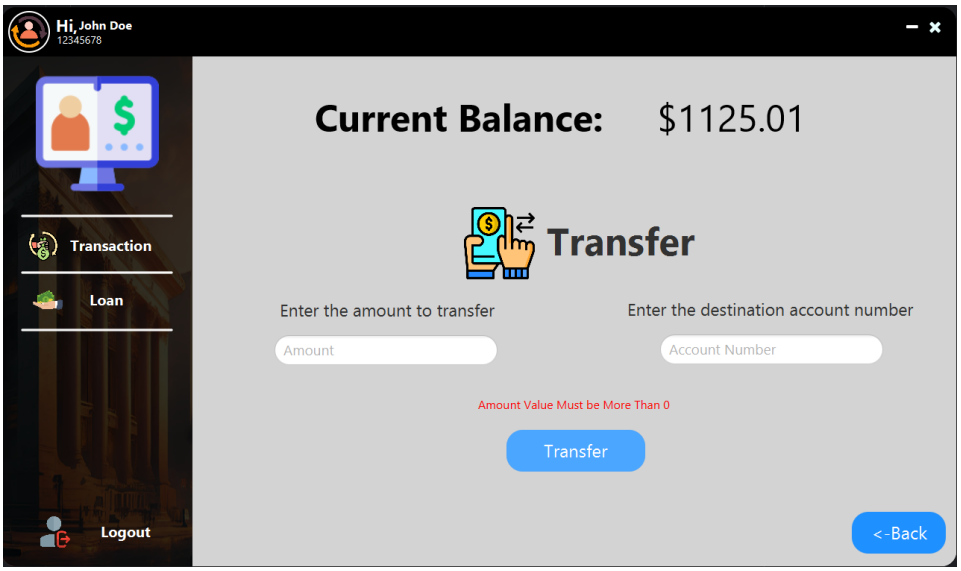


Figure 3.22: "Amount Value Must be More Than 0"

If you enter valid Amount (E.g 105.5) and Destination Account Successful Message Appears.

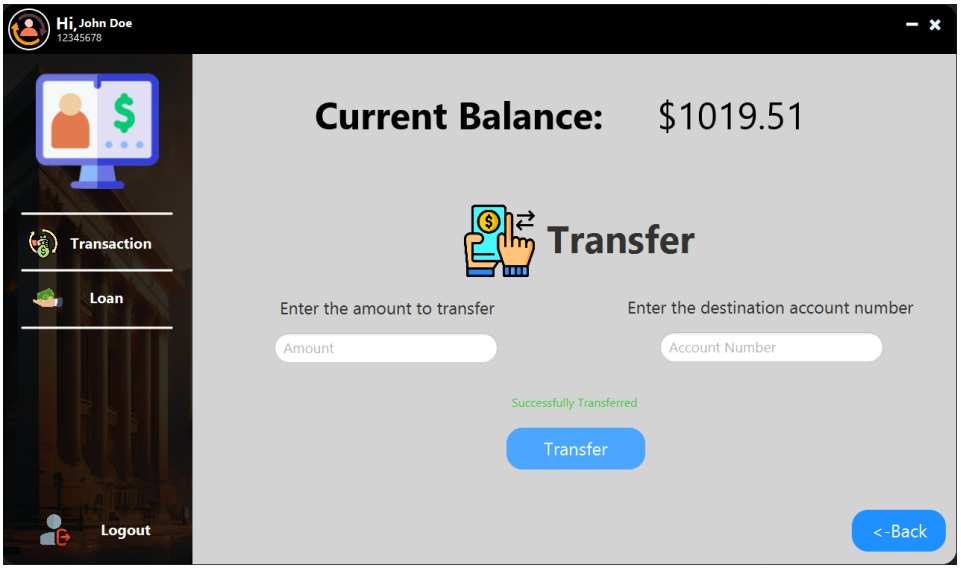


Figure 3.23: "Successfully Transferred"

3.3.7. Loan Page:

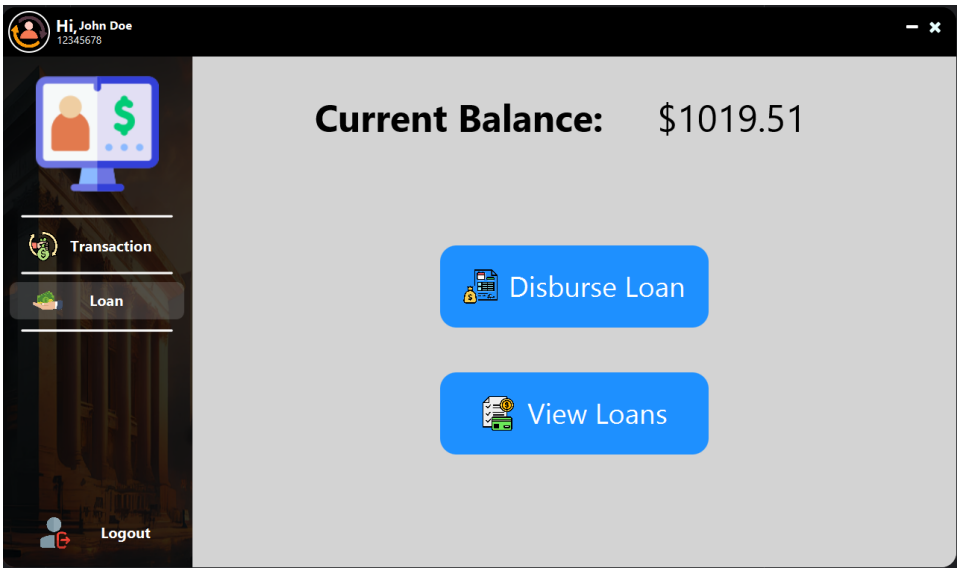


Figure 3.24: Loan Page

Here you can choose to go to to Disburse Loan page ,View Loans Page by pressing on the buttons containing their names. **See These Different Pages Below**

3.3.8. Disburse Loan Page:

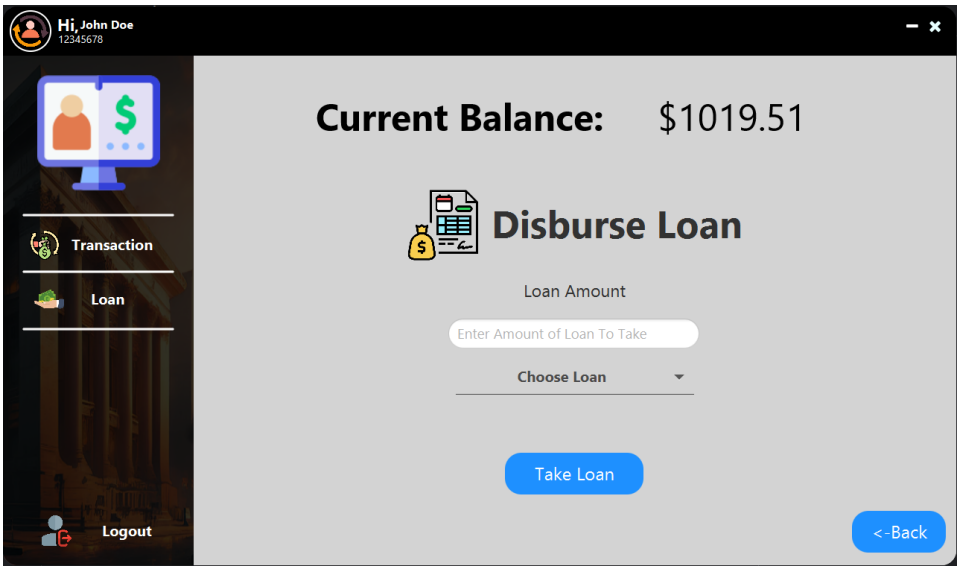


Figure 3.25: Disburse Loan

If you press Take Loan Button without entering an amount and without choosing a loan type Error Message appears.

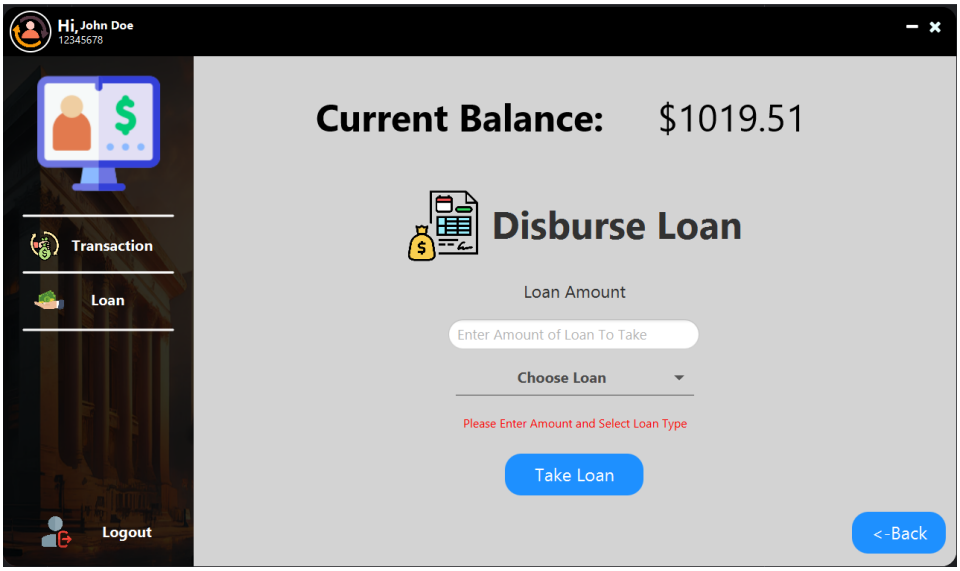


Figure 3.26: Please Enter Amount and Select Loan Type

If you press Take Loan Button after entering an amount without choosing a loan type Error Message appears.

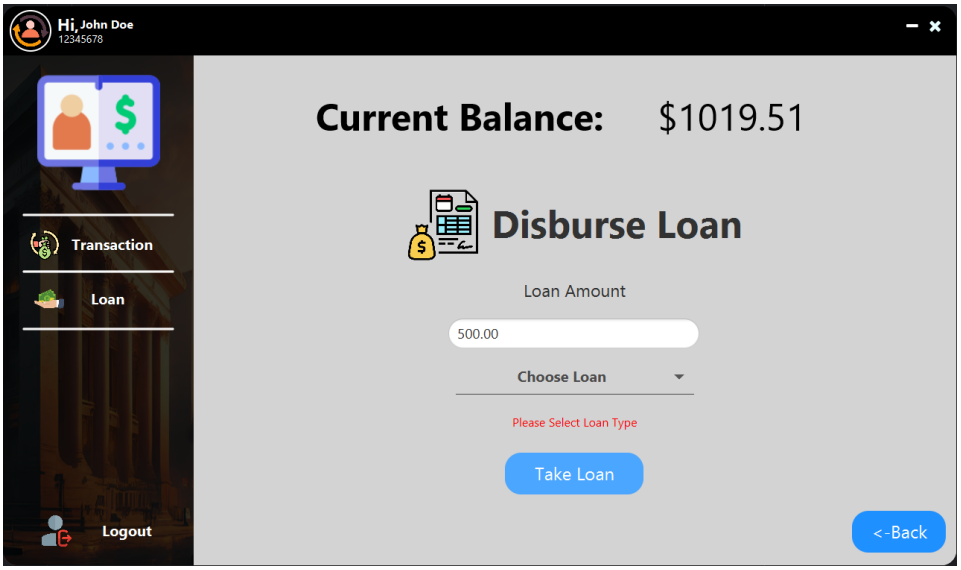


Figure 3.27: Please Select Loan Type

If you press Take Loan Button after choosing a loan type but not entering an amount Error Message appears.

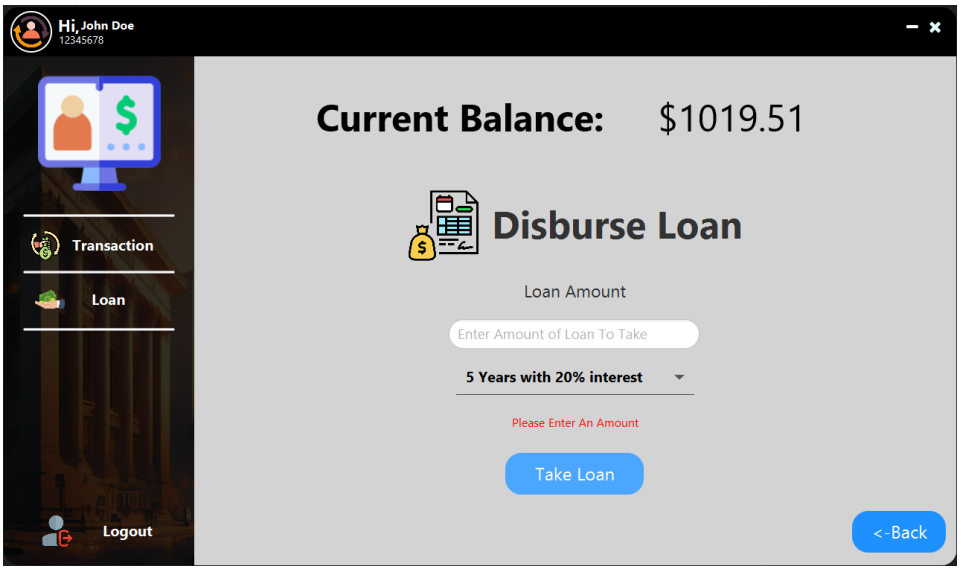


Figure 3.28: Please Enter Amount

If you enter amount of 0 and choose Loan Type then you press Take Loan an Error Message appears to tell you that the amount must be more than 0.

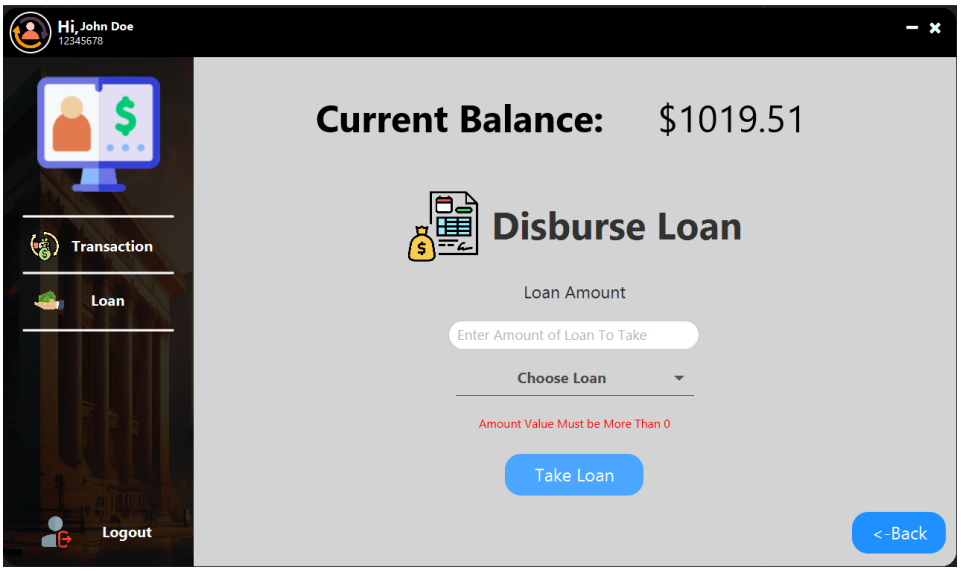


Figure 3.29: Amount Value Must be More Than 0

If you enter a valid Amount (E.g 500) and choose Loan Type Successful Message Appears.

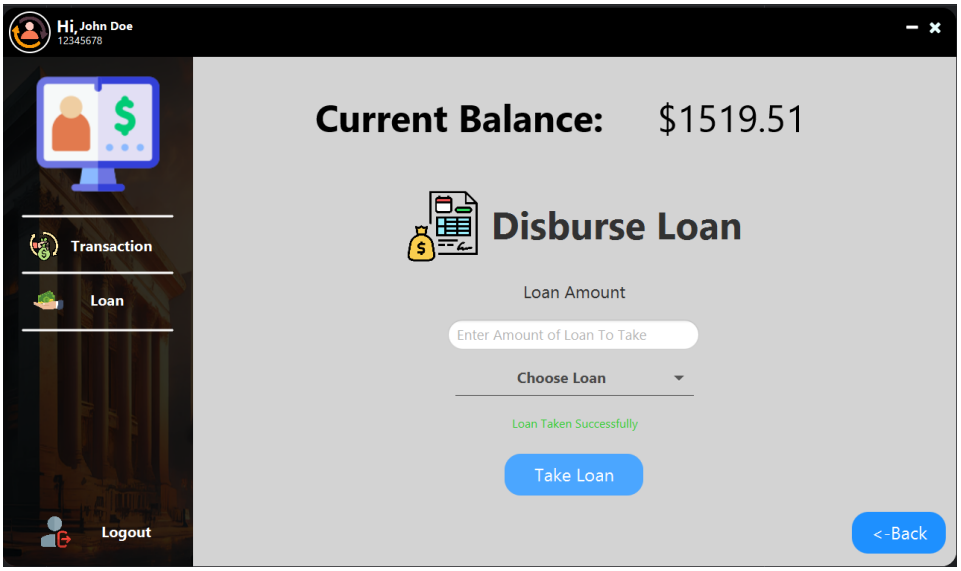


Figure 3.30: Successfully Taken

3.3.9. View Loans Page:

If you press Back and go to View Loans Page you will see all your disbursed loans.

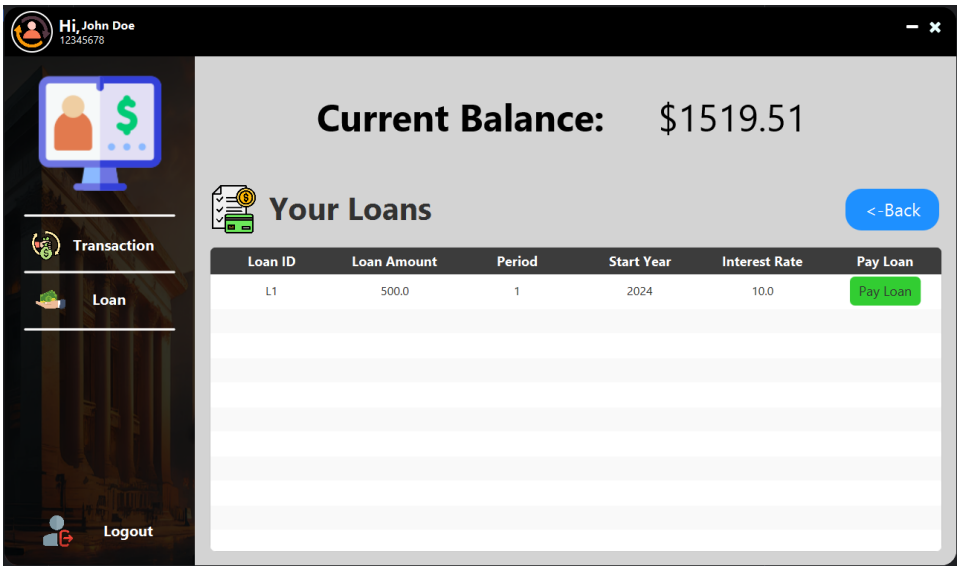


Figure 3.31: View Loans

If you press Pay Loan and you have enough balance to pay the loan and you didn't exceed the loan period Successful Message Appears.

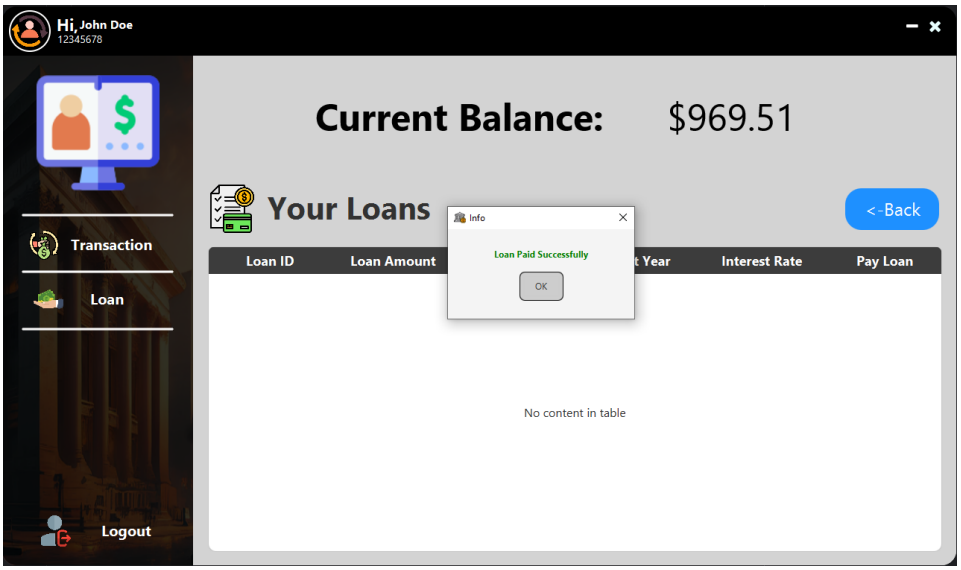


Figure 3.32: Loan Paid Successfully

If you try to pay the loan and you don't have enough balance to pay the loan Error Message Appears.

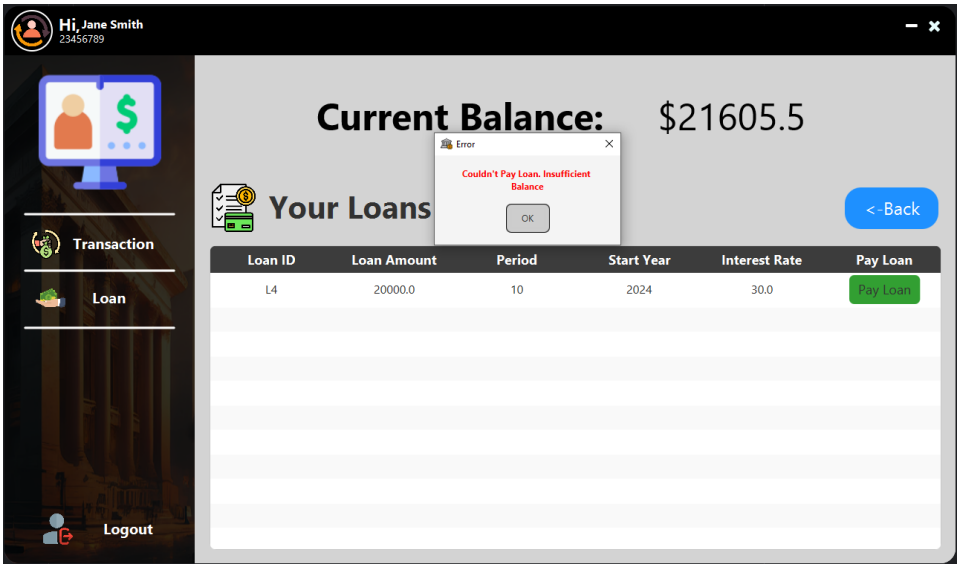


Figure 3.33: Couldn't Complete the Payment Insufficient Balance

3.3.10. Revisit Transactions (View Transactions Page):

If you press View Transactions Button View Transaction Page will open and you will see all your transactions.

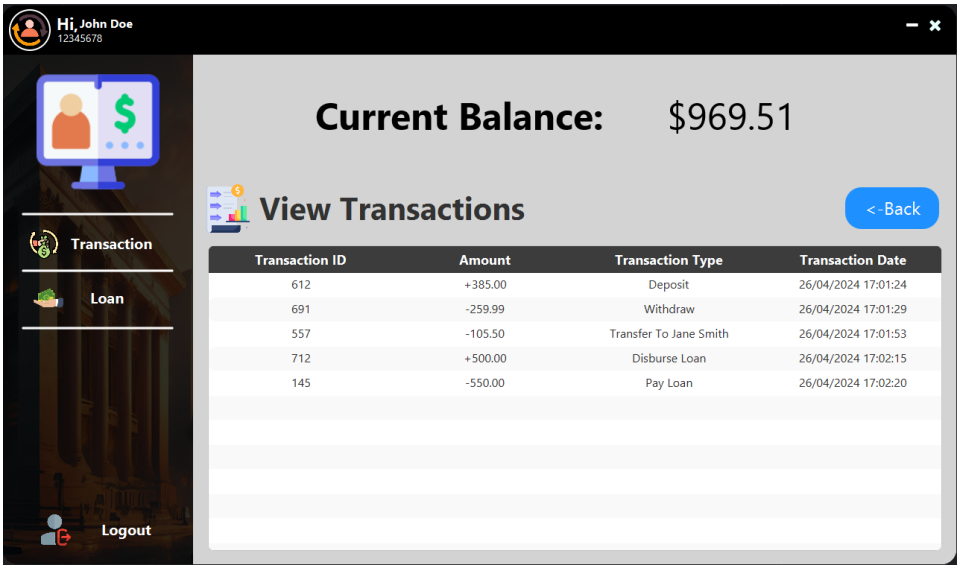


Figure 3.34: View Transactions Account 1

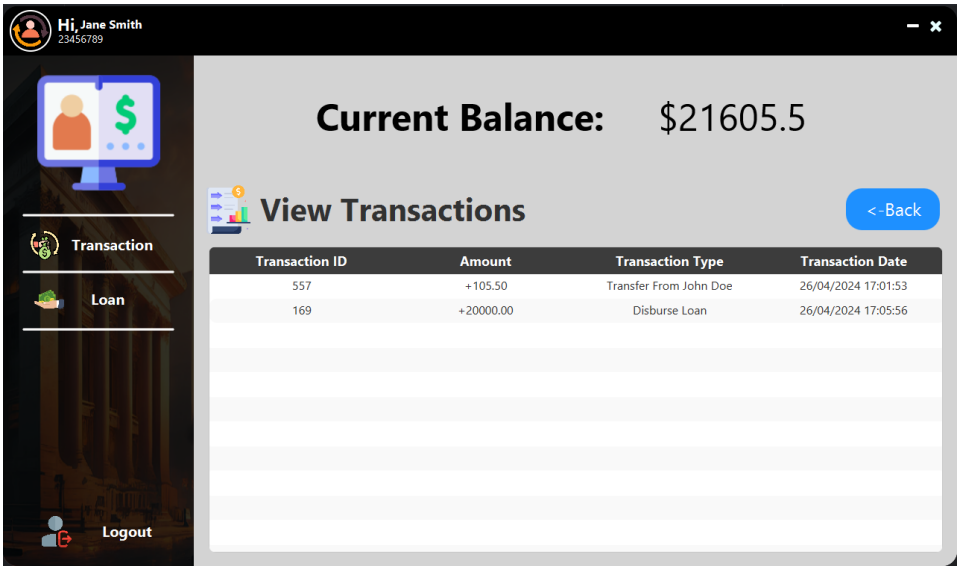


Figure 3.35: View Transactions Account 2

4

Requirement 3

4.1. White Box Testing

White box testing, also known as clear, glass box, or structural testing, is a method of testing software that involves detailed investigation of the internal logic and structure of the code. Unlike black box testing, which focuses on the input and output of the software system, white box testing requires knowledge of the internal paths, structures, and implementation of the software being tested. This type of testing is used to evaluate the flow of the program at various levels. It helps in validating the branches, loops, paths, and conditions of the software. White box testing is essential for optimizing the code and is integral in identifying every possible error in the software's logic.

4.1.1. Types of White Box Testing

Statement Coverage

Statement coverage testing aims to execute all the statements at least once in the source code. The goal is to ensure that all code paths are tested and that no part of the code is left unchecked.

4.1.1.1. Branch Coverage

Branch coverage, also known as decision coverage, involves executing each of the possible branches from each decision point at least once. This type of testing ensures that all branches in the code are tested, including true and false conditions.

4.1.1.2. Conditional Coverage

Conditional coverage testing ensures that each condition in a decision statement is evaluated both to true and false. It is more comprehensive than branch coverage because it considers the truth value of each condition separately, providing a more fine-grained assessment of the decision logic.

4.2. Function Code 1

```
1  public boolean processTransaction(double amount, String transactionType) {
2      if (transactionType.equals("D")) {
3          if (!(amount > 0.0)) {
4              return false;
5          }
6
7          this.balance += amount;
8          Bank.transactions.add(new Transaction(this, amount, this.formatter.
               format(this.date), transactionType));
9      } else if (transactionType.equals("DL")) {
10         if (!(amount > 0.0)) {
11             return false;
12         }
13
14         this.balance += amount;
15         Bank.transactions.add(new Transaction(this, amount, this.formatter.
               format(this.date), transactionType));
16     } else if (transactionType.equals("PL")) {
17         if (!(amount <= this.balance)) {
18             return false;
19         }
20
21         this.balance -= amount;
22         Bank.transactions.add(new Transaction(this, amount, this.formatter.
               format(this.date), transactionType));
23     } else {
24         if (!(amount <= this.balance)) {
25             return false;
26         }
27
28         this.balance -= amount;
29         Bank.transactions.add(new Transaction(this, amount, this.formatter.
               format(this.date), transactionType));
30     }
31     return true;
32 }
```

4.3. Types of White Box Testing Applied

4.3.1. Statement Coverage Testing

- Test Case 1: Deposit positive amount (Expected Output: true). Covers Statements: 1, 2, 3, 5, 6, 7, 8, 31.
- Test Case 2: Deposit zero amount (Expected Output: false). Covers Statements: 1, 2, 3, 4.
- Test Case 3: Payment less than balance (Expected Output: true). Covers Statements: 1, 2, 9, 16, 17, 19, 20, 21, 22, 31.
- Test Case 4: Payment equal to balance (Expected Output: true). Covers Statements: 1, 2, 9, 16, 17, 19, 20, 21, 22, 31.
- Test Case 5: Withdraw with amount less than or equal balance (Expected Output: true). Covers Statements: 1, 2, 9, 16, 23, 24, 26, 28, 29, 30, 31.
- Test Case 6: Withdraw with amount more than balance (Expected Output: false). Covers Statements: 1, 2, 9, 16, 23, 24, 25.

4.3.2. Branch Coverage Testing

In the following code snippet, after drawing the control flow diagram, we find that we have six possible paths that we are going to explain below.

- Test Case 1 (Path 1): Deposit with positive amount. (Covers successful deposit case), amount = 10, transactionType = D. Covers Statements: 1, 2, 3, 5, 6, 7, 8, 31.

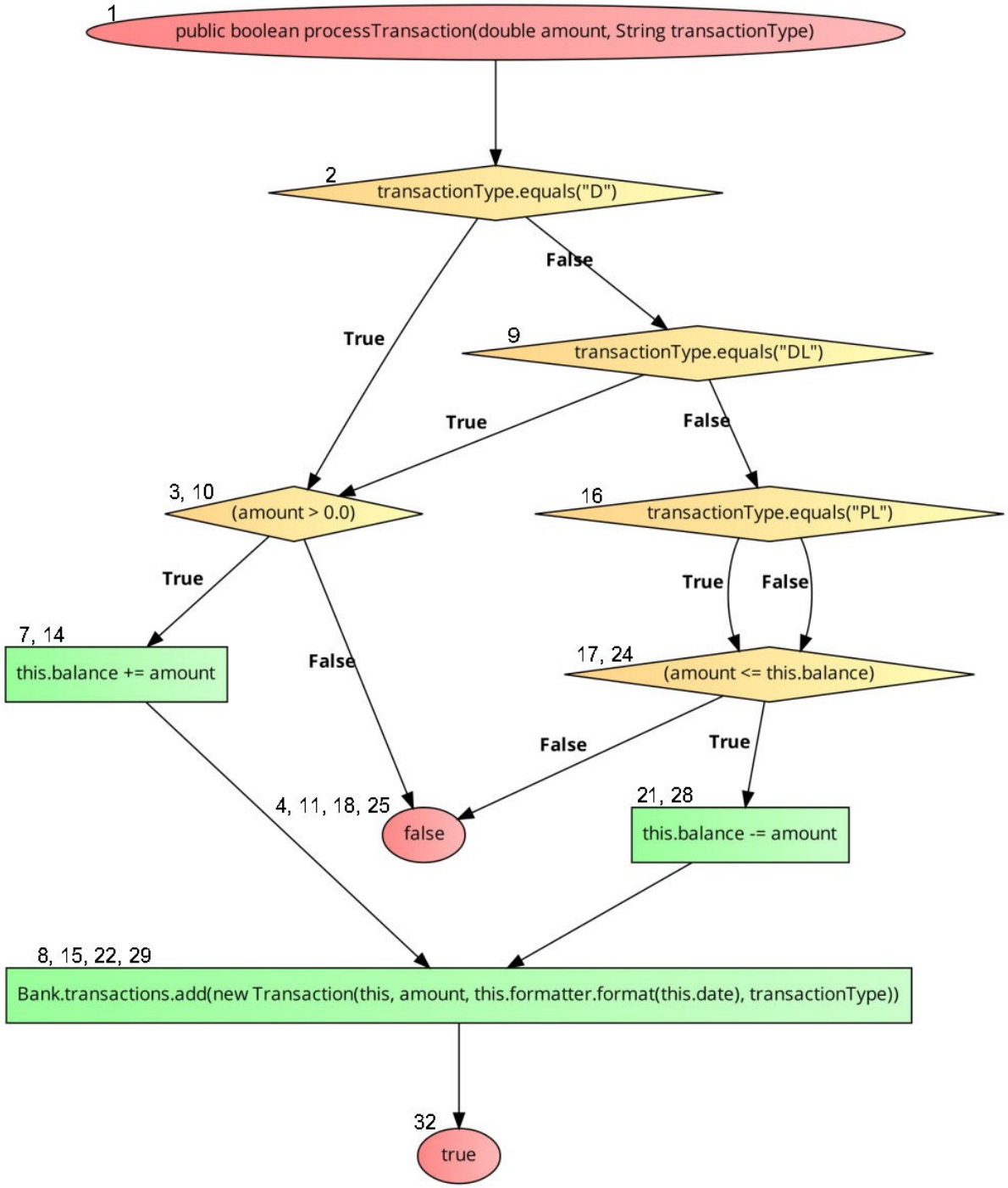
- Test Case 2 (Path 2): Deposit with negative amount. (Tests negative condition), amount = -10, transactionType = D. Covers Statements: 1, 2, 3, 4.
- Test Case 3 (Path 3): Disburse Loan with positive amount. (Covers successful disburse loan case), amount = 10, transactionType = DL. Covers Statements: 1, 2, 9, 10, 14, 15, 31.
- Test Case 4 (Path 4): Disburse Loan with negative amount. (Tests negative condition). Covers Statements: 1, 2, 9, 10, 11.
- Test Case 5 (Path 5): Pay Loan with amount less than or equal balance, Withdraw with amount less than or equal balance (Tests Success). Covers Statements: 1, 2, 9, 16, 17, 21, 22, 31 (in case of pay loan) or 1, 2, 9, 16, 23, 24, 28, 29, 31 (in case of withdraw).
- Test Case 6 (Path 6): Pay Loan with amount more than balance, Withdraw with amount more than balance (Tests Fail). Covers Statements: 1, 2, 9, 16, 17, 18 (in case of pay loan) or 1, 2, 9, 16, 23, 24, 25 (in case of withdraw).

Note that, although the statement numbers may differ in pay loan and withdraw, but they execute the same code logic and perform the same action, that's why we considered that they have the same path.

4.3.3. Conditional Coverage Testing

Given the following code, we need to add 8 test cases to cover the true and false cases of the conditions at least once, we will assume that balance in these test cases = 100:

- amount = 10, transactionType = D. Statements covered: 1, 2, 3, 7, 8, 31.
- amount = 10, transactionType = X. Statements covered: 1, 2, 9, 16, 23, 24, 28, 29, 31.
- amount = -10, transactionType = D. Statements covered: 1, 2, 3, 4.
- amount = 10, transactionType = DL. Statements covered: 1, 2, 9, 10, 14, 15, 31.
- amount = -10, transactionType = DL. Statements covered: 1, 2, 9, 10, 11.
- amount = 10, transactionType = PL. Statements covered: 1, 2, 9, 16, 17, 21, 22, 31.
- amount = 110, transactionType = PL. Statements covered: 1, 2, 9, 16, 17, 18.
- amount = 110, transactionType = X Statements covered: 1, 2, 9, 16, 23, 24, 25.


1
public boolean processTransaction(double amount, String transactionType)
2
transactionType.equals("D")
True
False
9
transactionType.equals("DL")
True
False
3, 10
(amount > 0.0)
16
transactionType.equals("PL")
True
False
True
False
7, 14
this.balance += amount
False
17, 24
(amount <= this.balance)
False
True
4, 11, 18, 25
false
21, 28
this.balance -= amount
8, 15, 22, 29
Bank.transactions.add(new Transaction(this, amount, this.formatter.format(this.date), transactionType))
32
true

4.4. Function Code 2

```
    public String takeLoan(String loanId, double loanAmount) {
        for (int i = 0; i < Bank.loans.size(); i++) {
            if (Bank.loans.get(i).getLoanId().equals(loanId)) {
                Bank.loans.get(i).setLoanAccount(this);
                Bank.loans.get(i).setLoanAmount(loanAmount);
                String msg = Bank.loans.get(i).disburseLoan();
                if (msg == null)
                    takenLoans.add(Bank.loans.get(i));
                return msg;
            }
        }
        return "Loan Not Found";
    }
```

4.5. Types of White Box Testing Applied

4.5.1. Statement Coverage Testing

To ensure that all executable statements in the code are executed at least once:

- Test Case 1: Loan ID exists (Expected Output: the return of `disburseLoan` if successful). Covers Statements: 1, 2, 3, 4, 5, 6, 7, 8, 9.
- Test Case 2: Loan ID does not exist (Expected Output: "Loan Not Found"). Covers Statements: 1, 2, 3, 12.

4.5.2. Branch Coverage Testing

To cover each possible branch of the code:

- Test Case 1: Loan ID matches and `disburseLoan` returns non-null (Expected Output: the return value of `disburseLoan`). Covers Statements: 1, 2, 3, 4, 5, 6, 7, 9.
- Test Case 2: Loan ID matches and `disburseLoan` returns null (Expected Output: null, and the loan is added to `takenLoans`). Covers Statements: 1, 2, 3, 4, 5, 6, 7, 8, 9.
- Test Case 3: Loan ID does not match any record (Expected Output: "Loan Not Found"). Covers Statements: 1, 2, 3, 12.

4.5.3. Conditional Coverage Testing

Ensuring that both true and false cases for each condition in the code are tested at least once:

- Test Case 1: Loan ID matches, `disburseLoan` returns non-null (Expected Output: the return value of `disburseLoan`). Covers Statements: 1, 2, 3, 4, 5, 6, 7, 9.
- Test Case 2: Loan ID matches, `disburseLoan` returns null (Expected Output: null, and the loan is added to `takenLoans`). Covers Statements: 1, 2, 3, 4, 5, 6, 7, 8, 9.
- Test Case 3: Loan ID does not exist (Expected Output: "Loan Not Found"). Covers Statements: 1, 2, 3, 12.

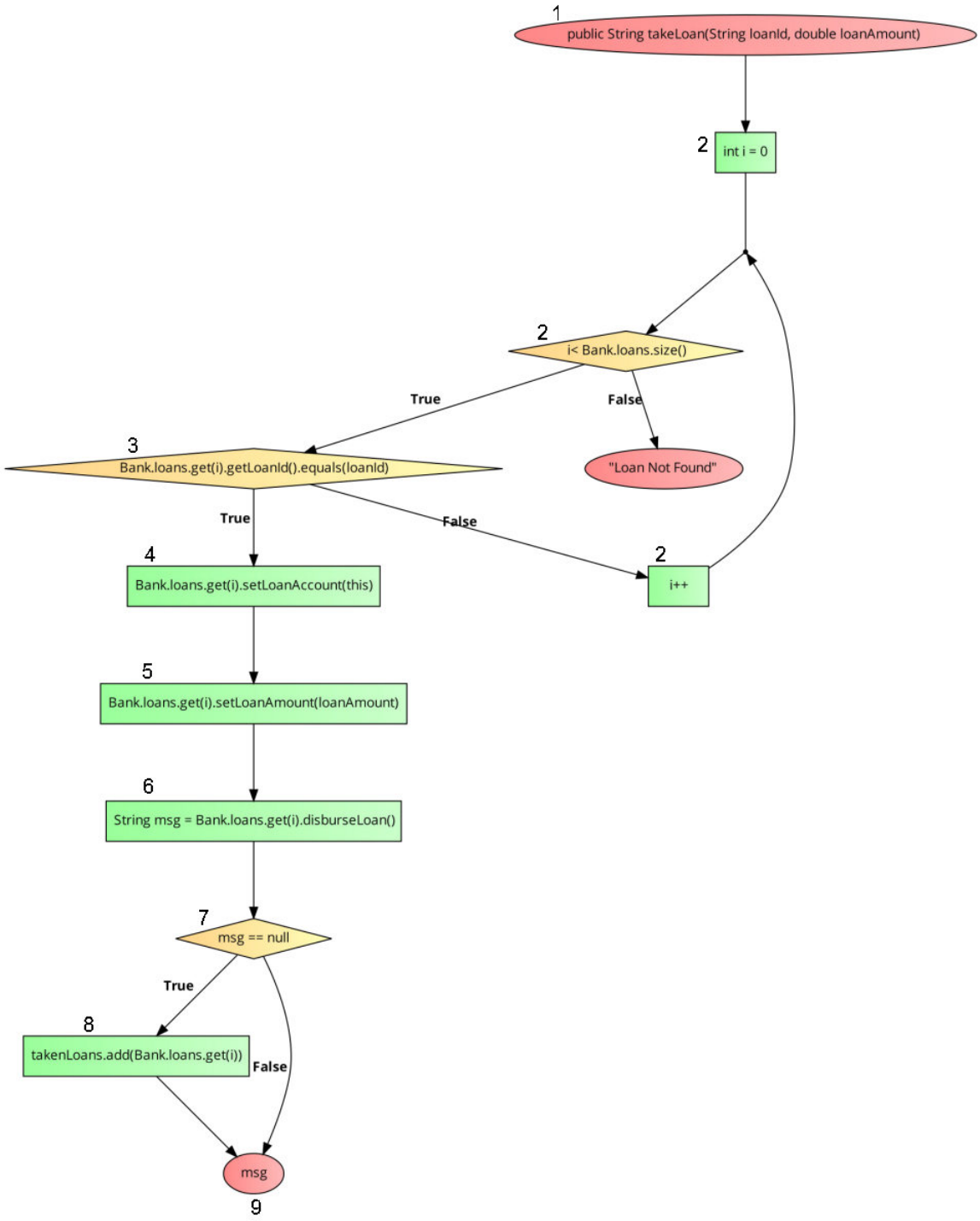


Figure 4.1: Control Flow Diagram

5

Requirement 4
Integration Testing

5.1. Methodology Comparison

Before detailing the integration strategy adopted for our banking system, a comprehensive comparison of different integration testing methodologies is necessary to justify the choice of the Top-Down Integration Testing approach over others like Big Bang, Bottom-Up, Sandwich, and Pairwise integration.

5.1.1. Top-Down Integration vs. Big Bang Integration

- **Top-Down Integration** provides early visibility into the system's functional flow by progressively integrating from the top-most to the lower-level modules. This method enables early prototyping and user interface testing.
- **Big Bang Integration** involves combining all components at once, which, while simple in smaller systems, poses significant challenges in isolating faults and debugging in complex systems such as ours where multiple modules like 'Account', 'Transaction', and 'Loan' are tightly interconnected.

5.1.2. Top-Down vs. Bottom-Up Integration

- **Bottom-Up Integration** tests from the lowest-level modules upwards. It requires drivers to test higher levels, potentially delaying the integration of critical business functionalities until later stages.
- **Top-Down Integration** allows for early testing of key functionalities and uses stubs to simulate lower-level modules, facilitating a more controlled and gradual integration.

5.1.3. Top-Down vs. Sandwich Integration

- **Sandwich Integration** combines Top-Down and Bottom-Up approaches, targeting both high-level and low-level module groups simultaneously. This can accelerate the integration process but requires significant coordination and resource allocation to manage two fronts of integration simultaneously.
- In our system, given the crucial need for early validation of high-level functionalities (e.g., overall account management and transaction flow), a pure Top-Down approach is preferred for its straightforward progression and simpler management.

5.1.4. Top-Down vs. Pairwise Integration

- **Pairwise Integration** focuses on integrating pairs of modules at a time, particularly useful for modules with significant direct interactions. This method can efficiently identify interface issues between closely coupled modules.
- For our banking system, while Pairwise could be effective for integrating modules like 'Transaction' and 'Account', a Top-Down approach ensures that the system's architecture and user interfaces are functional and robust from the outset.

Given the complexities and critical dependencies in our banking system, the Top-Down approach is more appropriate. It ensures that major functionalities are tested early, reducing integration risks by addressing issues in a systematic manner, and providing a clear path for integrating and testing lower-level functionalities through stubs.

5.2. Introduction

This Chapter outlines the integration strategy for the banking system, which includes the 'Bank', 'Account', 'Transaction', and 'Loan' modules, using a Top-Down Integration Testing approach. This strategy is designed to test high-level functionalities before progressing to more detailed, lower-level operations, ensuring a thorough and systematic integration.

5.3. Methodology

5.3.1. Top-Down Integration Testing

This method involves starting integration from the highest-level module and progressively integrating each subsequent module. It prioritizes early testing of major functionalities, utilizing stubs and drivers to simulate missing components.

5.3.2. Use of Stubs and Drivers

Stubs simulate lower-level modules by providing predefined responses, facilitating testing without the complete system. Drivers simulate higher-level module functionalities to test lower modules independently.

5.4. Integration Process

The process is detailed as follows:

1. **Initial Setup and Integration of the Bank Module**:
    - Develop a driver to simulate user inputs for account management and transaction initiation.
    - Integrate the 'Bank' module with a basic UI to handle user commands and display outputs.
    - Focus on ensuring the 'Bank' module can accurately direct commands to the appropriate modules and handle errors gracefully.
2. **Integration of the Account Module**:
    - Utilize a stub for the 'Transaction' module, which returns predefined results for transaction requests.
    - Test account creation, deletion, and updating functionalities thoroughly to ensure data integrity.
    - Validate the integration with the 'Bank' module by simulating scenarios like creating multiple accounts and handling concurrent access errors.
3. **Detailed Transaction Capabilities Integration**:
    - Replace the 'Transaction' stub with the real module.
    - Implement comprehensive tests for deposits, withdrawals, and transfers, ensuring transactions correctly affect account balances.
    - Test edge cases such as invalid transaction amounts and unauthorized transaction attempts to ensure robustness.
4. **Comprehensive Integration of the Loan Module**:
    - **Account Interaction:** Verify the integration with the 'Account' module by testing loan disbursements. Ensure that when a loan is disbursed, the corresponding account balance is updated accurately and the transaction is recorded in the account history.
    - **Transaction Logging:** Confirm that all loan-related activities, including disbursements and repayments, are logged as transactions within the 'Transaction' module. This integration is crucial for maintaining accurate and comprehensive financial records.

- **Stress Testing and Impact Analysis:** Conduct stress tests on loan repayments to ensure robustness under high-load scenarios by doing successive or simultaneously repayment of loan. Additionally, check the accuracy of how loan transactions affect customer account statements and overall financial status.

5. **Final System-Wide Integration and Testing**:

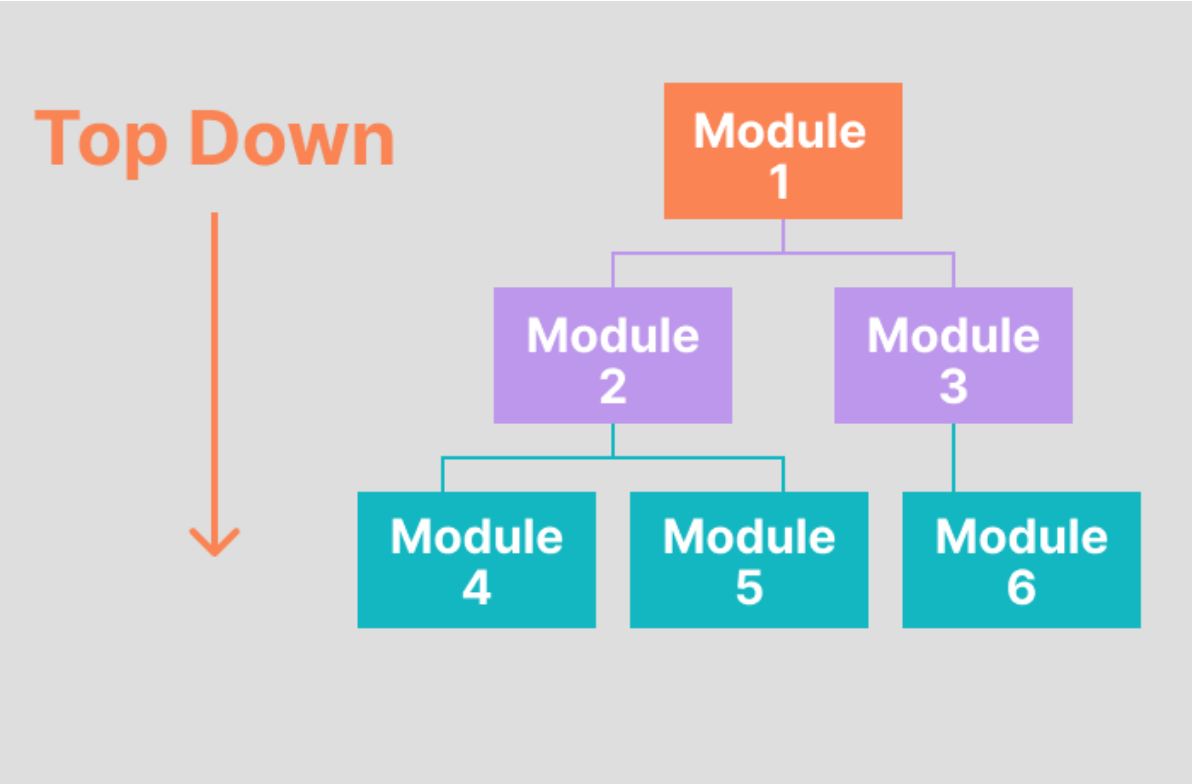
   - With all individual modules integrated, execute end-to-end system tests.
   - Test comprehensive scenarios involving all modules, such as a customer taking a loan and then performing various transactions.
   - Evaluate the system's ability to handle error states, rollback transactions, and maintain data consistency across all operations.

5.5. Challenges and Solutions

Significant challenges included managing complex dependencies between modules and ensuring consistent data across concurrent operations. Solutions involved refining the design of interfaces and improving the robustness of error handling mechanisms.

5.6. Conclusion

The Top-Down Integration Testing strategy proved highly effective for the systematic integration of the banking system's modules. This approach not only facilitated the early detection of integration issues but also ensured that high-level functionalities were robust before proceeding to more detailed integration tasks. Importantly, in the Top-Down approach, the number of stubs used is typically larger than the number of drivers. This is because each lower-level module or functionality is initially simulated by stubs as integration proceeds from the top. The extensive use of stubs supports the incremental and controlled testing of each module, contributing significantly to the smooth progression and stability of the overall integration process.



6

Conclusion

6.1. Conclusion

In this project, we developed a robust Java-based banking system designed to manage financial transactions, loans, and user accounts efficiently. The core of the system is encapsulated in several classes namely `Bank`, `Account`, `Transaction`, and `Loan`, each responsible for handling specific aspects of banking operations.

6.1.1. Key Features

The system supports basic functionalities such as:

- Creating and managing user accounts.
- Processing transactions including deposits, withdrawals, and transfers.
- Managing loans including disbursing and repayment.

These features are implemented with attention to real-world banking operations, such as transaction validation, balance checks, and date management using the Java `SimpleDateFormat` and `Year` classes.

6.1.2. Technological Overview

The implementation uses Java, a choice driven by its robust API, strong memory management, and extensive community support. We utilized the `ArrayList` from the Java Collections Framework to manage dynamic data collections, which allows our banking system to scale with an increasing number of transactions and users.

6.1.3. Challenges and Improvements

During the development, challenges such as error handling and data validation were addressed using Java's exception handling mechanisms. However, for future iterations, the application could be enhanced with a more granular exception handling strategy to provide more specific error messages.

6.1.4. Future Scope

Further development can introduce features like:

- Integration with real-time banking APIs for actual transactions.
- Extended user interface options, potentially through a Java-based GUI or a web interface.
- More complex financial products and services like mutual funds, insurance policies, and investment portfolios.

6.1.5. Conclusion

This Java banking system serves as a foundational platform for further development of a comprehensive banking application. It demonstrates the practical application of object-oriented principles and Java programming capabilities in solving real-world problems and providing a scalable solution for financial management.