23

PROXY

This pattern was previously described in GoF95.

DESCRIPTION

Let us consider the following code sample:

```
//Client
class Customer{
  public void someMethod(){
    //Create the Service Provider Instance
    FileUtil futilObj=new FileUtil();
    //Access the Service
    futilObj.writeToFile("Some Data");
  }
}
```

As part of its implementation, the `Customer` class creates an instance of the `FileUtil` class and directly accesses its services. In other words, for a client object, the way of accessing a `FileUtil` object is fairly straightforward. From the implementation it seems to be the most commonly used way for a client object to access a service provider object. In contrast, sometimes a client object may not be able to access a service provider object (also referred to as a target object) by normal means. This could happen for a variety of reasons depending on:

- *The location of the target object* — The target object may be present in a different address space in the same or a different computer.
- *The state of existence of the target object* —The target object may not exist until it is actually needed to render a service or the object may be in a compressed form.
- *Special Behavior* —The target object may offer or deny services based on the access privileges of its client objects. Some service provider objects may need special consideration when used in a multithreaded environment.

In such cases, instead of having client objects to deal with the special requirements for accessing the target object, the Proxy pattern suggests using a separate object referred to as a *proxy* to provide a means for different client objects to access the target object in a normal, straightforward manner.

The `Proxy` object offers the same interface as the target object. The `Proxy` object interacts with the target object on behalf of a client object and takes care of the specific details of communicating with the target object. As a result, client objects are no longer needed to deal with the special requirements for accessing the services of the target object. A client can call the `Proxy` object through its interface and the `Proxy` object in turn forwards those calls to the target object. Client objects need not even know that they are dealing with `Proxy` for the original object. The `Proxy` object hides the fact that a client object is dealing with an object that is either remote, unknown whether instantiated or not, or needs special authentication. In other words, a `Proxy` object serves as a transparent bridge between the client and an inaccessible remote object or an object whose instantiation may have been deferred.

`Proxy` objects are used in different scenarios leading to different types of proxies. Let us take a quick look at some of the proxies and their purpose.

Note: Table 23.1 lists different types of `Proxy` objects. In this chapter, only the remote proxy is discussed in detail. Some of the other proxy types are discussed as separate patterns later in this book.

PROXY VERSUS OTHER PATTERNS

From the discussion of different `Proxy` objects, it can be observed that there are two main characteristics of a `Proxy` object:

- It is an intermediary between a client object and the target object.
- It receives calls from a client object and forwards them to the target object.

In this context, it looks very similar to some of the other patterns discussed earlier in this book. Let us see in detail the similarities and differences between the Proxy pattern and some of the other similar patterns.

Proxy versus Decorator

- Proxy
  - The client object cannot access the target object directly.
  - A `proxy` object provides access control to the target object (in the case of the protection proxy).
  - A `proxy` object does not add any additional functionality.
- Decorator
  - The client object does have the ability to access the target object directly, if needed.
  - A `Decorator` object does not control access to the target object.
  - A `Decorator` adds additional functionality to an object.

Table 23.1 List of Different Proxy Types

Proxy Type	*Purpose*
Remote Proxy	To provide access to an object located in a different address space.
Virtual Proxy	To provide the required functionality to allow the on-demand creation of a memory intensive object (until required).
Cache Proxy/Server Proxy	To provide the functionality required to store the results of most frequently used target operations. The proxy object stores these results in some kind of a repository. When a client object requests the same operation, the proxy returns the operation results from the storage area without actually accessing the target object.
Firewall Proxy	The primary use of a firewall proxy is to protect target objects from bad clients. A firewall proxy can also be used to provide the functionality required to prevent clients from accessing harmful targets.
Protection Proxy	To provide the functionality required for allowing different clients to access the target object at different levels. A set of permissions is defined at the time of creation of the proxy. Subsequently, those permissions are used to restrict access to specific parts of the proxy (in turn of the target object). A client object is not allowed to access a particular method if it does not have a specific right to execute the method.
Synchronization Proxy	To provide the required functionality to allow safe concurrent accesses to a target object by different client objects.
Smart Reference Proxy	To provide the functionality to prevent the accidental disposal/deletion of the target object when there are clients currently with references to it. To accomplish this, the proxy keeps a count of the number of references to the target object. The proxy deletes the target object if and when there are no references to it.
Counting Proxy	To provide some kind of audit mechanism before executing a method on the target object.

Proxy versus Façade

- Proxy
  - A `Proxy` object represents a single object.
  - The client object cannot access the target object directly.
  - A `Proxy` object provides access control to the single target object.

- Façade
  - A `Façade` object represents a subsystem of objects.
  - The client object does have the ability to access the subsystem objects directly, if needed.
  - A `Façade` object provides a simplified higher level interface to a subsystem of components.

Proxy versus Chain of Responsibility

- Proxy
  - A `Proxy` object represents a single object.
  - Client requests are first received by the `Proxy` object, but are never processed directly by the `Proxy` object.
  - Client requests are always forwarded to the target object.
  - Response to the request is guaranteed, provided the communication between the client and the server locations is working.
- Chain of Responsibility
  - Chain can contain many objects.
  - The object that receives the client request first could process the request.
  - Client requests are forwarded to the next object in the chain only if the current receiver cannot process the request.
  - Response to the request is not guaranteed. It means that the request may end up reaching the end of the chain and still might not be processed.

In Java, the concept of Remote Method Invocation (RMI) makes extensive use of the Remote Proxy pattern. Let us take a quick look at the concept of RMI and different components that facilitate the RMI communication process.

RMI: A QUICK OVERVIEW

RMI enables a client object to access remote objects and invoke methods on them as if they are local objects (Figure 23.1).

RMI Components

The following different components working together provide the stated RMI functionality:

- *Remote Interface* — A remote object must implement a remote interface (one that extends `java.rmi.Remote`). A remote interface declares the

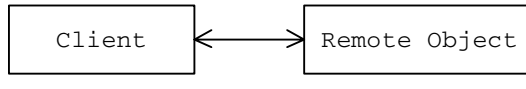

Figure 23.1 Client's View of Its Communication with a Remote Object Using RMI

methods in the remote object that can be accessed by its clients. In other words, the remote interface can be seen as the client's view of the remote object.

Requirements:

- Extend the `java.rmi.Remote` interface.
- All methods in the remote interface must be declared to throw `java.rmi.RemoteException` exception.

- *Remote Object* — A remote object is responsible for implementing the methods declared in the associated remote interface.

Requirements:

- Must provide implementation for a remote interface.
- Must extend `java.rmi.server.UnicastRemoteObject`.
- Must have a constructor with no arguments.
- Must be associated with a server. The server creates an instance of the remote object by invoking its zero argument constructor.

- *RMI Registry* — RMI registry provides the storage area for holding different remote objects.
  - A remote object needs to be stored in the RMI registry along with a name reference to it for a client object to be able to access it.
  - Only one object can be stored with a given name reference.
- *Client* — Client is an application object attempting to use the remote object.
  - Must be aware of the interface implemented by the remote object.
  - Can search for a remote object using a name reference in the RMI Registry. Once the remote object reference is found, it can invoke methods on this object reference.
- *RMIC: Java RMI Stub Compiler* — Once a remote object is compiled successfully, RMIC, the Java RMI stub compiler can be used to generate *stub and skeleton* class files for the remote object. Stub and skeleton classes are generated from the compiled remote object class. These stub and skeleton classes make it possible for a client object to access the remote object in a seamless manner.

The following section describes how the actual communication takes place between a client and a remote object.

RMI Communication Mechanism

In general, a client object cannot directly access a remote object by normal means. In order to make it possible for a client object to access the services of a remote object as if it is a local object, the RMIC-generated stub of the remote object class and the remote interface need to be copied to the client computer.

The *stub* acts as a *(Remote) proxy* for the remote object and is responsible for forwarding method invocations on the remote object to the server where the actual remote object implementation resides. Whenever a client references the remote object, the reference is, in fact, made to a local stub. That means, when a client makes a method call on the remote object, it is first received by the local stub instance. The stub forwards this call to the remote server. On the server the RMIC generated skeleton of the remote object receives this call.

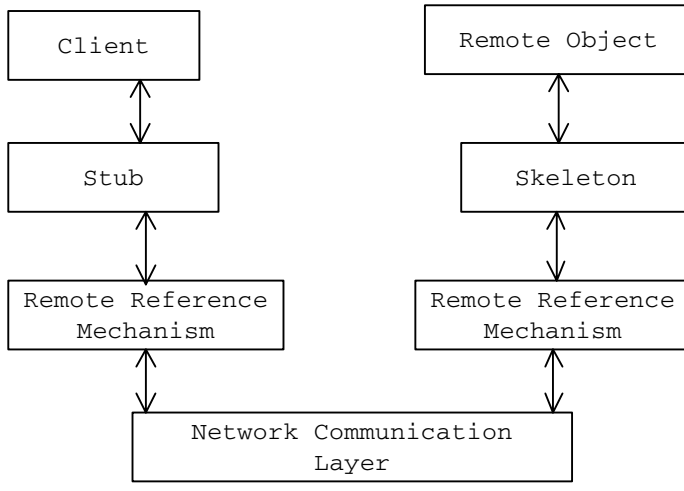

Figure 23.2 The Actual RMI Communication Process

The skeleton is a server side object and it *does not* need to be copied to the client computer. The *skeleton* is responsible for dispatching calls to the actual remote object implementation. Once the remote object executes the method, results are sent back to the client in the reverse direction.

Figure 23.2 shows the actual RMI communication process.

For more information on the Java RMI technology, I recommend reading the RMI tutorial at `java.sun.com`.

RMI AND PROXY PATTERN

It can be seen from the RMI communication discussion that the stub class, acting as a remote proxy for the remote object, makes it possible for a client to treat a remote object as if it is available locally. Thus, any application that uses RMI contains an implicit implementation of the Proxy pattern.

EXAMPLE

During the discussion of the Façade pattern, we built a simple customer data management application to validate and save the input customer data. Our design consisted of a set of three subsystem classes — `Account`, `Address` and `CreditCard` — representing different parts of the customer data.

Before applying the Façade pattern, the client `AccountManager` was designed to directly interact with the three subsystem classes to validate and save the customer data. Applying the Façade pattern, we defined a `CustomerFacade` `Façade` object to deal with the three subsystem classes on behalf of the client `AccountManager` (Figure 23.3).

In this application, both the subsystem components and the `Façade` object are local to the `AccountManager` client object.

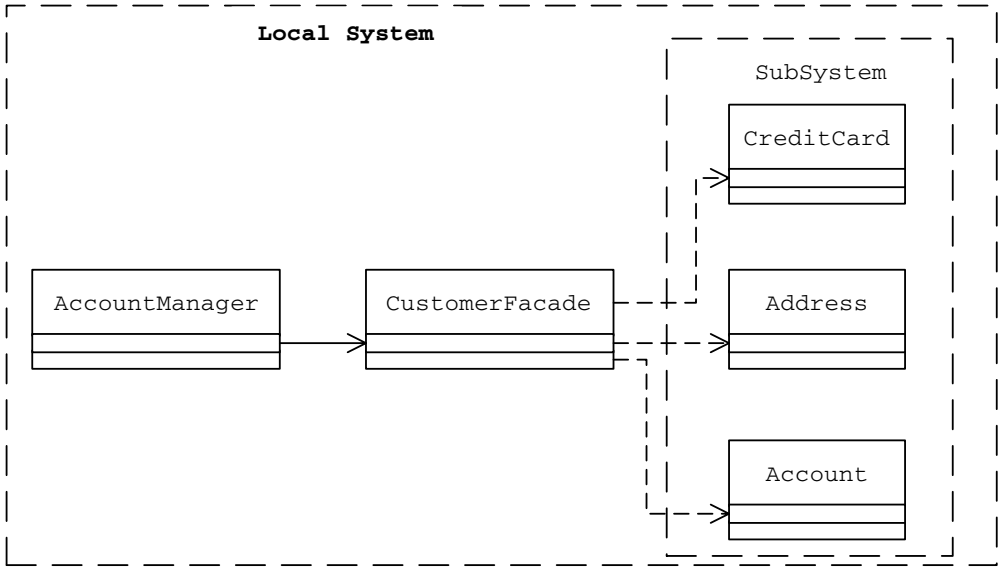

Figure 23.3 Customer Data Management Application for the Local Mode of Operation: Class Association

Let us build a different version of the same application that runs in the remote mode. In the remote mode, the application makes use of remote objects using the Java RMI technology.

In designing the application for the remote mode of operation, we would move all of the subsystem components (`Account`, `Address` and `CreditCard`) and the Façade (`CustomerFacade`) to a remote server (Figure 23.4) with the following advantages:

- Objects on the server can be shared by different client applications. Clients no longer have to maintain local copies of these classes and hence clients will be light-weighted.
- Leads to centralized control over processes involving changes, enhancements and monitoring.

Let us start designing our customer data management application for the remote mode of operation using the RMI technology.

As the first step, let us define a remote interface `CustomerIntr` that:

- Declares the methods to be implemented by the Façade
- Declares all such methods to throw the `RemoteException` exception
- Extends the built-in `java.rmi.Remote` interface

```
public interface CustomerIntr extends java.rmi.Remote {
  void setAddress(String inAddress) throws RemoteException;
  void setCity(String inCity) throws RemoteException;
```

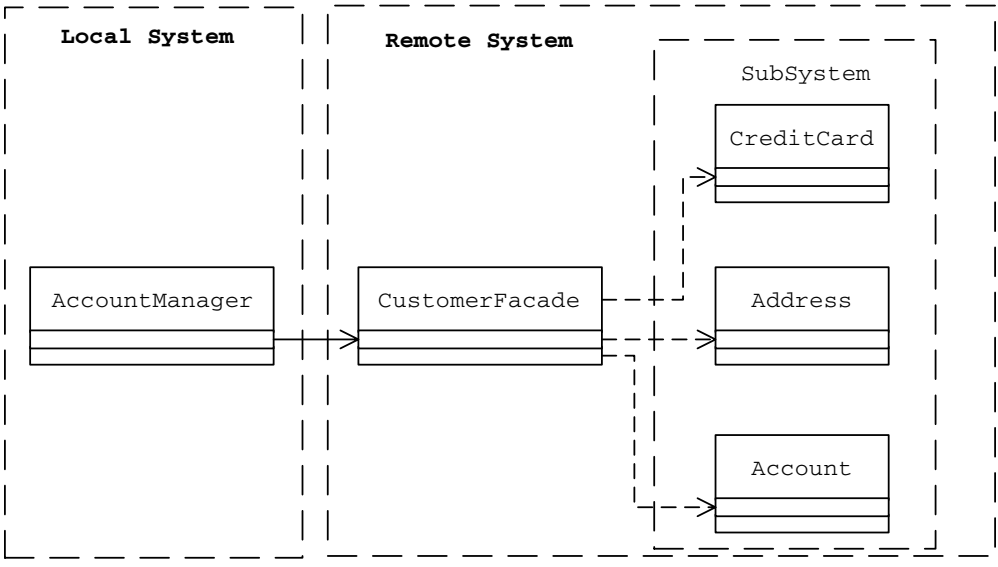

Figure 23.4 Customer Data Management Application for the Remote Mode of Operation: Class Association

```
    void setState(String inState) throws RemoteException;
    void setFName(String inFName) throws RemoteException;
    void setLName(String inLName) throws RemoteException;
    void setCardType(String inCardType) throws RemoteException;
    void setCardNumber(String inCardNumber)
      throws RemoteException;
    void setCardExpDate(String inCardExpDate)
      throws RemoteException;
    boolean saveCustomerData() throws RemoteException;
}
```

Let us redesign the `CustomerFacade` Façade class (Listing 23.1) so that it now implements the `CustomerIntr` remote interface. Different client objects can interface with the subsystem objects by invoking the `CustomerIntr` methods on the concrete `CustomerFacade`. Figure 23.5 shows the structure and the association between the `CustomerFacade` and the remote interface `CustomerIntr` it implements.

Because the subsystem components are local to the `CustomerFacade` class, it continues to refer to them as local objects without any changes in the way it instantiates and invokes methods on them. When executed, the `CustomerFacade` creates an instance of itself and keeps it in the RMI registry with a reference name. Client objects will be able to obtain this copy of the remote object using the reference name.

Listing 23.1 `CustomerFacade` Class: Revised

```
public class CustomerFacade extends UnicastRemoteObject
  implements CustomerIntr {
  private String address;
  private String city;
  private String state;
  private String cardType;
  private String cardNumber;
  private String cardExpDate;
  private String fname;
  private String lname;
  public CustomerFacade() throws RemoteException {
    super();
    System.out.println("Server object created");
  }
  public static void main(String[] args) throws Exception {
    String port = "1099";
    String host = "localhost";
    //Check for hostname argument
    if (args.length == 1) {
      host = args[0];
    }
    if (args.length == 2) {
      port = args[1];
    }
    if (System.getSecurityManager() == null) {
      System.setSecurityManager(new RMISecurityManager());
    }
    //Create an instance of the server
    CustomerFacade facade = new CustomerFacade();
    //Bind it with the RMI Registry
    Naming.bind("//" + host + ":" + port + "/CustomerFacade",
                facade);
    System.out.println("Service Bound…");
  }
  public void setAddress(String inAddress)
    throws RemoteException {
    address = inAddress;
  }
```

(continued)

Listing 23.1 `CustomerFacade` Class: Revised (Continued)

```
public void setCity(String inCity)
   throws RemoteException{ city = inCity;
}
public void setState(String inState)
   throws RemoteException{ state = inState;
}
public void setFName(String inFName)
   throws RemoteException{ fname = inFName;
}
public void setLName(String inLName)
   throws RemoteException{ lname = inLName;
}
public void setCardType(String inCardType)
   throws RemoteException {
   cardType = inCardType;
}
public void setCardNumber(String inCardNumber)
   throws RemoteException {
   cardNumber = inCardNumber;
}
public void setCardExpDate(String inCardExpDate)
   throws RemoteException {
   cardExpDate = inCardExpDate;
}
public boolean saveCustomerData() throws RemoteException{
   Address objAddress;
   Account objAccount;
   CreditCard objCreditCard;
   /*
    client is transparent from the following
    set of subsystem related operations.
   */
   boolean validData = true;
   String errorMessage = "";
   objAccount = new Account(fname, lname);
   if (objAccount.isValid() == false) {
    validData = false;
    errorMessage = "Invalid FirstName/LastName";
   }
```

(continued)

Listing 23.1 `CustomerFacade` Class: Revised (Continued)

```
        objAddress = new Address(address, city, state);
        if (objAddress.isValid() == false) {
          validData = false;
          errorMessage = "Invalid Address/City/State";
        }
        objCreditCard = new CreditCard(cardType, cardNumber,
                        cardExpDate);
        if (objCreditCard.isValid() == false) {
          validData = false;
          errorMessage = "Invalid CreditCard Info";
        }
        if (!validData) {
          System.out.println(errorMessage);
          return false;
        }
        if (objAddress.save() && objAccount.save() &&
            objCreditCard.save()) {
          return true;
        } else {
          return false;
        }
      }
    }
```

Because a client does not need to access any of the subsystem components directly, none of the subsystem components undergoes any changes in the new design for the remote mode of operation of the application.

Let us redesign the client `AccountManager` class (Listing 23.2).

Similar to the local mode of operation, `AccountManager` displays the necessary user interface to accept the input customer data (Figure 23.6). When the user enters the data and clicks on the `Validate & Save` button, it retrieves the remote object reference from the RMI registry using the reference name.

Once the remote object reference is retrieved from the registry, the client can invoke operations on the remote object reference as if it is a local object. Figure 23.7 depicts this behavior.

Note that the stub class corresponding to the compiled `CustomerFacade` class must be copied onto the client `AccountManager` location before executing the application. After the `CustomerFacade` is compiled, the stub and skeleton classes can be generated using the RMIC compiler on the compiled `CustomerFacade` class. Detailed instructions on compiling and deploying different application components are provided under the following "Additional Notes" section.

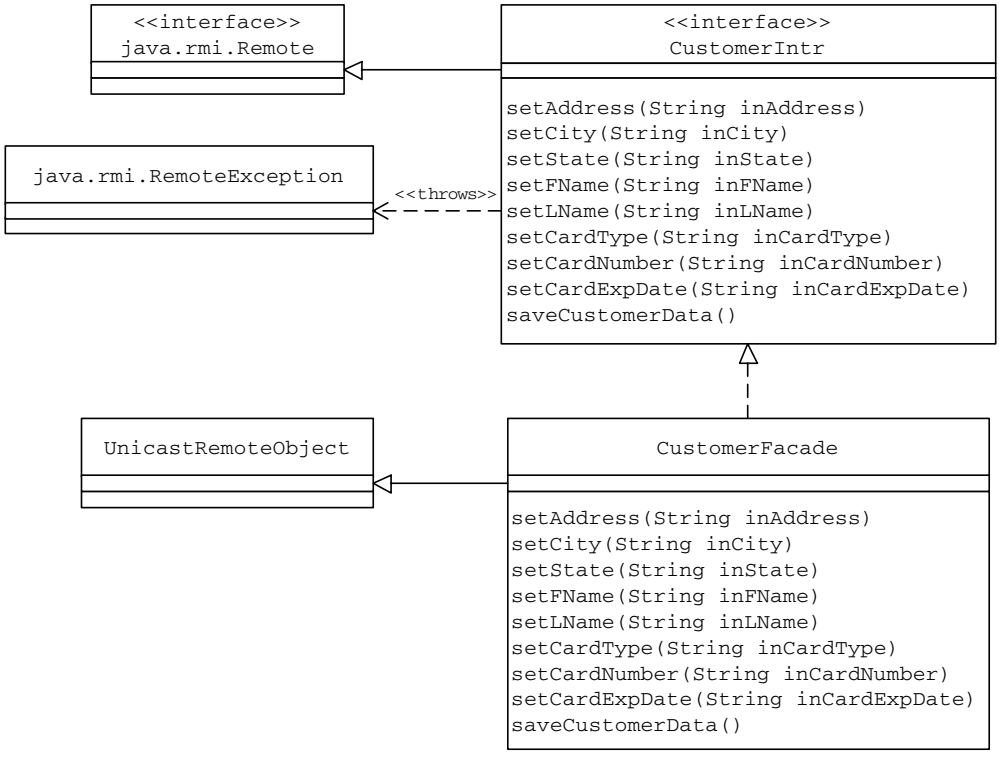

Figure 23.5 `Façade` Design: Remote Mode of Operation

In reality, when the client invokes a method such as `saveCustomerData` on the `CustomerFacade` remote object, the `CustomerFacade_stub` object, which is local to the client, first receives it. The `CustomerFacade_stub` then transmits the method call to the server for processing.

On the server side the `CustomerFacade_skel` is responsible for receiving the method call through the lower levels of the communication network. It then dispatches it to the actual `CustomerFacade` object on the server. In case of the `saveCustomerData` method, the `CustomerFacade` object creates the necessary subsystem objects and invokes the required methods on these objects to validate and save the customer data. The result of the processing is carried back to the client in the reverse manner. Figure 23.8 depicts this actual communication mechanism.

As can be seen from above, the `CustomerFacade_stub` class enables the client object to invoke methods on the remote `CustomerFacade` object as if it is present locally, which, otherwise, is not accessible by normal means. Thus the stub functions as a remote proxy.

Listing 23.2 `AccountManager` Class: Revised

```
            …
            …
    public void actionPerformed(ActionEvent e) {
            …
            …
      if (e.getActionCommand().equals(
            AccountManager.VALIDATE_SAVE)) {
        //get input values
        String firstName = objAccountManager.getFirstName();
        String lastName = objAccountManager.getLastName();
        String address = objAccountManager.getAddress();
            …
            …
        try {
          //Call registry for AddOperation
          facade = (CustomerIntr) Naming.lookup ("rmi://" +
                    objAccountManager.getRMIHost() + ":" +
                    objAccountManager.getRMIPort() +
                    "/CustomerFacade");
          facade.setFName(firstName);
          facade.setLName(lastName);
          facade.setAddress(address);
              …
              …
          //Client is not required to access subsystem components.
          boolean result = facade.saveCustomerData();
          if (result) {
            validateCheckResult =
              " Valid Customer Data: Data Saved Successfully ";
          } else {
            validateCheckResult =
              " Invalid Customer Data: Data Could Not Be Saved ";
          }
```

(continued)

Listing 23.2 `AccountManager` Class: Revised (Continued)

```
    } catch (Exception ex) {
      System.out.println(
        "Error: Please check to ensure the " +
        "remote server is running" +
        ex.getMessage());
    }
    objAccountManager.setResultDisplay(
      validateCheckResult);
    }
  }
        …
        …
```

ADDITIONAL NOTES

Compilation and Deployment Notes

Download the source code from the following Web site: http://www.crcpress.com/e_products/downloads/download.asp.

1. Compile all Java files in the *Proxy/Server* folder.
2. Execute the following command from the *Proxy/Server* folder:

   ```
   Rmic CustomerFacade
   ```

   This command invokes the RMI stub compiler and creates the stub and skeleton classes `CustomerFacade_Skel.class` and `CustomerFacade_Stub.class`, respectively.
3. Copy the following files from the *Proxy/Server* folder to the *Proxy/Client* folder:

   ```
   CustomerIntr.class
   CustomerFacade_Stub.class
   ```

4. Compile all Java files in the *Proxy/Client* folder.
5. Start the `rmiregistry`:

   ```
   start rmiregistry <objectRegistryPort> (Windows)
   rmiregistry & (Solaris)
   ```

   - `<objectRegistryPort>` – `This` is where the RMI registry needs to listen. The default port value is 1099.
   - Example:

     ```
     start rmiregistry
     ```

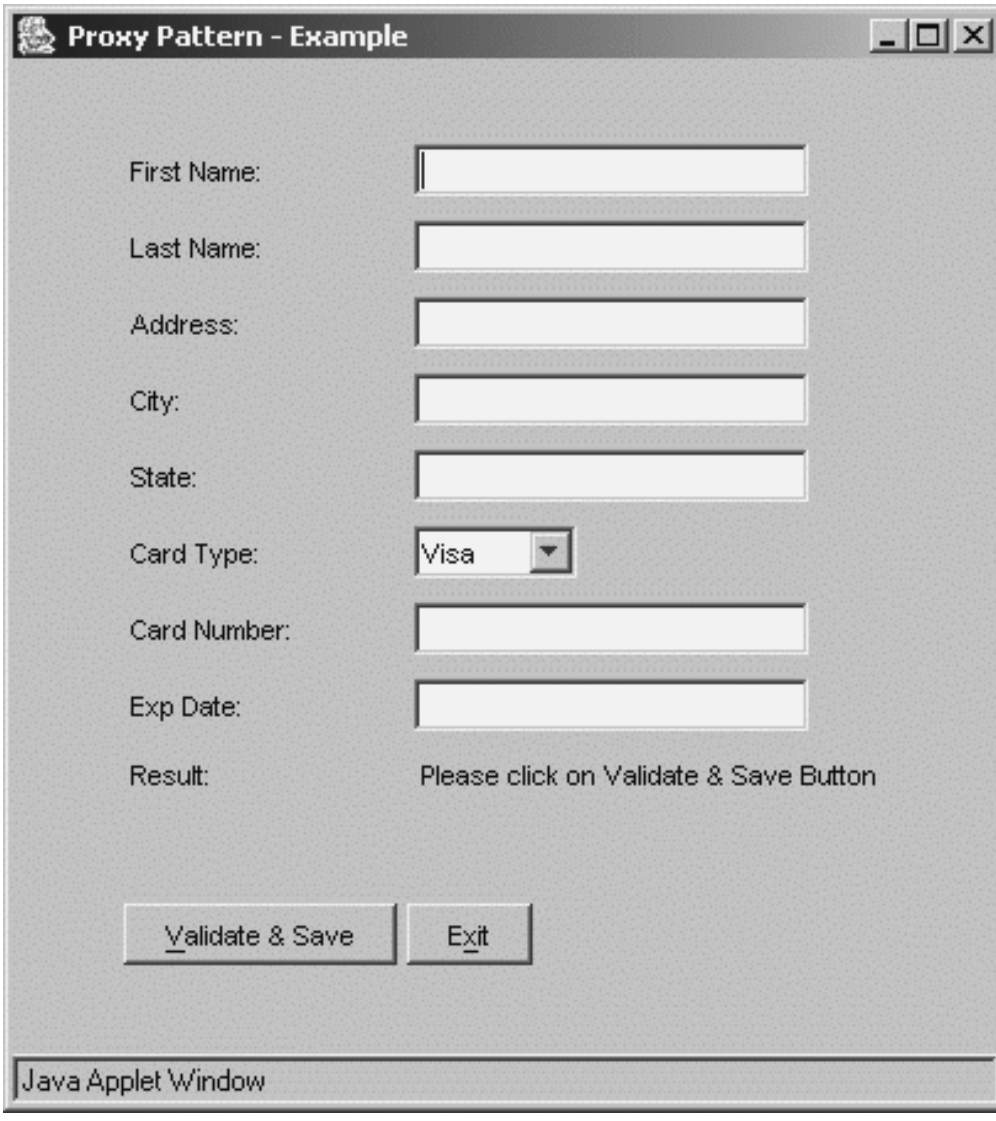

Figure 23.6 The User Interface: Remote Mode of Operation

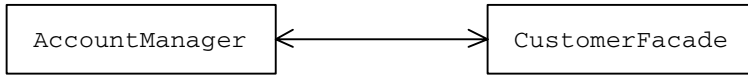

Figure 23.7 `AccountManager` View of Its Communication with the Remote `CustomerFacade`

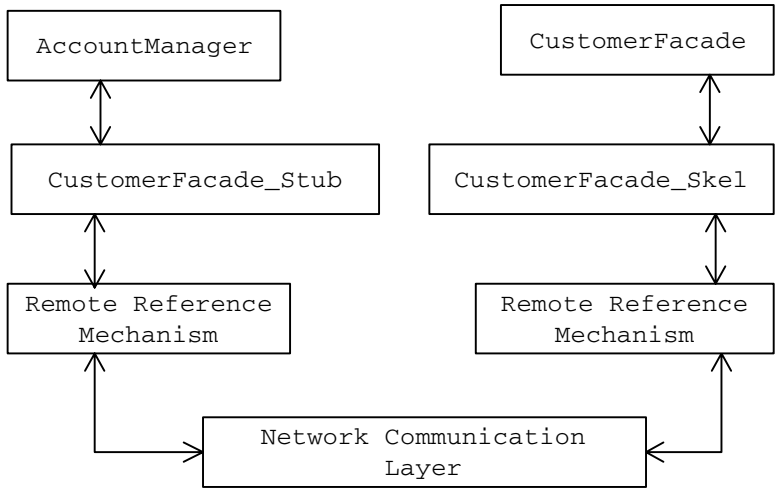

Figure 23.8 The Actual Flow of Communication

6. Run the following command:

```
java -Djava.security.policy=<PolicyFile> CustomerFacade
<RemoteRegistryHost> <RemoteRegistryPort>
```

   - Example:

```
java -Djava.security.policy=java.policy CustomerFacade
localhost 1099
```

   - `<policyFile>` – This is the name of the security file with permissions set for the application. The location of the file in the file system of the underlying operating system needs to be specified.
     Note: The `java.policy` policy file is available in the server folder.
   - `<RemoteRegistryHost>` – This is the DNS (Domain Name System) name or the IP address of the host machine where the object registry is running. For the same computer, use "localhost."
   - `<RemoteRegistryPort>` – This is the port where the object registry is listening on the specified `RemoteRegistryHost`. The default is 1099.

7. The following output will be displayed:

```
Server object created
Service bound...
```

8. Go to the folder *Proxy/client* and execute the following command to run the client:

```
java -Djava.security.policy=<PolicyFile> AccountManager
<RemoteRegistryHost> <RemoteRegistryPort>
```

- Example:

```
java -Djava.security.policy=java.policy AccountManager
localhost 1099
```

- `<policyFile>` – This is the name of the security file with permissions set for the application. The location of the file in the file system of the underlying operating system needs to be specified.
  Note: The `java.policy` policy file is available in the client folder.
- `<RemoteRegistryHost>` – This is the DNS name or the IP address of the host machine where the object registry is running. For the same computer, use "localhost."
- `<RemoteRegistryPort>` – This is the port where the object registry is listening on the specified `RemoteRegistryHost`. The default is 1099.

This executes the client `AccountManager` and the user interface will be displayed.

PRACTICE QUESTIONS

1. In our example design, a client can access only the `CustomerFacade` remote object. The `CustomerFacade` internally interacts with the remote subsystem components directly. But a client cannot access any of the subsystem components (`Account, Address` or the `CreditCard`). Make necessary changes to the `Account, Address` and the `CreditCard` classes and to the deployment process, to enable a client to access these subsystem components directly without having to go through the `CustomerFacade`.
2. Design and implement the purchase request Façade as a remote object. (Refer to Practice Questions 1 and 2 of Chapter 22 — Façade.)