Assignment 4 documentation

Bank management system

Litan Virgil

Group 30425

TA: Claudia Pop

Contents

1. Functional Requirements

The application should allow account and customer management. The user should be able to insert customers, create accounts for customers and also add and withdraw money from an account.

The customers should also be notified when an operation on their accounts is performed.

Another key feature of the application is storing the current state of all components from a run to another.

An aspect of paramount importance is designing the application using design by contract and observer design pattern.

2. Objectives

Design and implement a application considering an bank management application for processing customer accounts. The relational dependencies must be stored in a java collection data structure, namely a Hashtable and a Set.

2.1 Main objectives

Developing an application be means of design by contract and create a correct database-like structure.

2.2 Secondary objectives

High accessibility of the database values and tables in order to create different edge-case scenarios that enhance the testing process.

Another important aspect of the application is designing a comprehensible set of logs that would allow an advanced user to fully understand the mechanisms that underlie the interactions between the layers of the process.

Another objective would be designing the application such a way that it could be easily further developed by separating the models, the general user interface and the main controllers.

3. Problem analysis

The main use case is an administrator creating Accounts and Persons and creating links between those using the application. The main actors being the user and the bank. The user selectes the action to be performed and a pop up window will appear asking for additional information about the action.

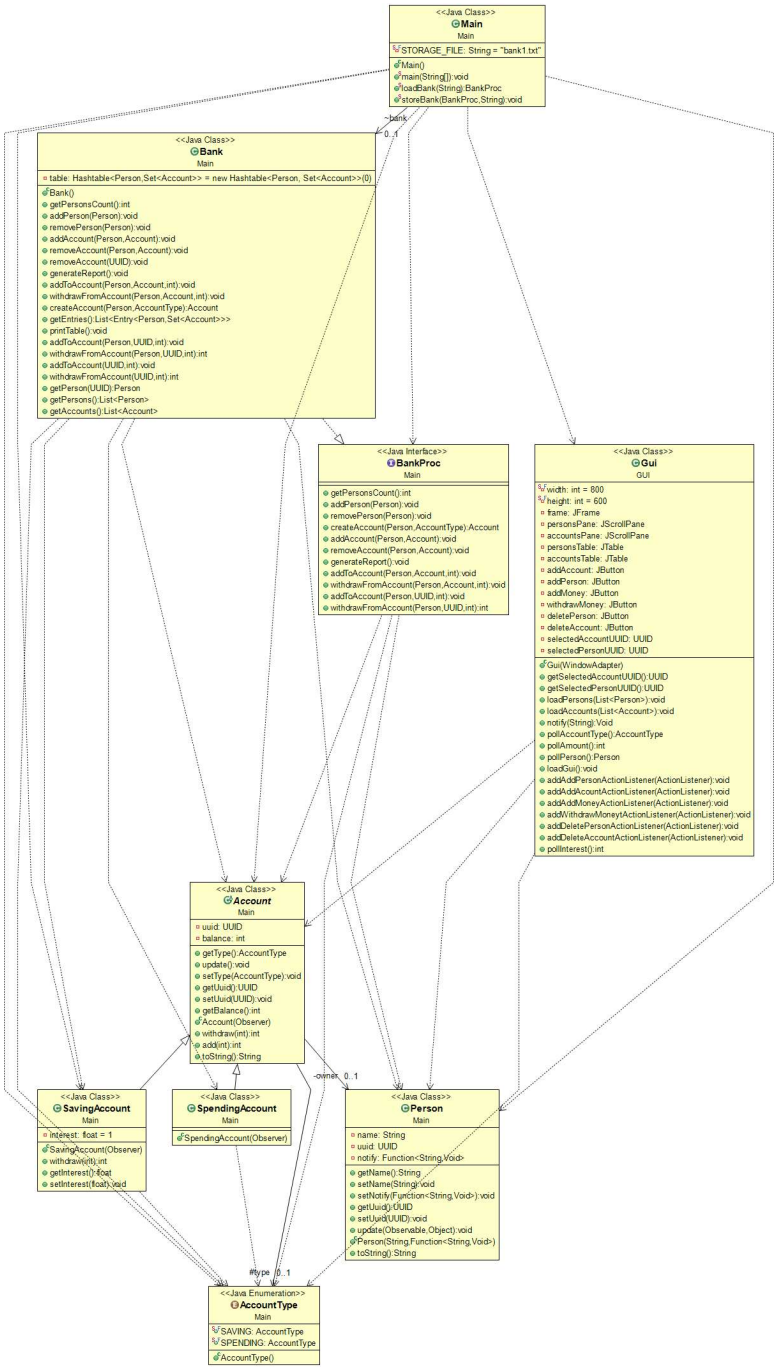
4. Design

4.1 Data structures

The used data structures where a Hashtable with Persons as keys and Sets of Accounts as values. This configuration ensures fast access and data integrity.

4.2 Class diagram

An important design pattern used in this application was Observer, in order to notify a person when modification upon one of its accounts is made.



5. Implementation

5.1)Account

The abstract account class is used in order to hold data about an account and to perform various actions. The class is abstract in order to make sure that any instance will be either of type SavingAccount or SpendingAccount.

```
public abstract class Account extends Observable implements Serializable {
    private UUID uuid;
    private int balance;
    protected AccountType type;
```

An UUID was used in order to uniquely identify each account, even through different runs of the application.

The two basic operations are withdraw and add (money into account). Given the fact that the class extends the “Observable” class the account can notify its owner about any modification made onto it.

```
    public int withdraw(int amount){
        if (amount > balance){
            System.out.println("[" + this.toString() + "]" + " not enough money in account");
            return (0);
        }
        setChanged();
        notifyObservers(-amount);
        balance -= amount;
        return (amount);
    }

    public int add(int amount){
        balance += amount;
        setChanged();
        notifyObservers(amount);
        return (balance);
    }
```

5.2)SavingAccount

The saving account overrides some inherited methods in order to conform with the specifications, namely a bigger quantity of money may be withdraw, according with the specified interest.

```
private float interest = 1;

public SavingAccount(Observer owner) {
    super(owner);
    type = AccountType.SAVING;
    // TODO Auto-generated constructor stub
}

@Override
public int withdraw(int amount){

    return (int) (super.withdraw((amount)) * interest);
}

public float getInterest() {
    return interest;
}

public void setInterest(float interest) {
    this.interest = interest;
}
```

5.3)Person

A person is uniquely identified via a UUID field, generated in the constructor and also can be distinguished by its name.

```
9       private String  name;
10      private UUID    uuid;
11      private transient Function<String, Void> notify;
```

Moreover, each person has a functional interface that has to be called when its notified by one of its accounts

```
@Override
public void update(Observable o, Object arg) {
    notify.apply("[Person" + " " + this.name + "] i got notified by " + o.toString() + " " +(((int)arg < 0)  ? " withdraw " : "added ") + Math.abs((int)arg));
}
```

5.4)BankProc

The BankProc interface ensures a design by contract design pattern by the methods it requires to be implemented and the specified post / pre and invariant conditions.

```
/*count of persons always positive
 * @post return > 0
 * */
public int getPersonsCount();

/*
 * @pre person != null
 * */
public void addPerson(Person person);
public void removePerson(Person person);

public Account createAccount(Person person, AccountType type);
```

5.5)Bank

The class Bank implements the interface BankProc and checks for all the specified conditions using assert.

```
@Override
public void addPerson(Person person) {
    assert(person != null);
    table.put(person, new HashSet<Account>());
}
```

The Bank class stores the data about each person and account using a Hashtable as primary datastructure and a Set of Accounts as a nested data structure.

```
private Hashtable<Person, Set<Account>> table
```

5.6)GUI

The General User Interface (GUI) is implemented using the Swing api. As required, it presents two JTable components, one for Persons and one for Accounts. It also allows the user to select the target of its action by clicking on it.

The data is loaded using reflection techniques in order to ensure more flexibility for future developments.

```
for (Field field : accounts.get(0).getClass().getSuperclass().getDeclaredFields()) {
    columnNames[i++] = field.getName();
}
i = 0;
for (Account o : accounts){
    int j = 0;
    for (Field field : accounts.get(0).getClass().getSuperclass().getDeclaredFields()) {
        field.setAccessible(true);
        try {
            rowsData[i][j++] = field.get(o).toString();
        } catch (IllegalArgumentException | IllegalAccessException e) {
            e.printStackTrace();
        }
    }
    i++;
}
final JTable table = new JTable(rowsData, columnNames);
table.setBounds(0, 0, accountsPane.getWidth(), accountsPane.getHeight());
table.setPreferredScrollableViewportSize(new Dimension(500, 70));
table.setFillsViewportHeight(true);
table.getSelectionModel().addListSelectionListener(new ListSelectionListener(){
    @Override
    public void valueChanged(ListSelectionEvent e) {
        selectedAccountUUID = accounts.get(table.getSelectionModel().getMaxSelectionIndex()).getUuid();
    }
});
```

All actions are performed using JButtons and the data is polled from the user using JOptionPane boxes.

```
public AccountType pollAccountType(){//TODO
    AccountType[] options = {AccountType.SAVING,
            AccountType.SPENDING};
    int n = JOptionPane.showOptionDialog(frame,
            "What type of account whould you like to
            "A Silly Question",
            JOptionPane.YES_NO_CANCEL_OPTION,
            JOptionPane.QUESTION_MESSAGE,
            null,
            options,
            options[0]);
    return options[n];
```

5.7)Main

The main controller is used in order to bind the front end with the back end of the application. This is done by setting the action listeners of the buttons. Also, at the beginning of the run the bank is loaded from a file and before closing the app a the current state of the app is stored in

the same file.

```
Gui gui = new Gui(new WindowAdapter()
{
    public void windowClosing(WindowEvent e)
    {
        storeBank(bank, STORAGE_FILE);
    }
});

gui.addAddAcountActionListener(new ActionListener(){

    @Override
    public void actionPerformed(ActionEvent e) {
        Account a = bank.createAccount(bank.getPerson(gui.getSelectedPersonUUID()), gui.pollAccountType());
        if (a.getType() == AccountType.SAVING){
            ((SavingAccount)a).setInterest(gui.pollInterest());
        }
        gui.loadAccounts(bank.getAccounts());
    }

});
```

6 Conclusions and Further Development

The development of this application was a very good exercise of OOP design but much more important it offered an insight of database related applications. Further development consists of more restrictive general user interface and more comprehensive(meaningful colors and current state display).