Sanasa Society Bank System

CO328 Software Engineering Final Project Report

Group Member
E/15/010
E/15/265
E/15/347

Content

1. Abstract

"Sanasa Society Bank System" is a software solution for managing members and money transactions of sanasa society. Banking system of Sanasa society is a little bit different from other banks.
There are 2 main reasons for that.

1. Sanasa branches operate as independent entities without many interactions with other sanasa branches of the country.
2. Since its necessary to become a member to open a bank account, its mandatory to manage membership accounts along with the bank accounts in sanasa society.
   (In a normal bank, the user detail are directly included in his bank account. Here, a bank account is an extend feature of the member account)

The software system addresses 4 key components of the sanasa society.

- Manage Members
- Manage Deposits
- Manage Loans
- Manage Documentations (Audit reports, attendance reports, etc.)

There are 3 user types for the system

- Administrators : who can create and delete accounts records
- Accountants : who can manage transactions of the accounts
- Audits : who can check reports generated by the systems

The SSBS aims at simplifying the processes of the society that are carried out manually by implementing a digital structure.

2. Introduction

"Sanasa" is the Sinhala acronym for the Movement of Thrift and Credit Co-Operative Societies in Sri Lanka. Sanasa is the only Micro Finance Cooperative network in Sri Lanka. It covers all provinces with 8424 primary societies. The membership of the movement consist of persons belonging to all races and religions numbering 805,000.

Purpose of "Sanasa" is mainly to develop the economy of rural areas by providing and relieving loan facilities under reducing balance method for the members and holding various types of deposit accounts with high valuable interests. In addition it helps to improve habit of children's savings. For that they are holding child saving accounts with high interests and gift awarding system.

Sanasa Society Bank System (SSBS) is a software application that runs along with a MySQL database. Its purpose is to facilitate an effective, efficient, interactive & convenient environment for Sanasa employees of "Sanasa" bank. Over the years the recordings of transactions, member details & even issuing of receipts have been carried out manually. Tabulation of handwritten data is a time consuming task and it's very common to cause errors. SSBS aims at overcoming those traditional issues with modern technology available at present.

SSBS provides a Graphical User Interface (GUI) for the user that allows them to complete a variety of tasks from account creation, transaction handling to even generate audit reports. The GUI and the functionalities available for users differ according to the level of authority.

Most important thing about this system is that it can be modified according to the needs of the Bank's requirement. The system can be further extended or simplified depending on the situation. The architecture of the SSBS gives the versatility for the system to change and adopt for multiple banking scenarios. Simply, SSBS can be extend even out from the scope of Sanasa bank to any other local banking system.

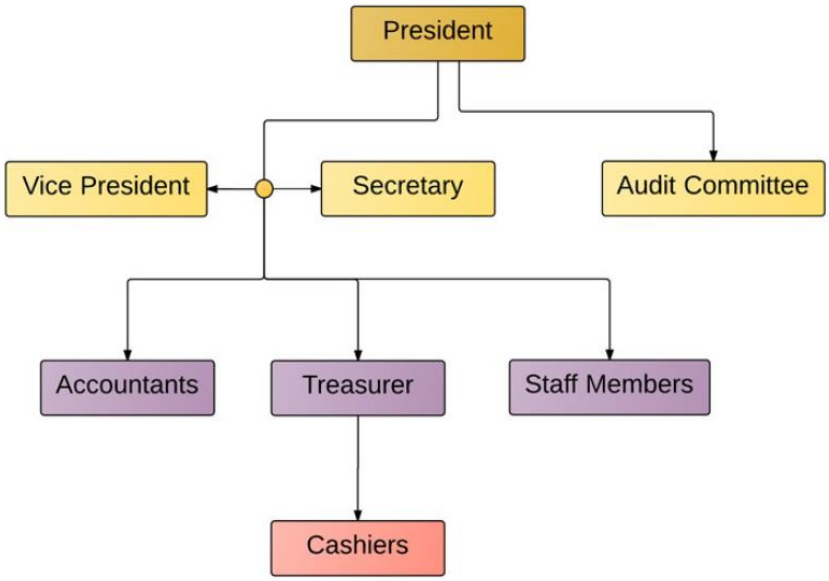
2.1.Background of the customer

- Name : Weedagama SANASA Society
- Address : Weedagama SANASA Society, Lake Road, Weedagama, Bandaragama.
- Tel : 038 5684168
- Reg. No : WP/k875
- Reg. Date : 28-09-1987
- Legal Form : The society is registered under Co-operative society ordinance of 1972 and Governed on Co-operative principals

2.2. Types of SANASA Clientele

- Farmers
- Self-employers
- Small employers
- Small business owners
- Fisherman
- Labors
- Vendor

2.3. Organizational Structure



Position	Service
President	This is the major position of the society. MI decisions are approved with the permission of this person.
Vice President	Involve in the duties of the president and help to make decisions.
Secretary	Make monthly minutes and responsible for all documentary records of the society.
Treasurer	Responsible for all financial dealings.
Accountants	Balance all accounts, calculate Interests, create monthly and annual cash summery report and responsible for annual internal audit report.
Cashiers	Interact with members of the society. Receive or issue money for members and record daily transactions and submit a daily report.
Staff Members	Active Members of the society. Attend Congress meeting of the society. Vote for amendment of constitution. Contribute for the discussion of new plans of the society.
Audit Committee	Analyze all transactions of the society and check Cashiers and Accountants behaviors

2.4. Objective

Mainly there are 3 objectives for this banking system.

- Implement digital methods to transaction activities that are carry out manually.
- Develop a system that efficiently store, update and retrieve data.
- Create a user friendly Interface for the employees to work with.

Each of these objectives have multiple number of sub goals. When implementing digital methods, the system aims to digitalize the activities such as manage transactions, record keeping, issuing and receiving of receipts, etc.

When developing an efficient system, the key areas that needed to be addressed are viewing account details, retrieving complex data queries, and accessing, searching large datasets in order to retrieve or edit the data.

In order to create a user-friendly Interface, a very simple and attractive model has been created for the system. The main goal here is to make the system as easy to handle with very little computer literacy.

2.5. Advantages

Managing Members

- The system can easily add someone as a Non-member or a child member at the first in same way. It decides automatically that he/she is adult or not. In addition, it keeps details about society member applicants or not from the Non-member list. After it checks automatically adult member's attendance for monthly congress meeting and verify him/her to society member after 3 months.

- It's easy to mark attendance using member id or member name.
    - System can search the attendance of each member of the given period or attendance of all members in a given date of monthly congress meeting. Manually it takes more time to search attendance and have to calculate attendance one by one when issuing a loan.
    - But sometimes monthly congress meeting doesn't hold regularly cause some unavoidable incidents.
    - So system keeps each meeting's dates for accuracy of analyzing member's attendance.

- This system is most important for managing loans and deposits.
    - When issuing a receipt or paying-in-slip, that transaction is added to the relevant documents automatically.
    - So it is no need to prepare Ledgers, Cash books or Balance sheets one by one.

- All the interests can be calculated automatically through the system and can be added into accounts to relevant rates. As well deposits and loans rates are often changing. So system works accurately above changes.

Managing Loans

- When we issuing a loan
- Easy to search loan history of the member
- Search whether the loan requirements are satisfied
- Fill Loan Application
- Select relevant loan module
- Issue Approved Loans

- We can create new loan modules under the congress meeting decisions. Thus system can edit loan modules and remove loan modules. But if we remove a Loan module that should be not used of someone for their loan transactions.

- According to the loan module it calculates interests and fines then shows automatically.
    - When someone neglect to pay installments then system sends notifications to user.

Managing Deposits

- Easy to make deposit and withdrawals.
- Easy to manage Normal deposits and fixed deposits of Members and Non-Members.
- As well easy to make compulsory deposits regularly. If some neglect to make compulsory deposit, then system sends a warning.
- When we issuing a loan, easy to check his/her deposits details.
- All the interest calculates automatically and added it into the account.
- A member can keep more than one fixed deposits for various time periods.
- Audit reports can be automatically calculated for given period and summery of a given ledger can be viewed for given date.

3. Requirement Analysis

3.1. Functional Requirements

Manage Members

- System should create 3 types of user accounts upon request.
    - Member
    - Non-Member
    - Child Member

- Child Member account can be opened by parents for their children. In addition, it has 2 types of following child accounts. The system should be able to create these two sub categorical accounts as well.
    - Guardian child deposit accounts
    - Self-child deposit accounts

- A Nonmember account should be able to convert into a member account (This should be with the approval from a current valid member - under the 1.6 and 1.7 sub constitution).

- In addition system should keep details about society member applicants and the Non-member applicants. Then the software should check the attendance of an adult member for monthly congress meeting for 3 months and depending on the result that member should be added to society member list.

- These account should keep record of the membership fees (Initially 300/=) other payments regarding society activity.

- Initially all the accounts should be created as Nonmember and then convert into member. (Someone likes to join the society as a member, first he has to join as a Non-Member depositor and should attend continuously for the monthly congress meeting within first 3 months. After Completing above process successfully he/she can get the membership of the society after receiving the approval from the member congress meeting.)

Manage Deposits

- System should be able to create following types of deposits and manage their transactions
    - Member and Non-Member normal deposits
    - Member and Non-Member fixed deposits
    - Compulsory deposits
    - Children's savings

- Regarding Deposit Management
    - It should be able to make deposits and withdrawals.
    - The system should manage Normal/Fixed deposits of Members and Non-Members.
    - Separately handle compulsory deposits and notify delayed deposits.
    - All the interests should be calculated and added to the account.
    - Should allow one member to handle more fixed deposits for various periods of time.

- Member and Non-Member Deposit Interest ratings are different for normal deposit and fixed deposit. (Generally, Member interest ratings are higher than Non-Member interest ratings.)

- Compulsory deposits include only for Society Members. It is Compulsory for members to pay that deposits account monthly to protect their membership.

Manage Loans

- There are some types of Loans defined by the society
  - Member Loans
  - Property Loans
  - Quick Loans
  - Fixed deposit basis Loans
  - Festival Loans
  - Short time loans

- Regarding Loan Management
  - This should be able to
    1. Easily search loan history of the member
    2. Search if the loan requirements are satisfied
    3. Ability to fill out the loan application
    4. Select relevant loan module
    5. Issue approved loan
  - Should be able to create new loan types and edit existing loan modules according to the congress meeting decisions.
  - Notifications must be shown for the delayed installments of each month.

- Most of these loan type interests are calculating under the reducing balance method.
- The limit of maximum personal loan amount depends on behavior history of the member for any Loan types without fixed deposit basis loans.

Manage Documentations

- Society issues following documentations for each transaction
  - Receipts
  - Paying-in slip
  - Voucher
  - Pass Book with transaction record

- In addition, they receive money deposited certificates for external bank transactions.

- They keep following records according to above documents
  - Cash Received Book
  - Cash Paid Book
  - Ledgers

- In addition, they maintain
  - Member Detail Book
  - Attendance Book
  - Loan application Book
  - Interest Calculation Book
  - Child Account Book

- At the end of each day, Clerk/Cashier should need to arrange a balance sheet.

- As well at the end of each financial year, society needs to be checked all the records and transactions again and prepare and Internal Audit Report. Including as following
  - Loan schedule
  - Deposit schedule
  - Monthly money summary

Managing User Levels

There are 3 basic user levels to handle the system.

- **Admin** : President, Vice President, Secretary, Treasurer.
- **Accountants** : Accountants, Cashiers, Staff Members.
- **Audit** : Audit from Same Branch, Audit from External Branch.

3.2. Non Functional Requirements

Hardware specification

- Processor : i5 Core Processor
- Clock speed : 2.5GHz
- Monitor : 1024 * 768 Resolution Color
- Keyboard : QWERTY
- RAM : 1 GB
- Input Output Console for interaction

Software specification

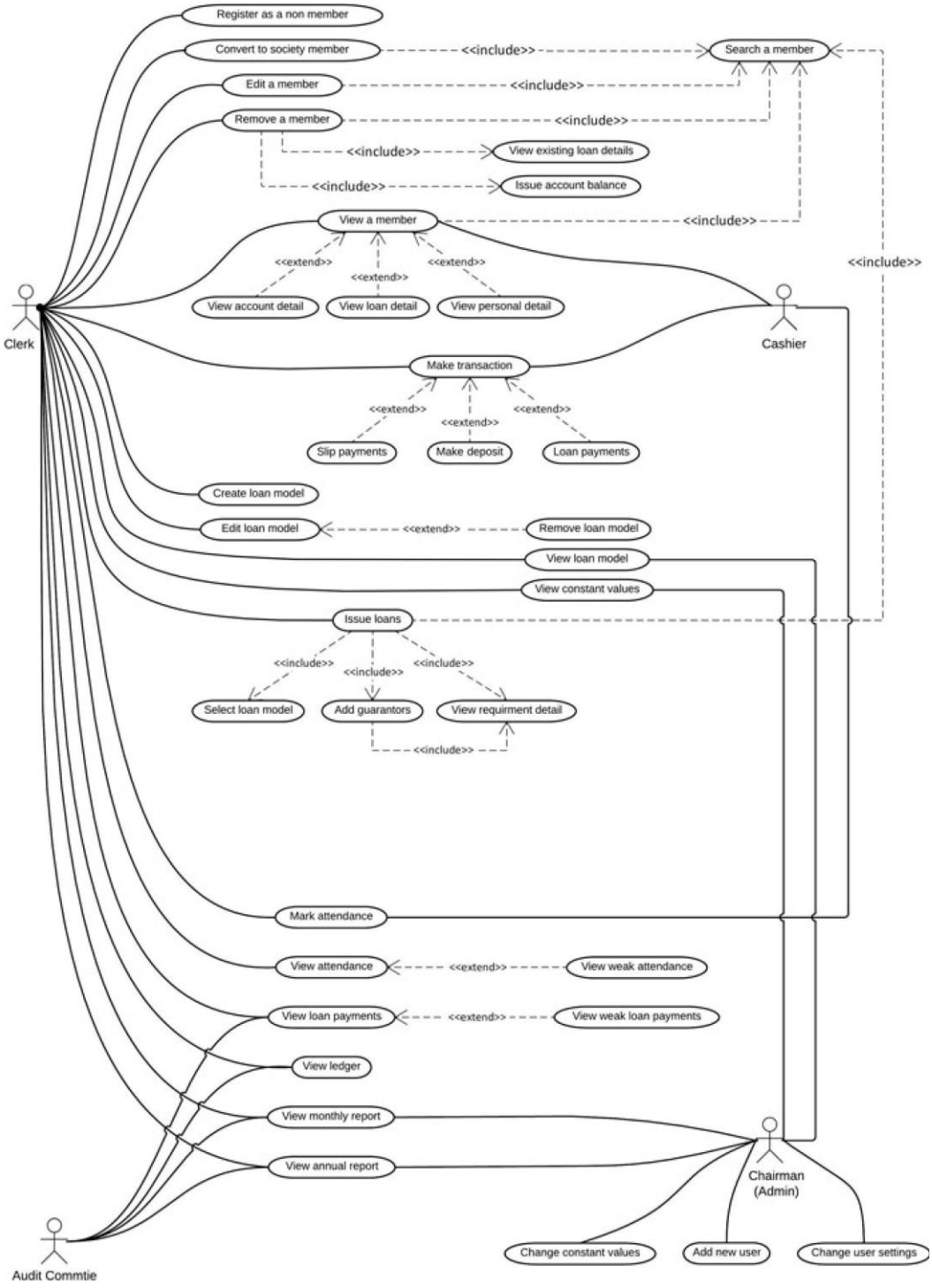
- MySQL Libraries
- MySQL Workbench 6.3 CE
- Eclipse IDE
- Apache Tomcat Server
- Operating system : Windows10

4. Design

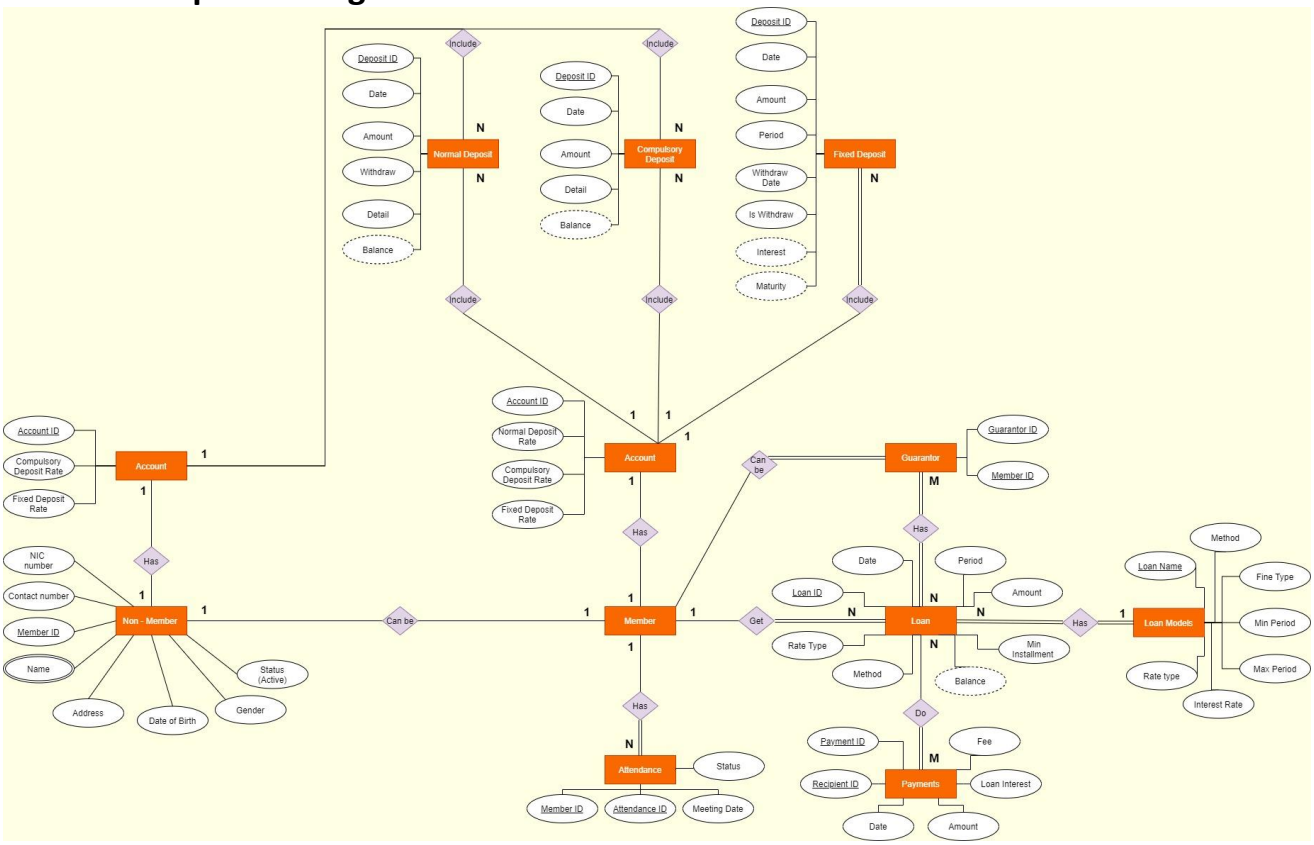
4.1. Design Diagrams

The Sanasa banking system have the capability of handling different type of processes. In here we have modeled the system scope in to a conceptual and physical model which make easy to identify about the system. With the conceptual model it shows the concepts behind the model with relations and entities and in the physical model it describes about the database-specific implementation of the data model which offers database abstraction and helps generate the schema. The Use Case diagram was used in describing the requirements of a system including internal and external influences with identifying actors. These models are modeled according to the information gathered from user requirements. This helps to visualize the schema behind the system.

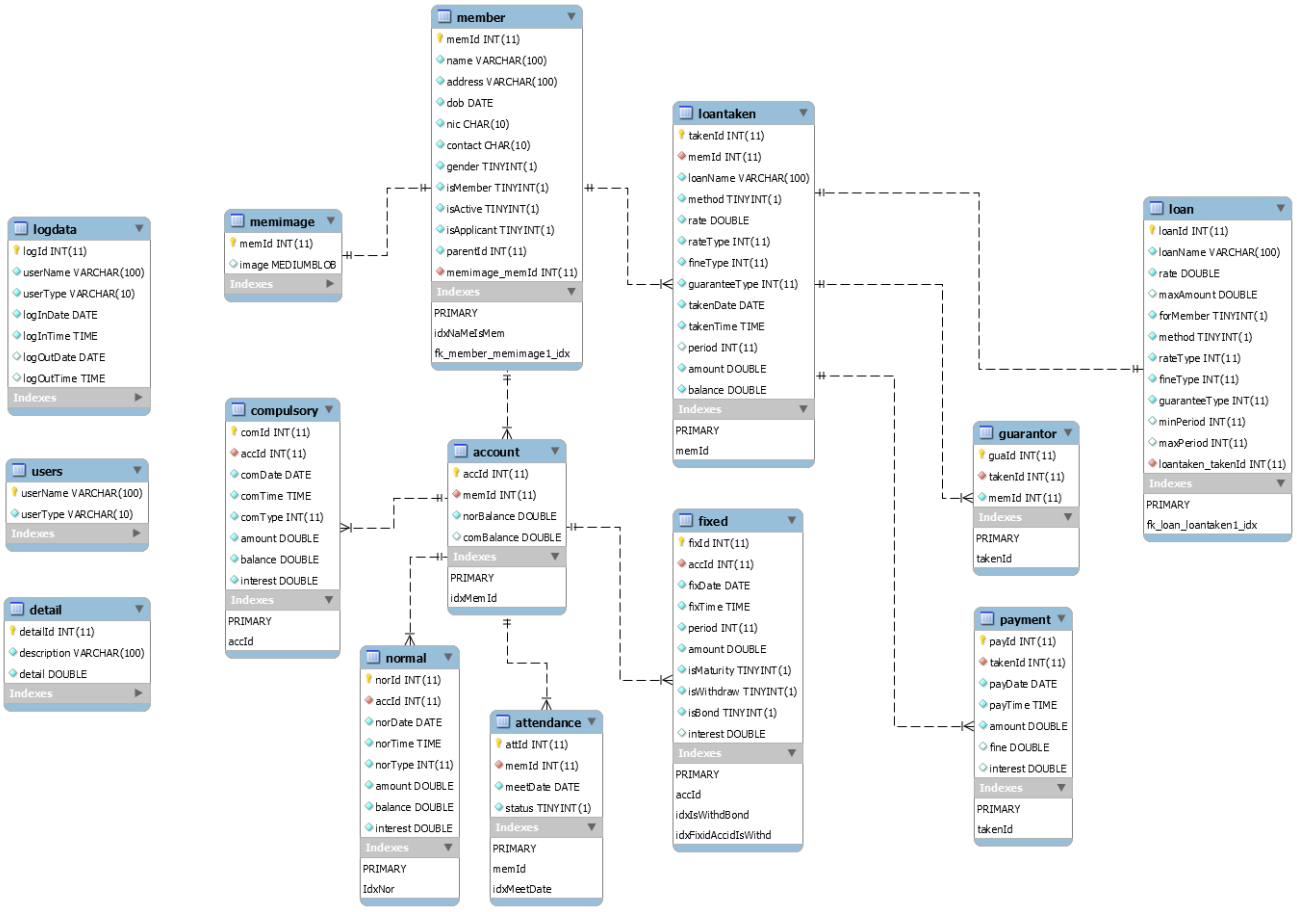
4.1.1. Use Case Diagram



4.1.2. Conceptual Design



4.1.3. Physical Model



4.2. Design Patterns

4.2.1. Singleton Design Pattern

Singleton pattern restricts an instantiation of a class from outside world and guarantees that there is only one instance of the class on the JVM at any given point of time. And it provides a global access point to get that single instance of the class. This pattern widely used for logging, drivers objects, caching and it also used for other design patterns implementations such as Builder, Facade etc.

In our project, the back end that makes connections to MySQL database. To Avoid Dirty connections across the database and java program we have used singleton pattern to ensure we have only one database connection available at any point of time. We did it because, database connections are a limited resource. Some DBs have a very low connection limit, and wasting connections is a major problem. By consuming many connections, you may be blocking others for using the database.

Implementation

There are several ways of implementing singleton design pattern such as Eager initialization Static block initialization, Lazy Initialization, Thread Safe Singleton etc. In our project we did something similar to Lazy Initialization. This method creates the instance in the global access method.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class DBConnection {
    private static DBConnection dbConnection;
    private Connection conn;

    private DBConnection(String username, String password ) throws ClassNotFoundException, SQLException{
        Class.forName("com.mysql.jdbc.Driver");
        if(conn != null) conn.close();
        conn = DriverManager.getConnection("jdbc:mysql://localhost/sanasa",username,password);
    }

    public static DBConnection createConnection() throws ClassNotFoundException, SQLException{
        return dbConnection;
    }

    public static DBConnection createConnection(String username, String password) throws ClassNotFoundException, SQLException{
        if(dbConnection == null){
            dbConnection = new DBConnection(username,password);
        }
        return dbConnection;
    }

    public Connection getConnection(){
        return conn;
    }

    public void closeConnection() throws SQLException{
        dbConnection = null;
    }
}
```

Code 1 - DBConnection.java

In our project which is a single-threaded environment, the above implementation works fine but when it comes to multi-threaded systems, it can cause problems if multiple threads are inside the “if condition” at the same time. This will break the singleton sequence, and the separate singleton class instances will be accessed from both threads.

And also we implemented singleton design inside the Factory design pattern too. Because there are 28 GUI controller classes. Inside all these classes, it might needs to create Model class objects and Database Controller class objects when it necessary. So avoid making Factory class instances everywhere we include this pattern to both “**ModelFactory**” and “**ControllerFactory**” classes.

However this patterns also has some drawbacks. If the object and the methods associated with it are so closely bound that it is difficult to check without writing a full-functional class dedicated to the Singleton, a Singleton may create difficulties writing testable code.

4.2.2. Factory Design Pattern

Factory method pattern enables to create an object without exposing the creation logic to the outside world and refer to the newly created object using a common interface. When we have a superclass with multiple sub classes we can return one of the sub-class based on a given input. This pattern takes out the responsibility of the instantiation of a class from the client program to the factory class.

Our Project has around 17 Model classes and 14 database controller classes. Creating objects from this classes all over the places in GUI controllers, the overall project code will become messy. That's why we implemented Factories called "**ModelFactory**" and "**ControllerFactory**" for both Model classes and database controller classes.

Implementation

To do that we extends all Model classes from a one Super class called "**SuperModel**" and used an enumeration class called "**ModelType**" for selecting Model type from the Factory class. Then we create a factory class and make it singleton. Then add a instance method called "getModel" which has a switch for selecting which Model class instance wants to user according to given ModelType enum.

```
public enum ModelType {
    ACCOUNT, ATTENDANCE, COMPULSORY, FIXED,
    GRAPH_MODEL, GUARANTOR, LOAN_MODEL,
    LOAN_TAKEN, LOG_IN, MEMBER, MEMBER_ACCOUNT,
    NEXT_PAYMENT, NORMAL, PAYMENT, PREVIOUS_PAYMENT,
    SHARES, USER ;
}
```

Code 2 - ModelType.java

```
import com.database.model.child.*;
import com.database.model.superb.SuperModel;
import com.manifest.ModelType;

public class ModelFactory {
    private static ModelFactory modelFactory;
    public static ModelFactory getInstance() {
        if (modelFactory == null) {
            modelFactory = new ModelFactory();
        }
        return modelFactory;
    }

    public SuperModel getModel(ModelType type) {
        switch (type) {
            case ACCOUNT:
                return new Account();
            case ATTENDANCE:
                return new Attendance();
            case COMPULSORY:
                return new Compulsory();
            case FIXED:
                return new Fixed();
            case GUARANTOR:
                return new Guarantor();
            case LOAN_MODEL:
                return new LoanModel();
            case LOAN_TAKEN:
                return new LoanTaken();
            case MEMBER:
                return new Member();
            case MEMBER_ACCOUNT:
                return new MemberAccount();
            case NEXT_PAYMENT:
                return new NextPayment();
            case NORMAL:
                return new Normal();
            case PAYMENT:
                return new Payment();
            case SHARES:
                return new Shares();
            case USER:
                return new User();
            default:
                return null;
        }
    }
}
```

Code 3 - ModelFactory.java

Above same procedure applied for database controller classes' factory implementation.

```
public enum ControllerType {
    ACCOUNT, ATTENDANCE, COMPULSORY,
    DETAIL, FIXED, GUARANTOR, LOAN_MODEL,
    LOAN_TAKEN, LOG_IN, MEMBER, NORMAL, PAYMENT,
    SHARES, USER ;
}
```

Code 4 - ControllerType.java

```
import com.database.ctrl.child.*;
import com.database.ctrl.superb.SuperController;
import com.manifest.ControllerType;

public class ControllerFactory {
    private static ControllerFactory controllerFactory;
    private ControllerFactory(){}

    public static ControllerFactory getInstance() {
        if (controllerFactory == null) {
            controllerFactory = new ControllerFactory();
        }
        return controllerFactory;
    }

    public SuperController getModel(ControllerType type) {
        switch (type) {
            case ACCOUNT:
                return new AccountDBController();
            case ATTENDANCE:
                return new AttendanceDBController();
            case COMPULSORY:
                return new CompulsoryDBController();
            case DETAIL:
                return new DetailDBController();
            case FIXED:
                return new FixedDBController();
            case GUARANTOR:
                return new GuarantorDBController();
            case LOAN_MODEL:
                return new LoanModelDBController();
            case LOAN_TAKEN:
                return new LoanTakenDBController();
            case LOG_IN:
                return new LogInDBController();
            case MEMBER:
                return new MemberDBController();
            case NORMAL:
                return new NormalDBController();
            case PAYMENT:
                return new PaymentDBController();
            case SHARES:
                return new SharesDBController();
            case USER:
                return new UserDBController();
            default:
                return null;
        }
    }
}
```

Code 5 - ControllerFactory.java

5. Implementation

This software is implemented with a graphical user interface that was built using JavaFX that runs on top a MySQL database. The following concepts are used to implement different structure and tables for the system in order to work as a standalone application.

5.1. Implementation of User Interface

The user interfaces of the software is designed with .fxml format. A basic example is as follows

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.Button?>
<?import javafx.scene.control.Label?>
<?import javafx.scene.control.PasswordField?>
<?import javafx.scene.control.TextField?>
<?import javafx.scene.image.Image?>
<?import javafx.scene.image.ImageView?>
<?import javafx.scene.layout.AnchorPane?>
<?import javafx.scene.layout.HBox?>
<?import javafx.scene.text.Font?>
<?import javafx.scene.text.Text?>

<AnchorPane id="AnchorPane" fx:id="loginPanel" prefHeight="300.0" prefWidth="500.0" style="-fx-background-color: rgb(43,87,154);" stylesheets="@../style/login.css"
xmlns="http://javafx.com/javafx/11.0.1" xmlns:fx="http://javafx.com/fxml/1" fx:controller="com.view.ctrl.LoginController">
   <children>
      <Text fx:id="weedagamaLabel" fill="WHITE" layoutX="20.0" layoutY="61.0" opacity="0.0" strokeType="OUTSIDE" strokeWidth="0.0" text="Sanasa Society"...>
      <Text fx:id="sanasaLabel" fill="WHITE" layoutX="199.0" layoutY="109.0" opacity="0.0" strokeType="OUTSIDE" strokeWidth="0.0" text="Bank System"...>
      <Label fx:id="closeLabel" layoutX="480.0" layoutY="5.0" text="X" textFill="WHITE" AnchorPane.rightAnchor="8.0" AnchorPane.topAnchor="8.0"...>
      <ImageView fx:id="loadImage" fitHeight="18.0" fitWidth="287.0" layoutX="151.0" layoutY="179.0" pickOnBounds="true" preserveRatio="true"...>
      <Text fill="WHITE" layoutX="8.0" layoutY="291.0" strokeType="OUTSIDE" strokeWidth="0.0" text="CO328 - Software Engineering Project"...>
      <HBox fx:id="userNameBox" alignment="CENTER" layoutX="62.0" layoutY="120.0" opacity="0.0" spacing="5.0"...>
      <HBox fx:id="passwordBox" alignment="CENTER" layoutX="62.0" layoutY="161.0" opacity="0.0" spacing="5.0"...>
      <Button fx:id="loginBtn" layoutX="316.0" layoutY="202.0" mnemonicParsing="false" onAction="#loginBtnEvent" opacity="0.0" prefHeight="33.0" prefWidth="123.0" text="Login" />
      <Text fx:id="userLabel" fill="WHITE" layoutX="128.0" layoutY="89.0" opacity="0.0" strokeType="OUTSIDE" strokeWidth="0.0" text="User Login Application"...>
      <Text fx:id="welcomeLabel" fill="WHITE" layoutX="24.0" layoutY="59.0" opacity="0.0" strokeType="OUTSIDE" strokeWidth="0.0" text="Welcome On"...>
   </children>
</AnchorPane>
```

Code 6 – login.fxml

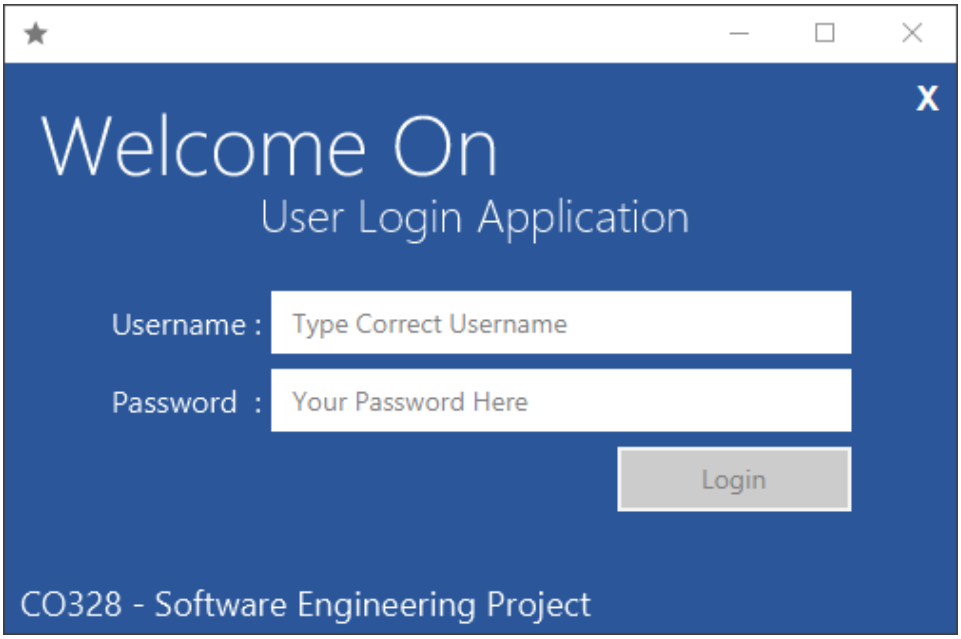


Figure 1 – LogIn Interface

A similar approach has been taken for the all the rest of GUI implementation.

addNewClient.fxml	makePayment.fxml
addNewLoanModel.fxml	member.fxml
attendance.fxml	message8ox.fxml
attendanceMarkTable.fxml	normalDepositTable.fxml
clientSummary.fxml	reports.fxml
compulsoryDepositTable.fxml	settings.fxml
dashboard.fxml	shares.fxml
deposits.fxml	sharesTable.fxml
fixedDepositTables.fxml	summary.fxml
graphs.fxml	takeLoan.fxml
loans.fxml	users.fxml
localSummary.fxml	viewAllLoansTable.fxml
login.fxml	viewAllMembersGraphic.fxml
main.fxml	viewAllMembersTable.fxml

5.2. Implementation of MySQL commands

All the MySQL commands to the database are executed through “Controller” classes. All such classes have the same basic structure.

1. Get an instance from necessary DBController for access the database

```
@Override
public void initialize(URL url, ResourceBundle rb) {

    memberDBController = (MemberDBController) ControllerFactory.getInstance().getModel(ControllerType.MEMBER);
    accountDBController = (AccountDBController) ControllerFactory.getInstance().getModel(ControllerType.ACCOUNT);
    sharesDBController = (SharesDBController) ControllerFactory.getInstance().getModel(ControllerType.SHARES);

    mainPain = MainController.mainPane;
    group = new ToggleGroup();
    maleRadio.setToggleGroup(group);
    femaleRadio.setToggleGroup(group);
    maleRadio.setSelected(true);

    clearFields();
    validationNodes();
    searchableCombo();
}
```

Code 7 - AddNewClientController.java initialize method

2. Take the user input and do the task using DBController instances we created above

```
@FXML
public void saveBtnEvent(ActionEvent event) {
    try{
        radio = (RadioButton) group.getSelectedToggle();

        //------- Creating Member Model Object---------//
        Member member = (Member) ModelFactory.getInstance().getModel(ModelType.MEMBER);
        member.setMemId(memberDBController.nextId());
        member.setName(firstNameText.getText()+" "+lastNameText.getText());
        member.setAddress(street1Text.getText()+","+street2Text.getText()+","+street3Text.getText()+","+street4Text.getText());
        member.setDob(new SimpleDateFormat("yyyy-MM-dd").format(new Date()));
        member.setNic(nicNumText.getText());
        member.setContact(contactText.getText());
        member.setGender(radio.getText().equals("Male"));
        member.setIsMember(false);
        member.setIsActive(true);
        member.setIsApplicant(yesToggle.isSelected());
        member.setParentId(yesToggle.isSelected() ? selectedParentId : -1);

        //--------create account Model object-----------------//
        Account account = (Account) ModelFactory.getInstance().getModel(ModelType.ACCOUNT);
        account.setAccId(accountDBController.nextId());
        account.setMemId(member.getMemId());
        account.setNorBalance(0.0);
        account.setComBalance(0.0);

        if (memberDBController.add(member) && accountDBController.add(account)) {
            memberDBController.initializeImage(member.getMemId());
            if(shareCheak.isSelected()){
                Shares share = (Shares) ModelFactory.getInstance().getModel(ModelType.SHARES);
                share.setShaId(sharesDBController.nextId());
                share.setMemId(member.getMemId());
                share.setShaDate(new SimpleDateFormat("yyyy-MM-dd").format(new Date()));
                share.setShaTime(new SimpleDateFormat("hh:mm:ss").format(new Date()));
                share.setAmount(Double.parseDouble(shareAmountText.getText()));
                share.setBalance(Double.parseDouble(shareAmountText.getText())+sharesDBController.getBalance(member.getMemId()));

                sharesDBController.addShare(share);
            }
            clearFields();
            MessageBoxController.showMessageBox("Add New Client","Sucsessfully Added Your New Client...!");
        } else {
            MessageBoxController.showMessageBox("Add New Client","Unsucsessfulled to do your task...!");
        }

    }catch(NullPointerException e){
        e.printStackTrace();
        System.out.println("Please Fill All Data..");
    } catch (ClassNotFoundException ex) {
        Logger.getLogger(AddNewClientController.class.getName()).log(Level.SEVERE, null, ex);
    } catch (SQLException ex) {
        Logger.getLogger(AddNewClientController.class.getName()).log(Level.SEVERE, null, ex);
    }
}
```

Code 8 - AddNewClientController.java “Save Button Event Method” (Save new Member)

5.3. Implementation of Tables

Tables are implemented using basic MySQL commands.

```
create table member(
          memId int,
          name varchar(100) not null,
          address varchar(100) not null,
          dob date not null,
          nic char(10) not null,
          contact char(10) not null,
          gender boolean not null,              /* true=male,  false=female */
          isMember boolean not null,
          isActive boolean not null,
          isApplicant boolean not null,
          parentId  int not null,
          constraint primary key(memId)
)engine=innodb;
```

Code 9 – Member Table Structure Implementation

6. Unit Testing

There are 2 basic types of classes in this software project. We have named them as,

1. View Controller Classes
2. Data Base Controller Classes

View Controller Classes handles the response from the user interfaces. Data Base Controller Classes execute SQL queries related to various instances. The requests made by the user is captured from the View Controller Classes and the produce the corresponding output using the methods available in Data Base Controller Classes.

All the methods in Data Base Controller Classes use SQL queries. Therefore the unit testing is carried out in 2 separate paths. Arithmetic calculations and other basic functions used in View Controller Classes are tested using JUnit while the methods that use database connections are tested using DBUnit. We separated these two unit testing as

1. Unit testing of components with the database connection
2. Unit testing of components within the application

6.1. Unit testing of components with the database connection

These tests are carried out in classes that uses methods to update and retrieve data from the main database. The followings are the software components that are used in these scenarios.

- Maven 3.3 – Built automation tool to manage projects with tasks like including Dependencies.
- InteliJ 2017.5 – IDE
- Dependencies
  - DBUnit 2.7 – Unit Testing Framework
  - H2 Database 1.4 – Relational Database Management System

Maven 3.3 is used here because there are many dependencies as well as plugins that are required to implement DBUnit tests. Maven-Compiler-Plugin is used to compile the project and run test cases.
DBUnit is a DBUnit is a JUnit extension targeted at database-driven projects that, among other things, puts the database into a known state between test runs. This is an excellent way to avoid the myriad of problems that can occur when one test case corrupts the database and causes subsequent tests to fail or exacerbate the damage. DBUnit has the ability to export and import the database data to and from XML datasets. However, most of DBUnit methods are extensions of JUnit 3, which is an older version of JUnit. Therefore JUnit 4 was also added as a dependency in order to execute tests easily. Especially since the "@Test" annotation was introduced in JUnit 4. In order to get Junit4 run compatibly with DBUnit, in Maven, Junit platform launcher and Junit platform engine are also added as dependencies. The complete Project Object Model (POM) is provided at the end of the report.

H2 Database is used in the tests to create mock tables and databases.
The basic structure of a method in Data Base Controller Class has the following shape. In the beginning the method would create a String query using the user input and then execute that query using the connection to the database. There are 4 classes that are responsible for creating the database connection

1. Main Class
2. LogIn Class
3. UserDBControllerClass
4. DBConnectionClass

In the start Main class calls an Instance of LogIn Class that gives the Login Interface. Once the user enters a username and password, it triggers the "userLogIn" method in UserDBController Class. "userLogIn" method creates a DBConnection Class object with the user name and password. This object can be used to get a connection with the database.

The above procedure is explained here to verify that it's **impossible to use the main database** without creating a connection using a valid credentials. Therefore in order to unit test all the methods that uses a Connection to the database, the following method need to be used.

```
private DBConnection(String username, String password ) throws ClassNotFoundException, SQLException{
    Class.forName("com.mysql.jdbc.Driver");
    if(conn != null) conn.close();
    conn = DriverManager.getConnection("jdbc:mysql://localhost/sanasa",username,password);
    //++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
    //UNIT TESTING MODULE
    try {
        JdbcDataSource dataSource = new JdbcDataSource();
        dataSource.setURL("jdbc:h2:mem:default;DB_CLOSE_DELAY=-1;init=runscript from 'classpath:../classes/sql/schema.sql'");
        dataSource.setUser("sa");
        dataSource.setPassword("sa");
        IDataSet data;
        data = new FlatXmlDataSetBuilder().build(getClass().getClassLoader().getResourceAsStream("../classes/xml/data.xml"));
        conn = dataSource.getConnection();
    }catch (Exception e){
        System.out.printf("Error" + e);
    }
    //++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
}
```

Code 10 - DBConnection.java

This is a snapshot of the DBConnection Class. This class is used by all other classes when they need a connection with the database. The only way to create a connection is to create an object of this class. "createConnection" method only returns a connection that was previously created. All the other classes uses this method because a connection is created when a user is log in.

However, this system is bypassed for the unit testing as showed in the snapshot. Here the original code is commented and the unit testing part is inserted temporally. An external mysql schema is used to create a temporary database and the data for that database is filled using "data.xml" file. This is possible due to the use of DBUnit and H2Database. In this way, a mock database is created when other classes try to get a connection.
Another advantage of using DBUnit is that the methods "getSetUpOperation()" & "getTearDownOperation()" are available to refresh and clean the cache before each test cases. Therefore each test can be run independently from previously created mock databases.

This is the initial setup for the unit testing with the Database. Afterwards we have identified 4 types of testing procedures to cover all the DBController Classes. Procedure no 1, 2, & 3 are used when the methods have return types of Boolean, int, double, or no return type (void). Procedure 4 is used when the return type is of a observable list or similar. The 4 procedures are as follow,

1. Create a mock table.
   Update a row with known value.
   Execute the query in the method.
   Compare the result with the known values used.
   This procedure is used when methods need to find the next ID for a table.

2. Create two mock tables. One is for the method and the other table is to store the expected value
   Update the table for the method using the query.
   Compare the result with the second table
   This procedure is used when data is added to the tables.

3. Create a mock table
   Update the table using the query in the method
   Retrieve the updated data from the table.
   Compare the results to verify if the table is properly updated
   This procedure is used when the tables are updated.

4. Create a mock table
   Store the expected results in Array Lists
   Execute the query in the method.
   Store the result using an Array Lists.
   Compare the Array Lists using JUnit.
   This procedure is used when the method returns an Observable List.

The implementation of these tests are explained using the "AccountDBControllerClass". This class handles all the methods that uses the "account" table. Following are the methods available in this class.

1. nextId() – Returns the next account ID when a new account is about to added.
2. add(Account account) – Add a new record to the account table.
3. updateNormalBalance(Account account) – Update the Normal Balance of a specific account.
4. updtaeCompulsoryBalance(Account account) – Update the Compulsory Balance of an account.
5. getNormalBalance(int memberID) – Return the Normal Balance of a specific account.
6. getCompulsoryBalance(int memberID) – Return the Compulsory Balance of an account.
7. getAccountSum() – Return the total sum of Normal, Compulsory & Fixed Balance of all the accounts.

```
create table IF NOT EXISTS account(
    `accId` int,
    `memId` int not null,
    `norBalance` double not null,
    `comBalance` double,
    PRIMARY KEY(`accId`)
);


create table IF NOT EXISTS testaccount1(
    `accId` int,
    `memId` int not null,
    `norBalance` double not null,
    `comBalance` double ,
PRIMARY KEY(`accId`)
);


create table IF NOT EXISTS fixed(
    `fixId` int,
    `accId` int not null,
    `fixDate` date not null,
    `fixTime` time not null,
    `period` int not null,
    `amount` double not null,
    `isMaturity` boolean not null,
    `isWithdraw` boolean not null,
    `isBond` boolean not null,
    `interest` double,
    PRIMARY KEY(`fixId`),
    foreign key(accId) references account(accId)
);
```

```
create table IF NOT EXISTS testaccount2(
    `accId` int,
    `memId` int not null,
    `norBalance` double not null,
    `comBalance` double ,
PRIMARY KEY(`accId`)
);


create table IF NOT EXISTS testaccount3(
    `accId` int,
    `memId` int not null,
    `norBalance` double not null,
    `comBalance` double ,
PRIMARY KEY(`accId`)
);
```

Code 11 - The schema that is used to create the mock database

```
<?xml version="1.0" encoding="UTF-8"?>
<dataset>
    <account accId='1' memId='1' norBalance='331630.94' comBalance='357820.32' />
    <testaccount1 accId='2' memId='1' norBalance='10.00'  comBalance='0.0' />
    <testaccount2 accId='2' memId='1' norBalance='20.00'  comBalance='0.0' />
    <testaccount3 accId='2' memId='1' norBalance='10.00'  comBalance='50.0' />
    <fixed
        fixId="1"
        accId="1"
        fixDate="1991-07-30"
        fixTime="08:10:59"
        period="30"
        amount="3438.18"
        isMaturity="1"
        isWithdraw="1"
        isBond="0"
        interest="0"/>
</dataset>
```

Code 12 - The ".xml" file used to fill the mock tables

6.1.1. Test for Account next ID

The procedure 1 is used in this instance. The name of the table is "account". In the test, the next account Id is calculated. It should be 2 because there is only 1 record in the mock table. The mock table use for the testing is as follow,

accId	memId	norBalance	comBalance
1	1	331630.94	357820.32

The test code for the unit test is as follow,

```
@Test
public void test1() throws Exception {
    UserDBController userDBController = (UserDBController) ControllerFactory.getInstance().getModel(ControllerType.USER);
    User user = userDBController.userLogIn("risith", "hello123");
    getSetUpOperation();
    AccountDBController accountDBController = (AccountDBController) ControllerFactory.getInstance().getModel(ControllerType.ACCOUNT);
    int realVal= accountDBController.nextId();

    int expectedVal = 2;
    assertEquals(expectedVal, realVal);
    getTearDownOperation();
}
```

Code 13 – Testing nextId()

6.1.2. Test for Add new Account

In order to add a new account the 2nd procedure is used. Here two mock tables are used ("account" & "testaccount1"). The table "testaccount1" has only 1 entry. The data in this entry is entered to the table "account" using the method.

Afterwards the last entry of table "account" is retrieved using a mysql query and stored in a Table data structure (ITable). The data in "testaccount1" is also retrieved and stored in similar Table data structure. Then these two are compares using the "assertEquals()" method in DBUnit.

The two mock tables and the test code is given below.

account

accId	memId	norBalance	comBalance
1	1	331630.94	357820.32

testaccount1

accId	memId	norBalance	comBalance
2	1	10.00	0.00

```
@Test
public void test2() throws Exception {
    DbUnitAssert testMe = new DbUnitAssert();
    getSetUpOperation();

    Account account = (Account) ModelFactory.getInstance().getModel(ModelType.ACCOUNT);
    account.setAccId(2);
    account.setMemId(1);
    account.setComBalance(10.00);
    account.setNorBalance(10.00);

    AccountDBController accountDBController = (AccountDBController) ControllerFactory.getInstance().getModel(ControllerType.ACCOUNT);
    accountDBController.add(account);

    ITable actualData = getConnection().createQueryTable("result_name","SELECT * FROM account WHERE accId='2'");
    IDataSet data = new FlatXmlDataSetBuilder().build(getClass().getClassLoader().getResourceAsStream("../classes/xml/data.xml"));
    ITable expectedTable = data.getTable("testaccount1");
    testMe.assertEqualsIgnoreCols(expectedTable, actualData, new String[] {"comBalance"});
    getTearDownOperation();
}
```

Code 14 – Testing add(Account account)

6.1.3. Test for Updating Normal & Compulsory Balances

For the updateNormalBalance(), the mock tables "account" & "testaccount2" are used. The 3rd procedure is implemented for this test. The table is initiated with only 1 row. Then using the methods available a new row is added with the known details.

Then the Normal Balance of the newly added row is updated so that the updated row is equal to the row in table "testaccount2"

Afterwards, the updated row and the row of "testaccount2" are stored in two ITable objects. These two are compared using "assertEqualsIgnoreCols()" method of DBUnit.

The steps are same for the updateCompulsoryBalance().The 3rd procedure is used for this as well. The tables used in this scenario are "account" & "testaccount3".
The mock tables and the test code is as follow,

account

accId	memId	norBalance	comBalance
1	1	331630.94	357820.32

testaccount2

accId	memId	norBalance	comBalance
2	1	20.00	0.00

testaccount3

accId	memId	norBalance	comBalance
2	1	20.00	0.00

```
@Test
public void test3() throws Exception {
    DbUnitAssert testMe = new DbUnitAssert();
    getSetUpOperation();

    Account account1 = (Account) ModelFactory.getInstance().getModel(ModelType.ACCOUNT);
    Account account2 = (Account) ModelFactory.getInstance().getModel(ModelType.ACCOUNT);

    account1.setAccId(2);
    account1.setMemId(1);
    account1.setComBalance(10.00);
    account1.setNorBalance(10.00);
    account2.setAccId(2);
    account2.setMemId(1);
    account2.setComBalance(10.00);
    account2.setNorBalance(20.00);

    AccountDBController accountDBController = (AccountDBController) ControllerFactory.getInstance().getModel(ControllerType.ACCOUNT);
    accountDBController.add(account1);
    accountDBController.updateNormalBalance(account2);

    ITable actualData = getConnection().createQueryTable("result_name","SELECT * FROM account WHERE accId='2'");
    IDataSet data = new FlatXmlDataSetBuilder().build(getClass().getClassLoader().getResourceAsStream("../classes/xml/data.xml"));
    ITable expectedTable = data.getTable("testaccount2");
    testMe.assertEqualsIgnoreCols(expectedTable, actualData, new String[] {"comBalance"});
    getTearDownOperation();
}
```

Code 15 – Testing updateNormalBalance()

```
@Test
public void test4() throws Exception {
    DbUnitAssert testMe = new DbUnitAssert();
    getSetUpOperation();

    Account account1 = (Account) ModelFactory.getInstance().getModel(ModelType.ACCOUNT);
    Account account2 = (Account) ModelFactory.getInstance().getModel(ModelType.ACCOUNT);

    account1.setAccId(2);
    account1.setMemId(1);
    account1.setComBalance(10.00);
    account1.setNorBalance(10.00);

    account2.setAccId(2);
    account2.setMemId(1);
    account2.setComBalance(50.00);
    account2.setNorBalance(10.00);

    AccountDBController accountDBController = (AccountDBController) ControllerFactory.getInstance().getModel(ControllerType.ACCOUNT);
    accountDBController.add(account1);
    accountDBController.updateCompulsoryBalance(account2);

    ITable actualData = getConnection().createQueryTable("result_name","SELECT * FROM account WHERE accId='2'");
    IDataSet data = new FlatXmlDataSetBuilder().build(getClass().getClassLoader().getResourceAsStream("../classes/xml/data.xml"));
    ITable expectedTable = data.getTable("testaccount3");
    testMe.assertEquals(expectedTable, actualData);
    getTearDownOperation();
}
```

Code 15 – Testing updateCompulsoryBalance()

6.1.4. Test for Retrieving Normal & Compulsory Balances

For both methods the 1st procedure is used. The mock table used in these situations is the table “account”.
In each test, the value of Normal Balance and Compulsory Balance is retrieved by executing the method. Then the result value is compared with the already known value from table “account”.
For the comparison, JUnit is used here. The table and the test methods are as follows.

Account

accId	memId	norBalance	comBalance
1	1	331630.94	357820.32

```
@Test
public void test5() throws Exception {
    getSetUpOperation();

    Account account = (Account) ModelFactory.getInstance().getModel(ModelType.ACCOUNT);

    account.setAccId(2);
    account.setMemId(1);
    account.setComBalance(10.00);
    account.setNorBalance(10.00);

    AccountDBController accountDBController = (AccountDBController) ControllerFactory.getInstance().getModel(ControllerType.ACCOUNT);
    accountDBController.add(account);

    Double actualValue = accountDBController.getNormalBalance(1);
    Double expectedValue = 331640.94;

    assertEquals(expectedValue,actualValue);
    getTearDownOperation();
}
```

Code 16 – Testing getNormalBalance()

```
@Test
public void test6() throws Exception {

    getSetUpOperation();

    Account account1 = (Account) ModelFactory.getInstance().getModel(ModelType.ACCOUNT);
    Account account2 = (Account) ModelFactory.getInstance().getModel(ModelType.ACCOUNT);

    account1.setAccId(2);
    account1.setMemId(1);
    account1.setComBalance(10.00);
    account1.setNorBalance(10.00);

    account2.setAccId(2);
    account2.setMemId(1);
    account2.setComBalance(50.00);
    account2.setNorBalance(10.00);

    AccountDBController accountDBController = (AccountDBController) ControllerFactory.getInstance().getModel(ControllerType.ACCOUNT);
    accountDBController.add(account1);
    accountDBController.updateCompulsoryBalance(account2);

    Double actualValue = accountDBController.getCompulsoryBalance(1);
    Double expectedValue = 357830.32;

    assertEquals(expectedValue,actualValue);
    getTearDownOperation();
}
```

Code 16 – Testing getCompulsoryBalance()

6.1.5. Test for Account Balance Summation

This method is tested using 4th procedure. Here two tables are used ("account" & "fixed").
Here, the sum of Normal Balance equal to the value of data in "norBalance" in table "account".
The sum of Compulsory Balance equal to the value of data in "comBalance" in table "account".
The sum of fixed Balance equal to the value of data in "amount" in table "fixed".
The result is an object that has all these 3 values.
The result after executing the method is stored in an array list.
It is compared with a predefined array list that already has these 3 values using JUnit.
The tables and test codes are as follow,

Account

accId	memId	norBalance	comBalance
1	1	331630.94	357820.32

Fixed

fixId	accId	fixDate	fixTime	period	amount	isMatuarity	isWithdraw	isBond	interest
1	1	1991-.7-30	08:10:59	30	3438.18	1	1	0	0

```
@Test
public void test7() throws Exception {

    getSetUpOperation();

    AccountDBController accountDBController = (AccountDBController) ControllerFactory.getInstance().getModel(ControllerType.ACCOUNT);
    ObservableList<GraphModel> actualVal =  accountDBController.getAccountSum();
    ArrayList<String>  actualValue = new ArrayList<>() ;
    ArrayList<String>  expectValue = new ArrayList<>() ;

    Double normal = 331630.94;
    Double compulsory = 357820.32;
    Double fixed = 3438.18;

    expectValue.add(normal.toString());
    expectValue.add(compulsory.toString());
    expectValue.add(fixed.toString());


    actualVal.forEach((tab) -> {
        actualValue.add(tab.getNumber().toString());
    });

    assertEquals(expectValue,actualValue);
    getTearDownOperation();
}
```

Code 17 – Testing getAccountSum()

6.1.6. Results

Following are the results of executing these tests for the AccountDBController Class,

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running com.database.ctrl.child.AccountDBControllerTest
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
Tests run: 7, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 1.402 sec

Results :

Tests run: 7, Failures: 0, Errors: 0, Skipped: 0

[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  5.429 s
[INFO] Finished at: 2020-08-19T20:09:44+05:30
[INFO] ------------------------------------------------------------------------
```

Figure 2 – Test run INFO

6.1.7. Continuation

Similarly the tests were carried out for all the Data Base Controller Classes and the results are shown below. The only failed cases happened in AccountDBController Class in the second time when test cases are running. That scenario is explained in the conclusion.

Class	Method	Description	Procedure				Status
			1	2	3	4	
AttendenceDBCOntroller	getMonthlyAttendance(int year)	Returns Monthly Attendance of all members of a certain year				X	Pass
CompulsoryDBController	nextId()	Calculate the next ID of the table	X				Pass
	getCompulsoryDeposits(int memberId)	Get the total amount of Compulsory deposit of a certain member				X	Pass
	getLastInterestDayCount(int memberId)	Get the previous day that the interest has been calculated	X				Pass
	getNextInterestDetail(int memberId)	Get a detailed report of interest rates for a curtained member				X	Pass
	addNewCompulsoryDeposit(Compulsory compulsory, Account account)	Create a new entity for the "Compulsory" table		X			Pass
DetailDBController	getDetail(int detailId)	Returns detail regarding certain Loan types	X				Pass
FixedDBController	nextId()	Calculate the next ID of the table	X				Pass
	add (Fixed fixed)	Add a new entity for the Fixed table		X			Pass
	makeFixedDepositBond (int fixId)	Update the Bond type of a specific Fixed deposits		X			Not Tested
	getMemberCurrentFixedDeposits(int accId)	Get all the fixed deposits owned by a specific member (Inactive ones included)				X	Pass
	getActiveFixedDepositCount(int memberId)	Get all the Active fixed deposit details of a specific member	X				Pass
GurrantorDBController	getNextGuarantorId()	Get the ID number for the next Guarantor	X				Pass
	addNewGuarantors(ObservableList< Guarantor> guarantorList)	Create a new entity in the "guarantor" table		X			Pass
LoneModelDBController	nextId()	Calculate the next ID of the table	X				Pass
	add(LoanModel loan)	Create a new entity in the "Loan" table		X			Pass
	getAll()	Get details of all the loans currently active in the bank				X	Pass
	countLoanModels()	Returns the number of Loan Models available	X				Pass
	deleteLoanModel(int loanId)	Return the specific detail of a loan model	X				Pass
LoanTakenDBController	nextId()	Calculate the next ID of the table	X				Pass
	add (LoanTaken loanTaken, ObservableList<Guarantor> guaList, ObservableList<Payment> paymentList)	Add a new entity for the "LoanTaken" table		X			Not Tested
	getDebtors()	get all the details of currently active debtors				X	Pass
	getClientLoans(int memId)	get all the loans taken by a specific member				X	Pass
	updateMemberLoanBalance(int takenId, Double balance)	Update the loan balance of a specific loan once an installment is paid.			X		Pass
	getLoanModelSum()	Get the sum of all the Loan Models available				X	Pass
LogInDBController	nextId()	Calculate the next ID of the table	X				Pass
	addNewLogIn(Log log)	Add a new entity for the "LogIn" table		X			Not Tested
	updateLogOut(Log log)	Change the status of a existing entity in the "LogIn" table to "LogOut"			X		Not Tested
	getAllLogs()	Get a detail list of participants in the table				X	Not Tested
MemberDBController	nextId()	Calculate the next ID of the table	X				Pass
	add(Member member)	create a new entity in the "Member" table		X			Not Tested
	search(int t)	Search a specific member			X		Not Tested
	likeSearchMember(String text)	Return a list of suggestions for the search				X	Not Tested
	likeSearchClient (String text)	Return a list of suggestions for the search				X	Not Tested
	getAllClientWithAccount(String dataType)	get the client list of the bank				X	Not Tested
	initializeImage(int memId)	Add an entity for the image of a client			X		Not Tested
	addImage (int memId, File file)	Add an image of a client		X			Not Tested
	getImage (int memId)	Return the image of a client			X		Not Tested
	updateMember(String text, int memId)	Update the membership status of a certain member			X		Not Tested

NormalDBController	nextId()	Calculate the next ID of the table	X				Pass
	getNormalDeposits(int memberId)	get the Normal deposit value of a specific member			X		Not Tested
	getLastInterestDayCount(int memberId)	Get the previous day that the interest was added	X				Not Tested
	getNextInterestDetail(int memberId)	Get a detail report of interest rates for a specific member	X				Not Tested
	addNewNormalDeposit (Normal normal, Account account)	Create an entity in the table		X			Not Tested
PaymentDBController	nextId()	Calculate the next ID of the table	X				Pass
	addNewPayments(ObservableList<Payment> paymentList)	Create a new entity in the table		X			Not Tested
	getNextPayments(int memberId)	Get the next payment amount for a specific member				X	Not Tested
	getPreviousPayments(int memberId)	Get a list of last payments amount for a specific member				X	Not Tested
	getLastPayment(int memberId)	Get the last payment amount for a specific member			X		Not Tested
SharesDBController	nextId()	Calculate the next ID of the table	X				Pass
	addShare(Shares share)	Create a new entity in the table	X				Not Tested
	getBalance(int memId)	Not Implemented	X				Not Tested
	getAllShareBalance()	Not Implemented				X	Not Tested
	getAccountSum()	Not Implemented				X	Not Tested
	getMemberShares(int memberId)	Not Implemented				X	Not Tested
UserDBController	addNewUser(User user, String password)	Create a new entity in the table				X	Not Tested
	getAllUsers()	Returns a list of all the users.		X			Not Tested

6.1.8. Conclusion

It was found that in the “account” table the columns “acId” & “memId” can have different values.
It is as if one member can have different accounts.
However that is not possible.
The only possible scenario is that one member can handover his ownership to another new member.
This was found out in the unit test for the AccountDBController.

The results are displayed below,

```
Results :

Failed tests:   test5(com.database.ctrl.child.AccountDBControllerTest): expected
  test6(com.database.ctrl.child.AccountDBControllerTest): expected:<357830.32> b

Tests run: 7, Failures: 2, Errors: 0, Skipped: 0

[INFO] ------------------------------------------------------------------------
[INFO] BUILD FAILURE
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  5.804 s
[INFO] Finished at: 2020-08-19T20:51:30+05:30
[INFO] ------------------------------------------------------------------------
```

Figure 3 – Result 1

```
Results :

Failed tests:   test6(com.database.ctrl.child.AccountDBControllerTest): expe

Tests run: 7, Failures: 1, Errors: 0, Skipped: 0

[INFO] ------------------------------------------------------------------------
[INFO] BUILD FAILURE
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  5.878 s
[INFO] Finished at: 2020-08-19T20:50:34+05:30
[INFO] ------------------------------------------------------------------------
```

Figure 4 – Result 2

6.2. Unit testing of components with the Functions

In here the functions with calculation and functions tested with the Junit 5.0. The functions selected and made test cases according to their functional attributes. Each unit is tested with different scenarios where the critical scenarios like edge values etc.

Junit 5.6.2
Date of Release: April 10, 2020

In here Junit 5.0 is used because,

- Enables writing test cases while developing the software that helps test early and detect issues.
- Ensure the functionality is performing as expected every time the code is modified by the use of repeatable automated test cases.
- Supported by all IDE including Eclipse, NetBeans and IntelliJ etc.
- Integrates with Ant and Maven (CI) that enables execution of test suites or test cases as part of the build process, capturing test result and reporting.
- High accuracy and efficient in use

The inputs are given according to the user cases and compared the result with the actual result it its same then the taste case is pass and if not it called as fail. It covers different cases and aspects of the application's behavior so assume that the all the real time scenarios are covered.

IN these test cases unit test is done for functions of of class NectPayment and Normal
getInterestAmount()-Calculate the interest amount of the user
getFineAmount()-Calculate the Fines of the user
getInstallAmount()-Calculate the Installed amount of the user
getPayAmount()-Calculate the amount of payable of the user
getWithdraw()-Withrawal amount
getDeposit()-Depositted amount

```
@Test
public void test1(){
    NextPayment nextpay = new NextPayment();
    nextpay.setRateType(0);
    nextpay.setRate(10);
    nextpay.setBalance(1000);
    nextpay.setNumOfDates(0);
    nextpay.setMethod(true);

    assertEquals(0,nextpay.getInterestAmount(),DELTA);
}
```

Code 18 – Testing initial interest amount

In Scope	Out of scope
Get methods which make calculations in the functions are tested here. Edge cases and the bank required functions are tested according to the input scenarios	Getter setter methods without internal calculations are not tested here because when using the application those functions come to function.

6.2.1. Test cases

ID	Test Case	Input Values	Expected Output
001	InterestAmount (Class NextPayment)	RateType = daily Rate = 10% Balance =1000 NumOfDates = 0	InterestAmount = 0
002	InterestAmount (Class NextPayment)	RateType = daily Rate = 10% Balance =1000 NumOfDates = 20	InterestAmount = 2000.00
003	InterestAmount (Class NextPayment)	RateType = daily Rate = 10% Balance =0 NumOfDates = 20	InterestAmount = 0
004	InterestAmount (Class NextPayment)	RateType = annual Rate = 10% Balance =1000 NumOfDates = 20	InterestAmount = 5.48
005	InterestAmount (Class NextPayment)	RateType = annual Rate = 10% Balance =1000 NumOfDates = 0	InterestAmount = 0
006	InterestAmount (Class NextPayment)	RateType = monthly Rate = 10% Balance =1000 NumOfDates = 20	InterestAmount = 65.75
007	InterestAmount (Class NextPayment)	RateType = monthly Rate = 10% Balance =1000 NumOfDates = 0	InterestAmount = 0
008	InterestAmount (Class NextPayment))	RateType = monthly Rate = 10% Balance =-1000 NumOfDates = 20	InterestAmount = 0
009	FineAmount (Class NextPayment)	FineType =cost Balance =1000 NumOfDates = 20	FineAmount = 0
010	FineAmount (Class NextPayment)	FineType = cost Balance =1000 NumOfDates = 31	FineAmount = 2.00
011	FineAmount (Class NextPayment)	FineType = cost Balance =1000 NumOfDates = 30	FineAmount = 0
012	FineAmount (Class NextPayment)	FineType =rate Balance =1000 NumOfDates = 31	FineAmount = 0.05
013	FineAmount (Class NextPayment)	FineType =rate Balance =1000 NumOfDates = 30	FineAmount = 0
014	FineAmount (Class NextPayment)	FineType =None Balance =1000 NumOfDates = 30	FineAmount = 0

015	InstallAmount (Class NextPayment)	Period=50 Balance=1000 Dates=31	InstallAmount=20.42
016	InstallAmount (Class NextPayment)	Period=12 Balance=1000 Dates=365	InstallAmount=1000.00
017	InstallAmount (Class NextPayment)	Period=1 Balance=1000 Dates=365	InstallAmount=1000.00
018	PayAmount (Class NextPayment)	RateType- monthly Rate=10% FineType=rate Balance=1000 NumOfDates=31 Period=120 Dates=5	PayAmount= 110.39
019	PayAmount (Class NextPayment)	RateType- monthly Rate=10% FineType=rate Balance=1000 NumOfDates=31 Period=10 Dates=0	PayAmount=201.97
020	FineAmount (Class NextPayment)	FineType =rate Balance =-1000 NumOfDates = 31	FineAmount = 0
021	Deposit (Class Normal)	Amount=1000	Deposit=1000.00
022	Deposit (Class Normal)	Amount=-1000	Deposit=****
023	WithDraw (Class Normal)	Amount=-1000	WithDraw=1000
024	WithDraw (Class Normal)	Amount=1000	WithDraw=****

6.2.2. Test Results

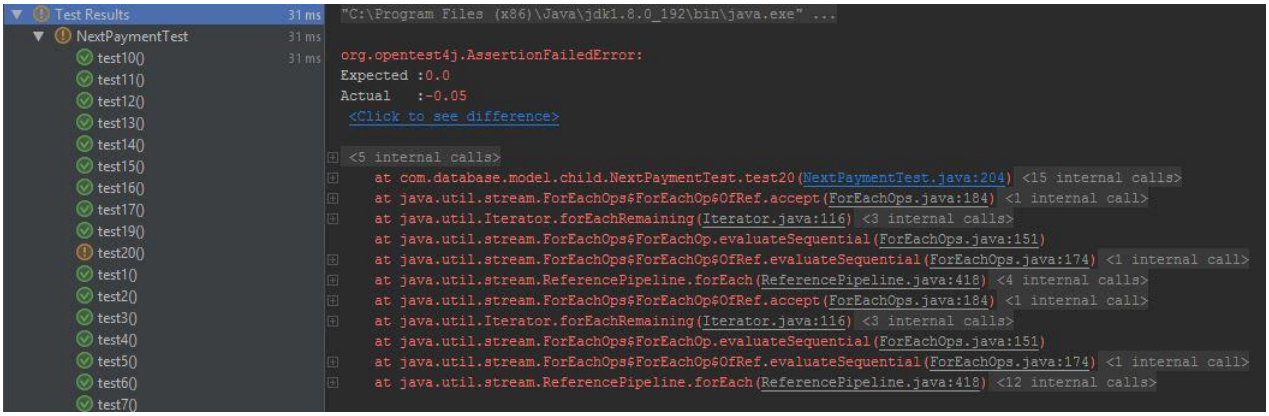


Figure 5 – Test Results

Following table shows 24 Test Results

ID	Test Case	Pass/Fail	Date tested
001	InterestAmount (Class NectPayment)	Pass	17.08.2020
002	InterestAmount (Class NectPayment)	Pass	17.08.2020
003	InterestAmount (Class NectPayment)	Pass	17.08.2020
004	InterestAmount (Class NectPayment)	Pass	17.08.2020
005	InterestAmount (Class NectPayment)	Pass	17.08.2020
006	InterestAmount (Class NectPayment)	Pass	17.08.2020
007	InterestAmount (Class NectPayment)	Pass	17.08.2020
008	InterestAmount (Class NectPayment)	Pass	17.08.2020
009	FineAmount (Class NectPayment)	Pass	17.08.2020
010	FineAmount (Class NectPayment)	Pass	17.08.2020
011	FineAmount (Class NectPayment)	Pass	17.08.2020
012	FineAmount (Class NectPayment)	Pass	17.08.2020
013	FineAmount (Class NectPayment)	Pass	17.08.2020
014	FineAmount (Class NectPayment)	Pass	17.08.2020
015	InstallAmount (Class NectPayment)	Pass	17.08.2020
016	InstallAmount (Class NectPayment)	Pass	17.08.2020
017	InstallAmount (Class NectPayment)	Pass	17.08.2020
018	PayAmount (Class NectPayment)	Pass	17.08.2020
019	PayAmount (Class NectPayment)	Pass	17.08.2020
020	FineAmount (Class NectPayment	Fail	17.08.2020
020	Deposit (Class Normal)	Pass	17.08.2020
021	Deposit (Class Normal)	Pass	17.08.2020
023	WithDraw (Class Normal)	Pass	17.08.2020
024	WithDraw (Class Normal)	Pass	17.08.2020

6.2.3. Conclusion

In the functional behavior, almost all the test cases got passed except few cases.According to the sheet that when the balance of the account get lower than 0 & a member have a loan to pay the fine amount shouldn't be calculated It seems the system almost works fine for except that scenarios. The calculation and selecting functions in these unit testing got correct results according to the actual bank system behavior.

7. Deployment

Sanasa Society Banking System is a standalone system. Since Sanasa society branches work as independent entities, it's not need to deploy into a cloud or make it public for security reasons. However, the client computer need to have

1. JRE 1.8 (at least)
2. MySQL db.

The installation package is made to install the executable JAR file to the system. However, before launching the executable JAR file for the first time, The SQL script must be executed in order to create the database for the first time. Here the database, basic users and their privileges are created.

After that step, the application can be launched using the shortcut icon that will be created in the installation process.

Furthermore, the application creates a backup database in the "backup.sql" file in the installation folder. This file is set to update automatically each day using the following syntax. This is a read only file, hence the data cannot be change in this file.

```
Mysql –u[user] –p[password] sanasa > backup.sql;
```

-END-