Lecture 5: Introduction to Design Patterns
A Case Study

Not finished yet, for preview only

Materials by courtesy of Dr. W. Robinson

A Case Study

- In this case study we will illustrate the role of design patterns in the OOA and OOD process
- The example for the Case Study is an extension to an Automated Teller Machine (ATM)
- We begin by developing an analysis model for the ATM
- Then to develop a design model we look at the design constraints arising from system requirements, and consider design approaches that address these constraints
A bank, MegaBank, wants to embellish ordinary ATM functionality. The core marketing concept is "one customer, one card (the BrilliantCard)"

- Kinds of Accounts: checking, savings, mortgage, with others to be added later
- Queries: Account balance and account history
- Transactions: debit account (e.g. dispense cash), credit account (e.g. deposit), transfer funds
- A customer should be able to access this from an enhanced ATM or from new home PC software.

Examples: Joe and Mary Smith have 5 accounts with MegaBank — joint checking, joint savings, mortgage, Joe's personal checking, Mary's personal checking
  - Give Joe a card, Mary a different card
  - Joe can access all but Mary's personal account, Mary all but Joe's personal account
  - Each can transfer funds among the checking and savings accounts and from the checking/savings to the mortgage
Desire Configuration (HW)

ATM
ATM
ATM
PC
PC

ATM Server

Main Bank Computer

Server Database

Corporate Database

The cluster can be repeated

Login Screens for the ATM

ATM Interface

Home PC Interface
Account Summary Screens for the ATM

ATM Interface

Home PC Interface

The Prime Directive - Separate Concerns

- The fundamental of goals of OOA an OOD are to understand the functionality desired and to allocate the functionality to an appropriate set of objects
- The idea of Separation of Concerns arises from the need to reduce coupling and to increase cohesion
  - For example, mixing display (presentation) and database access in the same object leads to a bloated object that's difficult to maintain. Thus we separate the concerns of presentation from the concerns of data management into different groups of objects
Three Specific Component Areas

- In the Case Study, we will look at three major groups of objects:
  - Problem Domain Objects,
  - Human Interaction Objects, and
  - Data Management Objects,
  representing 3 separate concerns

- This separation into three areas of concern does NOT imply a particular architecture or physical packaging of objects into processes

Problem Domain Objects

- These objects embody the content of the problem to be solved
- The Problem Domain Objects are the primary area of concentration for OOA
- For the MegaBank ATM these will be things like Customer, Account, Transaction
- These objects have little or no knowledge of the other kinds of objects
- Problem Domain Objects are also known as Business Objects or Model Objects
**Human Interaction Objects**

- These provide an interface between the problem domain objects and the external user.
- The Human Interaction Objects are usually defined at the end of the OOA process or at the beginning of OOD.
- Understanding Human Interaction Objects is a good source for fleshing out the set of methods required on the Problem Domain Objects.
- For the MegaBank ATM the Human Interaction Objects will be things like *Account Summary Display*, *Login Dialog*, and so on.

**Data Management Objects**

- These objects are responsible for storage and retrieval of the state of Problem Domain Objects.
- They can also encapsulate technical facilities such as caching.
- Data Management Objects are implementation oriented and therefore are primarily in the sphere of OOD.
ATM Problem Domain Objects

By reading through the requirements for the MegaBank system, we might list things such as the following as potential class names:

- Account
- ATM Server
- Mortgage
- Home PC
- Transaction History

- Transaction
- Checking
- Session
- Bank

- Customer
- Savings
- ATM
- Corporate DB
Human Interaction Objects for the ATM

Potential classes related to Human Interaction are things like

- Login Dialog
- Account History Display
- Account Debit Dialog
- Account Summary Display
- Account Credit Dialog
- Funds Transfer Dialog

Class Diagram for Human Interaction Objects

```
   Dialog
 /    \
LoginDialog  Account Debit Dialog  Account Credit Dialog  Funds Transfer Dialog

   Display
 /    \
Account History Display  Account Summary Display
```
Relating Human Interface Objects and Problem Domain Objects

- The **LoginDialog** "knows" the Bank, and uses it to verify the user's credentials
- The **AccountDebitDialog** and other classes "know" a particular account and update it with the latest **Transaction**

Data Management Objects for the ATM

- There is one Data Management Object for each Problem Domain Object that requires persistent storage
- Example Data Management Objects include
  - **CustomerDM**
  - **AccountDM**
  - **TransactionDM**
  - **SessionDM**
- The **SessionDM** writes data to the ATM Server database. The others read/write data from/to the corporate database
Applying Design Patterns to the ATM

- We will identify design problems for each of the three component areas described above.
- Then we'll analyze these problems according to whether they are creational, structural or behavioral issues and how specific patterns can help with the solution.
- Design problems arise naturally from a statement of the functional requirements, the set of design constraints, and the object model for the system.

Design constraints for the ATM

- The software must support multiple kinds of front-ends: standard ATM, including cash dispensing, as well as home software (without cash dispensing).
- Initially, the server will use Informix for the statistical database. MegaBank has made a strategic decision to use Oracle, but that won't be in place for another 6 months.
- All Problem Domain Objects (Customer, Account, etc) are to be located on the ATM Server and accessed from the front-end systems.
Design the Problem Domain Objects

- How do we create the correct subclass of Account?
- How do we ensure that there is only one instance of Bank?
- How do we provide remote access from the Human Interaction Objects to the Problem Domain Objects?
- How do we provide traversal of "container" objects (e.g., traversing the list of Transactions associated with an Account)?

Design the Human Interface Objects

- How do we anticipate the need for implementations of the Human Interface Objects on at least two Windowing systems?
- How do we deal with user command processing?
- How do we deal with interactions among screen elements?
Design the Data Management Objects

- How do we anticipate the need for implementations of the Design the Data Management Objects on two DBMS packages

Solving the Design Problems

- We will provide brief descriptions of solutions to the four design questions raised for the Problem Domain Objects
- From these we will show how more specific solutions can be generalized into design patterns
- The four problems discussed are labeled 1-4 in the next several slides
1-Creating the Correct Subclass of Account

- Suppose we have account number 105978321 corresponding to the checking account for Joe Smith
- We'd like to create a CheckingAccount object using that number. However, in certain contexts we might know only the account number, and not know it corresponds to a checking account
- Writing code such as
  
  ```java
  Account currentAccount = new Account (105978321);
  ```

  will not suffice

Use a Special Method to Create Accounts

- We can solve this particular problem by declaring a method

  ```java
  public Account createAccount (String account ID);
  ```

  that returns an instance of the right subclass of Account based on the key passed in
Where Does `createAccount()` Belong?

- In the ATM example, there are two choices
  - place the method on an existing class, such as `Bank`
  - add a *new class* whose sole concern is creation of objects. In this case we would probably call the new class `AccountFactory`

Using the `createAccount()` Method

- Assume that we have added the public method `createAccount()` to the `Bank` class
- Suppose `theBank` is an instance of the `Bank` class. Then client code would create a new account by invoking the `createAccount()` method as follows:

```java
//Setup the account identifier
String accountId = getId();
Account currentAccount =
    theBank.createAccount(accountId);
```
Two Creational Patterns

- Note that the same techniques can be used to construct instances of other classes in the ATM example, such as Transaction or Customer.
- `createAccount()` is an example of a Factory Method, one of the Creational patterns we will cover later.
- `AccountFactory` is an example of the Creational pattern known as the *Abstract Factory Pattern*.

2- Ensuring One Instance of Bank

- What we want in this case is not to generalize the creation of Bank instances but to restrict the use of constructors.
- We can make all constructors for the Bank class `protected`.
- Any code such as

  ```java
  Bank theBank = new Bank();
  ```

  would be **illegal** outside of the class Bank.
Accessing the Unique Instance of Bank

- We provide access to the unique instance of Bank by adding a class-level method to the Bank class declared as `static public Bank instance();`
- Since the method `instance()` is static, we'd write code such as `Bank theBank = Bank.instance();` to access the unique instance of the Bank class
- This is an example of a Creational Pattern known as the Singleton Pattern

3- Remote Access to Problem Domain Objects

- The Human Interface Components will be executing in a process on a different machine from the machine in which the Problem Domain Objects execute
- Each method that needs to be invoked across a network will need some kind of remote invocation mechanism (such as a remote procedure call)
- We can create a thin client-side object that encapsulates the mechanics of moving the call parameters to the receiving side (and returning the out parameters to the caller)
Example-Defining Remote Access to Account

- The ATM's client program needs to show the balance for any kind of account.
- Account has a method called `getBalance()` that accesses the balance attribute.
- Given the distributed access requirement, a concrete `Account` object, say an instance of `CheckingAccount`, is not in the same process as the client program.
- We define a class, `AccountProxy`, that encapsulates the communication required between the client and the real `Account` instance.

The AccountProxy Class

```java
class AccountProxy extends Account {
    public AccountProxy(String accountID);
    public double getBalance();
    //...

    //private internal data that AccountProxy needs
    //to find the real Account object
};
```
How AccountProxy is Used

Clients now deal with an AccountProxy, rather than with CheckingAccount or other concrete subclasses of Account:

```java
AccountProxy currentAccount = new AccountProxy(accountID);
double currentBalance = currentAccount.getBalance();
```

Extending the Problem Domain Class Diagram
4- Traversing Container Objects

- We want to be able to retrieve the Transaction "parts" of the container objects Account, for instance when we want to display the Transaction History for a particular account
- We do not want to expose the implementation mechanism
- We create a special class, which we will call TransactionIterator, to encapsulate access to the list of Transaction objects

The TransactionIterator Class

class TransactionIterator {
    public void first();
    public void next();
    public boolean isDone();
    public Transaction currentItem();
    // declare private data needed to keep track
    // of the current position in the list
}

Using the TransactionIterator Class

Account currentAccount;
// add some code to set currentAccount
// now create the Iterator
TransactionIterator accountIterator = currentAccount.createTransactionIterator();
// The for loop should be enclosed in
// a try-catch block
for (accountIterator.first(); !accountIterator.isDone(); accountIterator.next();)
{
  // do something with
  // accountIterator.currentItem()
}

Generalizing TransactionIterator

- TransactionIterator is an example of a design pattern called the Iterator Pattern
- TransactionIterator is in the category of behavioral design patterns because it allocates the behavior of traversing a structure to a specific class
Summary of the Case Study

- We have raised a series of design questions that arise from the requirements and preliminary analysis of the extended ATM system
- Each of these lends itself to solution that is described formally by a design pattern
- There will always be design questions for which a design pattern does not supply a ready solution - do not force design patterns into your design if they are not needed