PROXY PATTERN

- Toni Linares -

CONTEXT I

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

CONTEXT II

In such cases, 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

- Client objects need not even know that they are dealing with Proxy for the original object.

-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 VS DECORATOR

Proxy

– The client object cannot access the target object directly.
– A proxy object provides access control to the target object 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 if needed.
– A Decorator object does not control access to the target
– A Decorator adds additional functionality to an object.

PROXY VS 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 VS 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.

REMOTE PROXY : RMI

- In Java, the concept of Remote Method Invocation (RMI) makes extensive use of the Remote Proxy pattern.

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

- Any application that uses RMI contains an implicit implementation of the Proxy pattern.

RMI COMPONENTS I

Remote Interface — A remote object must implement a remote interface (one that extends java.rmi.Remote).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 COMPONENTS II

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.

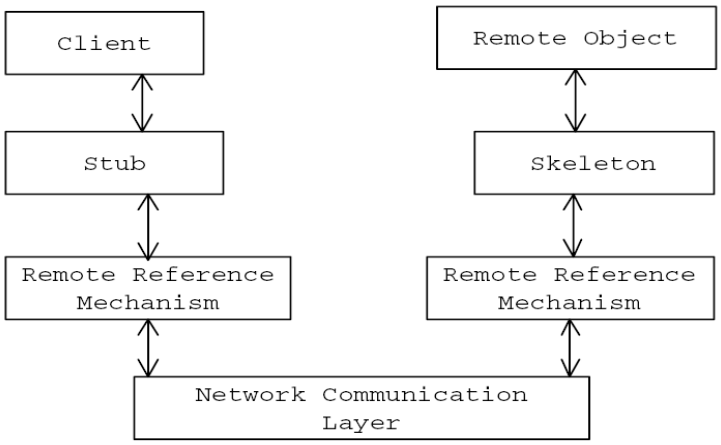
RMIC: Java RMI Stub Compiler — Once a remote object is compiled successfully, RMIC, the Java RMI stub compiler can be used to generate *stub and skeleton* class files for the remote object. Stub and skeleton classes are generated from the compiled remote object class. These stub and skeleton classes make it possible for a client object to access the remote object in a seamless manner.

RMI COMMUNICATION MECHANISM I

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.

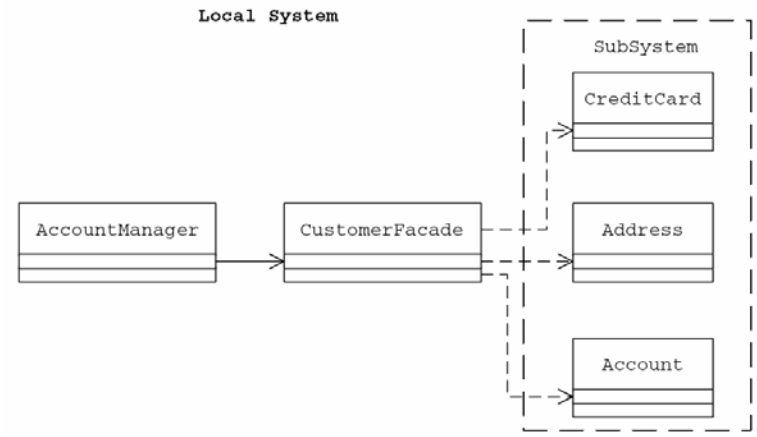
RMI COMMUNICATION MECHANISM II



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.

EXAMPLE

During the discussion of the Façade pattern, we built a simple customer data management application to validate and save the input customer data.



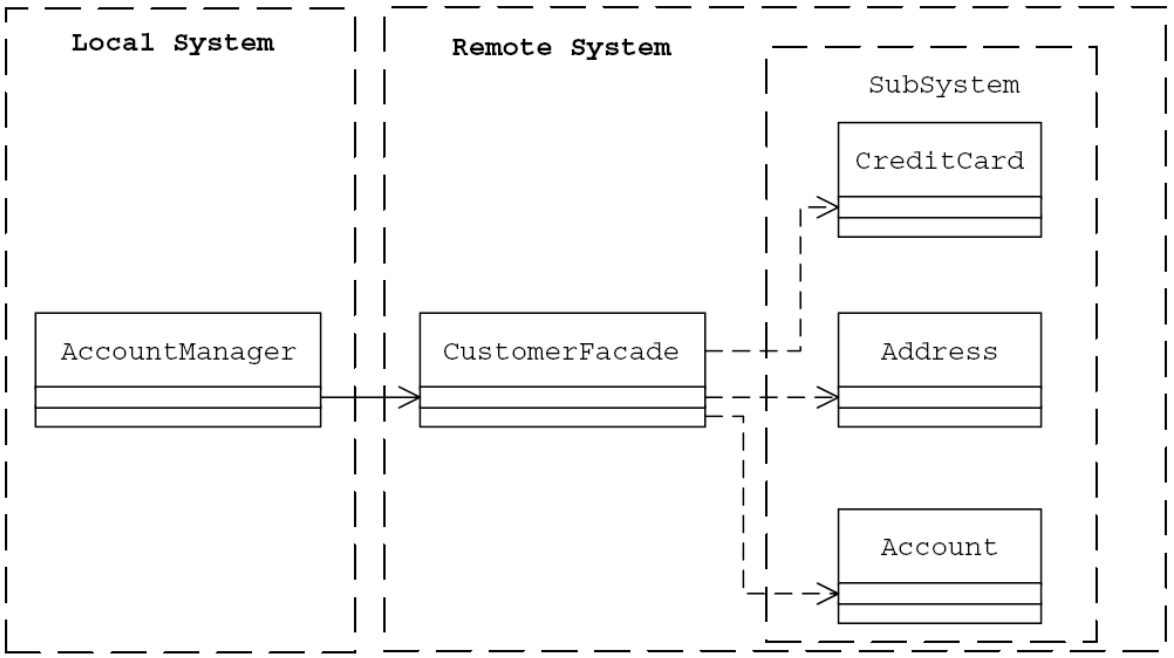
Let us build a different version of the same application that runs in the remote Mode.
In designing the application for the remote mode of operation, we would move all of the subsystem components (Account, Address and CreditCard) and the Façade (CustomerFacade) to a remote server

ADVANTATGES OF NEW DESIGN

- 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.

NEW DESIGN



REDESIGNING CustomerFaçade

-CustomerFacade Façade now implements the CustomerIntr remote interface.

- Different client objects can interface with the subsystem objects by invoking the CustomerIntr methods on the concrete CustomerFacade.

- 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.

```
public class CustomerFacade extends UnicastRemoteObject
  implements CustomerIntr {
                              ...
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…");
}
```

REDESIGNING AccountManager

- Similar to the local mode of operation, AccountManager displays the necessary user interface to accept the input customer data.

-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.

- 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 Customer-Facade class.

FINAL RESULT