Financial Application Framework

Group 7 (Naimish Viradia, Dhruba Adhikari, James Singh)

List of Deliverable

1. UML
    a. UML Class Diagram
        i. Common framework class diagram
        ii. Banking application class diagram
        iii. Credit card application class diagram
    b. UML Sequence Diagram
        i. Register a personal saving account
        ii. Deposit money to saving account
        iii. Add interest to all bank accounts.
        iv. Deposit money to silver credit card account
        v. Register a silver credit card account for a person.
        vi. Withdraw money from saving account
        vii. Charge money to a silver credit card
2. Description of patterns used (Can be seen in UML Class diagram)

    a. **Party**
    Used to define standard parties for *Person* and *Company*. The interface *IParty* serves the purpose of defining interfaces, and the attributes are defined in *AParty* abstract class. Next, there are two interfaces *ICompany and IPerson* implementing *IParty* that are used to define two types of party.

    By extending *AParty* and implementing *ICompany*, *Company* defines itself as a company, gets the behaviors defined in *IParty*, and attributes of *AParty*. Similarly by extending *AParty* and implementing *IPerson*, *Person* defines itself as a person, gets the behaviors defined in *IParty*, and attributes of *AParty*.

    b. **Abstract Factory**
    We have an abstract factory named *AbstractFactory* that returns instance of *PartyFactory*, *AccountFactory* or *TransactionFactory*. These factories can be used to obtain the concrete classes of party, account and transaction respectively.

    In our common framework, we have concrete implementations *Company* and *Person* for party, *DefaultDeposit* for account and *Withdraw* and *Deposit* for transaction (under *Debit* and *Credit* types respectively).

In the banking application, we have overridden the account to create *SavingAccount* and *CreditAccount*. Also, we have added *Interest* concrete class to for Add Interest use-case.

In the credit card application (designed, but not developed), we will have *CreditCardAccount* account further extended by *Gold, Silver* and *Bronze* varieties. For transaction, there will be *Charge* (under *Debit*) additionally to use for credit card chargings/deductions.

c. **Account**
Account design pattern has been used to show the relation between account and transaction entities.

d. **Observer + Mediator**
The application GUI consists of several receiver components that change state based on some action. The framework uses mediator pattern to collaborate among the colleague components. In our scenario, there are some components which behave only as observers.

Collaborator	Subject / Observer	Action
Table model entry	Subject	On clicking any account, the mediator will be called.
Add Interest button	Observer	Enabled only when an account is selected in the account list.
Withdraw button	Observer	Enabled only when an account is selected in the account list.
Account Manager	Subject	Anything related to change in accounts will trigger the change to mediator.
Transaction Manager	Subject	New transactions lead to change in account attribute (eg. balance), and will trigger mediator to communicate the change of state.

e. **Strategy + Template**
Each use case is a strategy and has a controller. Each controller has several templates that display different dialog.

f. **Iterator**
The application has an aggregate class ASDArrayList which extends from the standard class ArrayList. A developer can get a sorted list of items by passing an attribute name by which it is to be sorted.

In our application, the account list table titles (account number, party name, city, party type, account type, balance) are clickable. When a title is clicked, the *getSortedIterator()* method of ASDArrayList is invoked with the respective attribute parameter that will return an iterator (instance of *ISortedIterator*) with list of accounts sorted by that field. The iterator is then traversed to get the accounts in sorted order by that field.

g. **Command**
Command pattern is used for accounts and transactions to manage calls to the entities. The application uses *AccountManager* and *TransactionManager* concrete classes for the purpose. *AccountManager* maintains a list of all the accounts, and *TransactionManager* maintains a list of all the transactions.

h. Singleton
We have used three singletons for *AccountFrm*, *AccountManager* and *TransactionManager* which are accessible through *ClassicSingleton*.

3. Description of plug-in points
   a. FactoryProducer

      Example
      In Bank.java, we are injecting a new factory to the framework as below:

```
FactoryProducer.addAbstractFactory(MyAccountType.MYAC, new
MyAccountFactory());
```

   b. Buttons for various use cases. We inject new controllers to the framework through action listeners.

```
JButton_CompAC.addActionListener(new AddCompanyController());
JButton_PerAC.addActionListener(new AddPersonController());
JButton_Deposit.addActionListener(new DepositController());
JButton_Withdraw.addActionListener(new WithdrawController());
```

4. Schedule

Name \ Day	Aug 11	Aug 12	Aug 13
Naimish Viradia	GUI, Sequence diagram, Class	GUI + Bank application	Credit card application on top

	diagram	(extension from common framework)	of the framework; Sequence and class diagram modification
Dhruba Adhikari	Common Framework development (account, and transaction)	Continuation	Final UML diagrams, Formatting and printouts
James Singh	Common Framework development (account, and transaction)	Common framework development continued.	Final UML diagrams, Formatting and documentation.

5. Metrics:
    a. Total Effort metrics: 3 person * 11 hrs/day * 3 days = 99 hours
    b. Effort for UML: 3 person * 11 hrs/day * 0.5 day = 16.5 hours
    c. Effort for development: 3 person * 11 hrs/day * 2.2 days = 72.6 hours
    d. Effort for presentation/demo: 3 person * 11 hrs/day * 0.3 = 9.9 hrs

6. Source code repository:
   https://github.com/JBaba/Bank (Branch: master)