# Framework Development Lab

1) Design the class diagram to a banking system with the following Use cases:
   - Create a personal account
   - Create a Company account (checking or savings)
   - Deposit money
   - Withdraw money
   - Add interest
   - Generate a report of accounts

   The user interface is that of the first frameworks homework (you can also re-use the UML from that assignment).

   The operations for each action are:

   - When the *add-interest* button is pressed, the system adds interest to all accounts in the bank.
   - When a *deposit* or *withdrawal* is done to a company account, the system sends the company an Email about the transaction.
   - When a deposit or withdrawal is done to a personal account, and the amount was larger than $500 or the resulting amount is negative, the bank sends the person an Email about the transaction.
   - Customers can have more than one account. The bank system needs to keep track of the history of deposits and withdrawals.

   The class diagram should show all classes, relationships, etc.

   Include a sequence diagram for each of the major use cases (above).

2) Design the class diagram for a Credit-Card processing system with the following use cases:
   - Create a credit card account
   - Deposit money
   - Charge the account
   - Add interest
   - Generate monthly billing report

   The user interface is that of the first frameworks homework (you can also re-use the UML from that assignment).

   Based on those client specified interfaces, you should provide 3 types of credit cards, with different interest rates for each:

| | **gold** | **silver** | **bronze** |
|---|---|---|---|
| **monthly interest (MI)** | 6% | 8% | 10% |
| **minimum payment (MP)** | 10% | 12% | 14% |

   The operations for each action are:

   - When a card is charged for more than $400, the system sends the cardholder an email about the transaction.
   - Customers can have more than one credit card (account)
   - The system needs to keep track of the history of charges and payments. For every payment or charge, the system needs to store the date, name, and amount of the transaction.

- The monthly billing report (generated from the *monthly report* button) should show:
  - previous balance: balance from last month
  - total charges: total of all charges for this month
  - total credits: total of all payments for this month
  - new balance = previous balance – total credits + total charges + MI * (previous balance – total credits)
  - total due = MP * new balance
- When a deposit or withdrawal is done to a company account, the system sends the company an Email about the transaction.
- When a deposit or withdrawal is done to a personal account, and the amount was larger than $400 or the resulting amount is negative, the bank sends the person an Email about the transaction.
- Customers can have more than one account. The bank system needs to keep track of the history of deposits and withdrawals.

The class diagram should show all classes, relationships, etc.

Include a sequence diagram for each of the major use cases (above).

3) Design a Framework for both of these applications.

   The framework should abstract out all of the common operations between these two applications, and others that would be similar in nature. The evaluation criteria for the project will be how well you have captured common parts of the intended application domain(s) in the framework, and it's internal design via patterns.

   For the framework design, use the account pattern, the party pattern, and the observer pattern.

   We will call the framework the *observable party account*.

4) Implement the Framework.

   Put the code into three packages;

   - framework
   - banking
   - creditcard

5) Implement the Banking Application using the Framework.
6) Implement the Credit Card Application using the Framework.
7) First implement the basic framework, and test for the two applications.

   For each sub-system, you should hand-in documentation including:

   - the UML design diagrams
   - descriptions of the patterns used, and the role that they play in the solution
   - A description of the plug points of the framework, and how it could be used for other applications.
   - Sequence diagrams for the major use-cases for one of the applications.

## Extra Credit

8) Add the ability to have general rules associated with various account or transaction objects. Use the *rules pattern* discussed in class.

There are several other good references, do some net-searching to research it.

9) Migrate the framework from a *white-box* to a *black/gray-box* architecture.
   a) first add several internal objects which would be typical elements to compose an application
   b) Then provide external hook-methods to compose from them
10) Add persistence for the database. You should use a *façade pattern* to hide the details of the persistence mechanism.
11) Add a third application from the framework. Consider what candidate applications could be a good match.

    Consider a retail application (e.g. bookstore). If the framework needs significant extensions to do this, consider a second framework, a *retail management framework*.
    a) How should you create this from the first framework?

       Are they co-frameworks, or does the retail framework encapsulate the billing framework?
12) Add a visual composition method to create applications from the framework. [*Very advanced*!]
    a) Design such an application, it's components, and use cases, and architecture (and patterns used).
    b) Implement it.

## References


- For a reference on the rule pattern, see:

  http://www.mum.edu/cs_dept/aarsanjani/oopsla2000/Rule Object Pattern Language7 (??)

- *A Pattern Language for Adaptive and Scalable Business Rule Construction*, Ali Arsanjani

  Business rules tend to change more frequently than the rest of the business object with which they are associated. These rules are typically implemented within the rule methods of a business object. Rules also refer to other business objects that their encompassing business object associates with; creating a web of implicit and increasingly un-maintainable dependencies. This pattern language provides a set of patterns that address the increasing need for handling scalability, adaptability and complexity.

  http://www.cbdiforum.com/downloads/ruleobject.zip

- *Rule Object: A Pattern Language for Pluggable and Adaptive Business Rule Construction*, Pattern Languages of Programming 2000

  http://jerry.cs.uiuc.edu/~plop/plop2k/proceedings/Arsanjani/Arsanjani.pdf

- *Analysis, Design, and Implementation of Distributed Java Business Frameworks Using Domain Patterns*, Proceedings of Tools '99 (IEEE Computer Society Press1999), pp. 490-500

  http://www.computer.org/proceedings/tools/0278/02780490abs.htm

- Other Ali references:

  http://www.mum.edu/cs_dept/aarsanjani/aarsanjani_ref.html

## Lab Requirement Revisions (!)

13) Add the ability to have general rules associated with various account or transaction objects. Use the *rules pattern* discussed in class.

    a) add a rule for CCards: if a purchase is more than twice the average amount of other purchases, send a fraud alert letter to the customer.

    b) add a bank rule: Individual bank clients must keep a balance of at least $400, or else pay a monthly fee or $10. (auto-deduct from their account.)

14) Add a third application to your framework, it should be an online bookstore. The requirements are:

    a) You should have three categories of products, books and videotapes.

    b) Customers can pay by creditCard, or online (PayPal).

    c) You should track the number of successful sales to each customer, and have a category of premium customer (more than 10 purchases).

    d) Premium customers get a 10% discount on all books.