OS Default Project:
A Simple Electronic Commerce System

Abstract:

Our distributed system attempts to model a distributed commerce application. Four clients are allowed to view inventory and buy from two stores. In order for the purchases to be approved the store must first check with the bank which then checks with the client to make sure the store and amount are correct. The approval process is asynchronous; this allows the clients to shop at other stores or make another order at the same store while the original order is making its way through the approval process.

The technology used to implement the Graphical User Interface is the Java Swing package. The underlying technology that facilitates the distributed objects is the Java RMI package.

Through this project we hope to gain a better understanding of the issues involved in distributed systems. Through our implementation we expect to learn more about Java and learn RMI by using it for the first time.

Design:

Conceptual design for the commercial simulation system began by identifying the actors involved in the system. In this case, one could easily obtain this information through simple lexographic analysis of the project specification presented for the default project. Clearly, the key players—stores, clients, and a bank—would have to be represented functionally in the code. Then, of course, actions would later be defined during the implementation design phase. Some of these actions included “authorizing an expenditure”, “purchase an item”, “verify an account”, and “confirm a request.” Subsequently, each actor was briefly discussed and possible attributes and collaborations between them noted.

Having built the framework for the design, implementation was the next consideration. This included a refinement of the conceptual design in the way of recognizing and pointing out smaller, more atomic entities that would be necessary that were not readily obvious during the initial conceptual design phase. Implementation proceeded with the design of the basic data structures and atomic entities. Following this, the project tended toward a halving of the implementation effort—the store and bank being one half and the client the other. Thus, the store / bank were developed concurrently with the client as this seemed to be the most logical and efficient approach. After necessary functionality was implemented for these main objects, a user interface was added to facilitate ease of use.

Interface design was approached differently for each of the logical partitions mentioned above. More energy was focused on the client interface design since this needed to interact with the general public and would be the driver behind this commercial simulation system. Implementation was concluded by enveloping the current project design within a Remote Methods Invocation (RMI) construct. Once this final stage was concluded, the design portion of the system was complete. Our intended design methodology was meant to model a wrapping or coating scheme—adding additional
layers incrementally after the current layer had been satisfactorily completed. The intention behind this scheme was to allow each step of the design to be its own conceptual module that would be clearly discernable from the other steps—thus hopefully facilitating a better product and more focused learning.

**Object Implementation:**

**Data Structures**

We defined three data structures, or atomic classes, for our project: CInventory, PurchaseOrder, and Transaction. The CInventory object was designed to represent an object in the store’s inventory list. It is the most atomic of all the objects as it is the basic unit in any commercial system. This simple object had as its data attributes all Java primitive data types. This data models what we believed to be the key attributes of an item to be purchased—the number available in inventory, the ID number, the purchase price per unit, and a brief textual description to allow the user to identify with the item in question. Basic “get” and “set” functions were provided for this object as well as a other various monotonous methods.

A PurchaseOrder was developed to represent a collection of objects a user would want to order as well as any summary or clerical information that would be associated with purchasing objects from a business. To implement this we used the Java Vector class to represent the list of items desired (reasons for this choice will be discussed later). In addition we felt the need to include as part of the PurchaseOrder, the users account number with the bank and a total order price. To simplify procedure calls, we provided Add(), Remove(), and RemoveAll() methods to interface with the calls to the Vector class and perform the clerical functions associated with maintaining a purchase order.

Our third and final data structure was the Transaction object. This object was the top level of an abstract view of what the user is trying to accomplish at an online commercial site. It is composed of only two things—the users PurchaseOrder and a transaction number associated with that order. We felt encapsulating these two items together would create a more meaningful representation of a real world user-system interaction.

Similar to our design process, we attempted to use this idea of aggregation when creating these composite objects to reinforce the idea of layering and atomic modules. In this particular case, the problem seemed to render itself well to this method.

**Key classes**

In our implementation of the key classes—Store, Bank, and Client—we tried to represent a “passing the torch” calling approach presented in the problem statement in our design of the methods. Unfortunately, as a result, it is nearly impossible to describe the function of one key class without somehow mentioning the others. Thus, below is a narrative detailing the data and methods of each key class and how each key class is involved in the system process.

Initiation of activity starts when the client calls methods of the Store object. This object contains three data items: a list of current inventory, a list of outstanding transactions, and an identification number. The purpose of each of these data items should be evident. Methods available for the store object include: viewInventory(), purchase(), and confirm(). On initiation, on of the store’s responsibilities includes populating the inventory list. For simplicity we implemented this as a simple text file although in a less esoteric application, one would want to consider a database for the inventory back-end. Clients initiate activity when they call viewInventory() to receive a list of the current inventory of the store. Once a purchase order has been made and the client is ready to proceed, they call the Store’s purchase() method passing it the PurchaseOrder. The store then
computes a unique transaction number associated with that user, the time the order was placed, and the store involved. Once availability of all the items on the purchase order has been satisfied, the Store attempts to verify the user’s account with the Bank by calling the Bank’s confirmRequest() function. The confirm() function of the Store is called by the Bank and is designed to handle the response to the confirmRequest() call. If the Bank authorizes the account, this function’s role is to process the transaction and notify the user of success or failure by calling the Client’s purchaseNotification() function.

The Bank contains lists that are populated with the client account numbers that do business with the bank and the store identification numbers. The Bank’s main function is to act as a mediator between the client and the store. It’s simplicity is evident by having only two, albeit very critical, functions. The confirmRequest() function is called by the Store to confirm with the Bank that the account number given by the client truly exists. This function performs the requested lookup and then, in turn, calls the Client’s confirmRequest() function. This is one step in the verification process of the transaction. The next function, confirm(), is a method that is called by the client in response to the Bank’s confirmRequest() call above. This method receives the user’s confirmation of whether to actually perform the transaction, looks up the store ID number associated with the transaction, and notifies the store of the user’s response with a call to the Store’s confirm() function.

Both of the Client’s main functions have already been mentioned in the dialog above. The confirmRequest() matches the transaction number associated with the purchase order with a transaction number on a list of pending transactions. If there is a match, then the transaction is allowed to proceed. Otherwise, the user does not give permission for the transaction to progress. The purchaseNotification() function merely serves as an informant to the user as to whether the transaction that it tried to place actually succeeded, or for some reason failed. The PlaceOrder() function provides the initiation of the process and represents the user’s action of actually placing an order.

Consider the following activity outline and illustration:

1. Client calls Store.viewInventory()
2. Client calls Store.purchase()
3. Store calls Bank.confirmRequest()
4. Bank calls Client.confirmRequest()
5. Client calls Bank.confirm()
6. Bank calls Store.confirm()
7. Store calls Client.purchaseNotification()
8. Transaction is complete!

In the second stage of the conceptual design, we discovered the need for another class in our implementation—a central, overseeing object to assist in resolving object names and addresses. We developed this idea as an extension of the real-world Domain Name Server and called it CNameServer. Its responsibilities are singular in nature and it’s functions are self-documenting. Its purpose is to provide to the other key classes a way of resolving names and addresses by maintaining
and registering the names of each of the objects as they enter the system. This object serves as a
director or interpreter, to help guide object messages to the correct location.

While this method of calling provides a coherent model of the process involved, it does not solely
provide for the asynchronization between store and client required by the project specification. This
is desirable because it allows the client to shop at other stores while waiting for the first transaction to
process. In other words, it prevents the client from blocking while placing an order at a store.

Technology utilized:

RMI

Java introduced a Remote Method Invocation architecture with the release of JDK 1.1. We felt that
this technology would be ideal for the class of application that we were attempting to develop. One of
its main objectives is to allow remote method invocations on a distributed system through an
interface with the “look and feel” of a non-distributed system. This fit our distributed design needs
perfectly. Additionally, RMI handles many of the more complex tasks involved with this type of
programming—including addressing, synchronization, etc.

When wrapping our program with an RMI layer we simply provided a Client, Server, and
Implementation version of each of the key classes—that is, for each class that would need distributed-
object computing capability. We included both Server and Client versions because in this instance,
each of the main classes would at some point in its life cycle necessitate serving in either of those
capacities. The majority of the RMI functionality located within the key classes was relegated to a
function called InitNetwork() in each of these classes. This functions responsibility was to initiate
network function by requesting the IP address of the DNS model from the user and communicating
with this object to resolve class names involved in the distributed function calling.

Threads

To implement our synchronization we introduced threads into our code. We instantiated two
thread objects for each user (client)--one for each store. In this way, we could allow a client to place
an order to once store without worrying about it’s blocking preventing the user from placing an order
at another store. This also addresses the need for synchronization in the inventory. Because this
method is asynchronous between the user and the store, but synchronous in each store, it prevents
concurrent access to the inventory list preventing problems similar to lost update.

Constraints and Limiting Factors:

Several constraints had to be considered when designing our system. One of the major constraints
involved the very core activity of communicating among these distributed objects. This affected our
design decisions in our choices for data types as well as our overall design structure. In order to
ensure that communication would be possible in this distributed environment, it was necessary to
guarantee that the objects and their subsequent data members could be serialized. We also had to
consider the issue of synchronization. From this concern grew our “passing the torch” idea. We could
not use simple function return values, nor could we rely on argument variables to transport our
information due to synchronization/blocking issues. This need led to our design of sets of
complementary functions that would simply perform small steps of the verification process
incrementally using threads as an catalyst. Not only does this alleviate the blocking issues—since
each single step is its own operation performed on a single store, but once clearly communicated,
offers a comprehensive and easily understandable representation of the problem.
Probably our biggest obstacle involving this project was the language and the RMI architecture. Neither project member had previous knowledge of Java. This made creating the project a bit more difficult than usual considering the members had to include the overhead of language difficulties with normal design issues. More imposing was the RMI implementation. RMI provides an invaluable resource for distributed object computing. However, the implementation is not always as straightforward as other methodologies. Finding working examples and clear understandable documentation was not as easy as we had expected. These two problems posed the major set-backs for the project implementation and design.

Summary:

We learned about the issues involved in distributed computing. Our system was challenging enough to build in a semester, but it opened the door to a number of other issues that we could appreciate from dealing with our small simple model. For example, how would we deal with an infinite amount of clients? And what would happen if those clients were prone to disconnection problems while waiting for purchase approval? Issues such as naming, network failure, security, and load balancing were easier to understand and appreciate their complexity.

In addition to the distributed issues, we obviously learned more about the Java language, RMI, and Swing technology. As mentioned, before this semester neither of us had experience in any of these areas.

It is easy to see how our project model could be extended into practical applications. When thinking about such an endeavor it is impossible to avoid dealing with issues such as network failure, security, etc. Our architecture is functional, but it is not close to achieving the robustness and flexibility that would be needed in a commercial system. In a real commerce system it is hard to imagine a system with exactly four clients, two stores, and one bank at all times. Also network failure and recovery would be a constant issue to be dealt with.

Illustrations and Diagrams:

Case Relationships
Module class Diagrams

```
<interface>
BankImpl
<interface>
StoreImpl

JFrame fame
JSplitPane pane;
JTextArea clientOut;
JTextArea bankOut
Bank bank
Vector clients

void initNetWork()
void popInventory()

StoreImpl

StoreImpl

StoreImpl

<<interface>>
BankImpl

<<interface>>
Store

<<interface>>
Bank

vector viewInventory()
void confirm(bool authorized, int
int purchase(PurchaseOrder ord)
```
**Data Structures**

**COrderPanel**

- JComboBox storeSelect
- JButton PlaceOrder,
- JButton ClearOrder,
- JButton PreviewOrder,
- JButton AddItem,
- JTextField quantity,
- JTextField itemNum,
- Vector currentInventory,

String getStore();
void ClearOrder()
void AddCurrentToOrder()
void PlaceOrder()
void PreviewOrder();