Multiple Secure Observers using J2EE

Version 1.4 | 11-Aug-10

Author: Vivek Gondi

Abstract

The concept of notifying messages to multiple parties is a common scenario in several applications. The sender might need to send messages to a particular receiver but cannot reach him directly due to some constraints like lack of access, cumbersome to reach etc. We use this problem to model a pattern to send notifications/messages from source to destination in a secure and automatic way. The pattern uses multiple observers and security concepts to demonstrate this problem.

The paper discusses notifying messages between multiple objects using observer classes. It also discusses how messages can be transmitted between them by using encrypted messages at each step. In this way secure transmission of messages between them is achieved.

1 Introduction

1.1 Name of the Pattern

This pattern is called ‘Multiple Secure Observers’ as it uses a combination of observer classes and security mechanisms to relay messages from the starting point to the end point in a secure manner. Messages are sent to several subscribers using the framework of PKI and observer pattern. It is specific to J2EE framework as it uses observer classes provided by the Java framework but can be used in other languages using the similar idea. [2]

1.2 Classification

It can be categorized under behavioral pattern where the different objects communicate via “observers” located in the system. ‘Observers’ are chained in the system so a chain of observers are initiated for a particular action. It also uses the idea behind Façade pattern where the sender invokes a simple method to execute his service but is oblivious of the complexity behind the scenes. [4]

1.3 Context

In distributed systems, communication between objects is a key factor for a good system design. It is required that the starting point and end point needs to be connected in a secure and automatic fashion. Often at each step in the intermediate steps, action needs to be taken without any effort from the sender. For e.g.: look ups need to be performed and proper action needs to be taken in the intermediate steps. Based on the result of the lookup, another action needs to be initiated. And actions might need to be initiated in a chained manner. This kind of situation can be addressed using this pattern.

1.4 Problem

In several applications, it is hard for user to send messages to multiple subscribers in a secure fashion. Since many external subscribers are involved, the message can be intercepted by unauthorized persons. This is not a desired situation. This pattern addresses a secure way to transmit the required information using encryption using public/private key pairs.

Also many times, it is a cumbersome task to contact several subscribers for sending a uniform message. The problem being the user tries to contact several parties but cannot achieve it in a simple manner. This pattern allows the user to complete this task by invoking appropriate observer classes. Both of the above aspects are dealt with this pattern and the following sections explain how this pattern can help in solving these problems [Ref 1]

2 Solution

Overview: The overall workflow for the solution is explained here. Assume a typical use case, where the user would like to communicate to an end point for some requirement. In the beginning, the credentials of the user are verified by an 'Authentication Service'. The credentials could be valid SSN/Date of Birth/password etc or any identity that the user has registered. Once the authentication is successful, the flow passes on to the 'Broker' Class. In the solution, a 'Broker Class' mediates all requests from the user and routes them to the 'Service Provider' using lookup tables. Both the 'Broker' and 'Service Provider' implement 'Observer' and extend 'Observable' classes. The service provider in turn routes the request message to the required 'Service Implementer'. The end point in this pattern which executes the user's request is called 'Service Implementer'. Encryption and decryption of messages is handled by the 'Message Handler' class. All key related information is obtained from the 'KeyStore' class.

The details of the flow are discussed individually in each class below.

Login Class: The first step it performs is validating the credentials of the user using 'Authentication service' class. The credentials could be valid SSN, Date of Birth, Bank Details etc or any identity that the system is registered with. Upon successful login, it instantiates the 'MessageHandler' class and sets the required input message to the handler object. It is an 'Observable' class with add and notify methods. The addObserver() contains the Broker as the observer and notifyObserver() is responsible for publishing the message to the Broker automatically using Observable class

Finally, it contains the executeService() method. This method sends the encrypted message received from the MessageHandler class and calls the notifyObserver() method to notify the observers subscribed to this object.

MessageHandler Class: This is an utility class for handling all encrypted messages between different objects. The main responsibility of this class is to return an encrypted message (getEncryptedMsg()) to the calling class using the private key of the subscriber. When a subscriber gets an encrypted message, it is also responsbile for decrypting the message (getDecryptedMsg()) by obtaining the required public key from the 'KeyStore' class. So essentially, this class is responsible for the security aspects of messaging based on keys obtained from 'KeyStore' class.

ServiceLookup Class: This class provides the list of observers for a particular object. For e.g.: If Broker class is the input to the lookup() method, the method would return the list of 'Service Observers' for the particular 'Broker' class. It contains one overloaded method to accept different objects.

Broker Class: This class extends 'Observable' class and also implements the 'Observer' interface. The Broker is the 'Observer' for the Login class. Broker acts as the first 'Observer' for calls received by the client. Broker class also extends the ‘Observable’ class as any change to this object is ‘observed’ by the Service Provider class. The addObserver() contains the list of ‘Service Provider(s)’ that are “watching” the changes in Broker class. The relationship between the broker and service provider is provided by the Service Lookup class. Broker class gets the encrypted message from the message handler and decrypts the message by obtaining the public key from the ‘KeyStore’ class.

Service Provider Class: The service provider class receives requests from the ‘Broker Observer’. Just like Broker it is both ‘Observable’ and also extends the ‘Observer’ interface. This class observes’ changes to the Broker class and notifies the Service Implementer classes. This class is responsible for categorization of requests and passing the control to the appropriate ‘Service Implementer’ using look up. Just like the Client-Broker-Service Provider channel, the Broker-Service-Provider-Service Implementer communicates in a similar way. Thus multiple secure observer structure is set up.

Service Implementer Class: They are the ‘Observer(s)’ to the Service Providers and hence listen to all updates in the Service Provider class. The service implementers actually execute the service requested by the client and are the destination point for which the client is looking to. They are the end-point in this hierarchy to achieve the service what the client is looking for. It contains the execute() which accepts the service message, decrypts using the public key and performs the required action.

KeyStore Class: This contains public key for all observers to decode the encrypted messages. The key pair generation for each of the Observer classes is handled here. Every Observer object gets its public key from this class to read the encrypted messages when required. For e.g.: The Broker Observer class decrypts the message, deciphers what service the client is looking for using the key obtained from this class. It then forms the required content for message using the Message Handler class getEncryptedMessage(). The message is signed using Broker private key and passed to Service Provider Observer. Upon receiving the message by the ‘Service Provider’, it calls getDecryptedMessage() and decrypts the message as per key obtained from the ‘KeyStore’.

Since the messages pass via various third parties, asymmetric keys are used for generation. We want that the messages intended to service provider 1 should be not be read by service provider 2. Hence these asymmetric keys would help in keeping the privacy of the data.

The class diagram is illustrated below.

2.1 Class Diagram

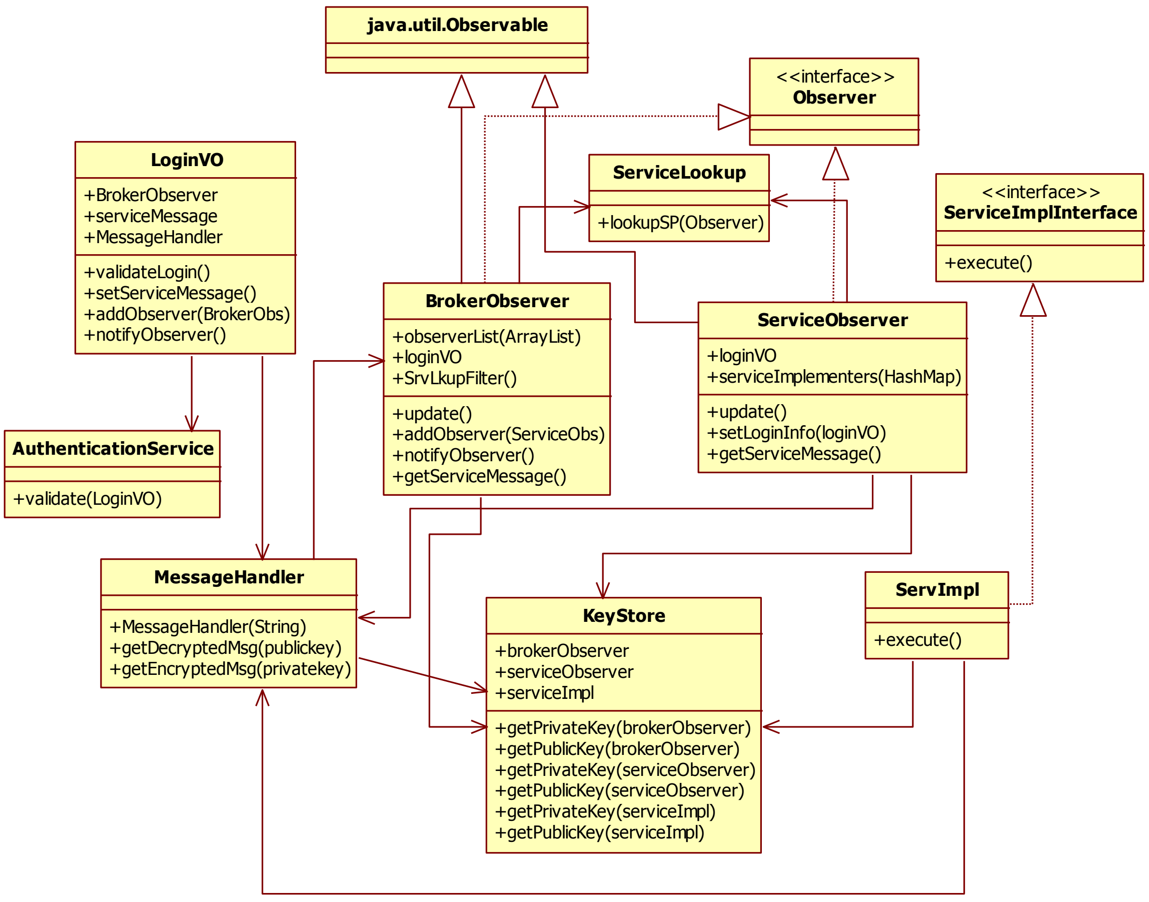

Fig 2.1.1: Class diagram for multiple secure observers

2.2 Collaboration

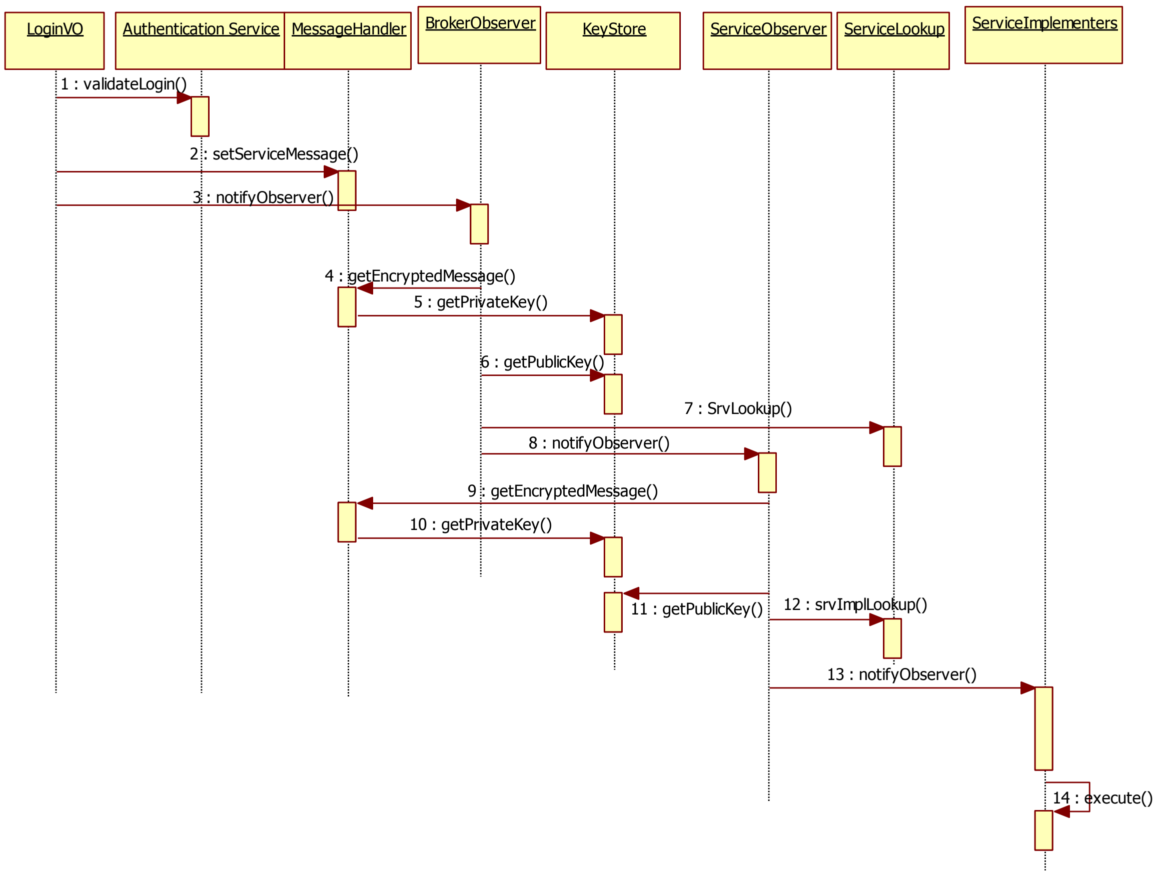

Fig 2.2.1: Sequence Diagram depicting the publishing of messages to subscribers

(We assume that the registering of the subscribers is done prior to this step)

Client contacts the broker for a particular action. Broker uses lookup service to find the service providers (e.g. Bank Entity, Insurance Entity etc). Each service provider has in turn service implementers. For e.g.: A bank entity would have Savings Account, Credit Card, Loan etc as service implementers. The service implementers execute the service requested by the user.

2.3 Sample Code

The detailed sample code is placed in the Appendix section (Section 2.5)
Only the test class is shown here for brevity.

Test Class:

```
package util.test;

public class TestEncryptedObserver  {

    public static void main (String args[]){
        try{
        LoginVO loginVO = new LoginVO();
        MessageHandler messageHandler = new MessageHandler();
       messageHandler.setMessage("Test message from client- update personal Info");
        loginVO.setMessageHandler(messageHandler);
        loginVO.executeService();
        }catch(Exception ex){
            //catch and log exceptions
        }
    }
}
```

<u>Output after running the Test class in Eclipse IDE:</u>

```
called the message Test message from client- update personal Info
called the validate method
called the broker update method
service observers added in loop
called the update method in ServiceObserver
called the service implementor method for client

Process finished with exit code 0
```

3 Consequences

Benefits:

- The pattern brings ease of transmission of information by the client to several other parties in a single call. It brings about uniform dissemination of information to all parties concerned via the broker interface in an encrypted manner and hence there is no ambiguity of information being sent to several parties.
- It aids in sending immediate warning message to several providers in a short span of time with less effort from the client.
- The secure transmission aids in privacy of the data being considered and is therefore very useful in many situations. The customer can be rest assured that his data is not being made public and misused by unwanted parties.
- It is also possible to add digital signatures to let the recipients verify the authenticity of messages.
- The pattern can be expressed also in other languages, e.g. .NET, with a few changes.
- It brings forth integration between client, broker and subscribers for different queries and issues. Since the information is securely transmitted, all communications between the parties can be stored, analyzed and can be used for non-repudiation purposes.

Downside:

- This pattern assumes proper communication channels between broker provider and the subscriber interfaces. But there might be disagreements to terms and conditions between all parties involved and hence messages might not reach concerned party in time.
- The customer information to some extent becomes public to broker system as messages can be decrypted by an intelligent hacker if the key is known and the message can be read.
- The formats of messages need not be standardized between different service providers. One provider might be using XML and another might be using a simple text message etc. Thus the integration between various layers is a challenge. Added to this, the complication of encryption and decryption at each layer might differ if different algorithms are used since several parties are involved.
- This pattern only uses a one way communication from the sender to the end point. The reverse communication is not handled online. The reverse communication could add more flexibility and usage to the pattern.

4 Known Uses

Some of the applications of this pattern are:

1) Many websites send automatic updates automatically to its subscribers when any subscribed services change (e.g.: Stock value changes in a range defined by the user). These messages preferably should not be intercepted by other users as it is meant for a particular user and hence this pattern is used for secure relay of messages.

2) Users of ‘FaceBook’ can subscribe to messages from ‘Twitter’ so that updates on ‘Twitter’ can be sent via SMS to the ‘FaceBook’ users. This uses the secure observer pattern where messages are transmitted securely.

3) Information received at a particular headquarters of an office is dispatched to multiple branches of the office via text messaging services. The sub-branches are informed about any updates, important communication regarding business updates automatically using the observer pattern. This kind of business related information is confidential and therefore encryption of messages provides required service. Some companies on mobile space like secureVoicegsm.com use this kind of idea.

5 Related Patterns

Item	Description
Front End Interceptor	Used in routing requests in the broker component
Publish/Subscribe	Used in the service provider layer to send messages to various service implementers
Façade	Calling service implementer methods from service observers
MVC	Used by the observer classes as they use the model-controller pattern

6 References

SNo	References
1.	A. Braga, C. Rubira, and R. Dahab, “Tropyc: A pattern language for cryptographic object-oriented software”, Chapter 16 in Pattern Languages of Program Design 4 (N. Harrison, B. Foote, and H. Rohnert, Eds.).
2.	F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal., Pattern-oriented software architecture, Wiley 1996
3.	www.javacamp.org
4.	Eric Gamma, Richard Helm, Ralph Johnson: Design Patterns: Elements of Re-Usable Object Oriented Software

7 Appendix

LoginVO Class:

```
package util.test;

import java.util.Observable;

//this is the only observable interface in the pattern
public class LoginVO extends Observable {

    BrokerObserver brokerObs = new BrokerObserver();
    //the message to be sent
    private String serviceMessage;
    MessageHandler messageHandler = new MessageHandler();
 //the brokerobservers are added here, in our pattern only one observer is added
blic void addObserver(BrokerObserver obj){
    this.addObserver(obj);
}
```

//This method validates the user credentials
//Further it assigns the message read by the MessageHandler to the observer(s) added to the system using notifyObserver() and executes the requested service

```
public boolean executeService(){
messageHandler.setServiceMessage(serviceMessage);
    System.out.println("called the message reader "+messageHandler.getMessageHandler());
//notify the Observer, in this case BrokerFilter
    System.out.println("called the validate method");
    notifyObserver();
    return true;
}

//calls the update of next observer i.e. the broker with the required message
public void notifyObserver(){
    brokerObs.update(messageHandler);
}
```

```
}
```

MessageHandler Class:

```
public MessageHandler();
   //some custom defined Reader to read messages
   public MessageHandler getMessageHandler() {
      return messageHandler;
   }

   public void setMessageHandler(MessageHandler someHandler) {
      this.messageHandler = someHandler;
   }

//returns encrypted signed message
   public String getServiceMessage() {
          // encrypt the data
          return message;
   }

   public void setServiceMessage(String serviceMessage) {
      this.serviceMessage = serviceMessage;
   }
```

BrokerObserver Class:

```
package util.test;

import java.util.*;

public class BrokerObserver extends Observable implements Observer {

//lookup for service providers based on the service description by the loginVO
private HashMap serviceProviders = new HashMap();
//holds list of observers for this subject(broker is now subject for serviceproviders)
private ArrayList observerList = new ArrayList();
 private MessageHandler messageHandler ;


//get list of service providers for the service from lookup
ServiceLookup svfilter = new ServiceLookup();
HashMap serviceProviderList =svfilter.lookupSP(this);
ServiceObserver servObs;

 //get the signed message from the handler
   public void update(Observable obj,Object ob){
      if(obj instanceof MessageHandler)
      this.setMessageHandler((MessageHandler) obj);
      System.out.println("called the broker update method");
      String brokerMessage = decryptBrokerMessage(messageHandler.getServiceMessage());
      addObserver(brokerMessage);
      notifyObserver();
```

```
    }


private void setMessageHandler(MessageHandler messageHandler)
{this.messageHandler=messageHandler;   }
    private MessageHandler getMessageHandler() {return messageHandler; }
}


//decrypt messages using the key obtained
    private  String decryptMessage(String encryptedMessage){
    //decrypt message and route to service provider
        return KeyStore.getKey(this);

    }

public void addObserver(String message){
        //adding a test service observer object  from message received
        serviceProviderList.put(new ServiceObserver(),new ServiceObserver());
        Iterator it =   serviceProviderList.values().iterator();
        while(it.hasNext())
            observerList.add(it.next());
    }
    public void notifyObserver(){
        Iterator it=serviceProviderList.values().iterator();

        while( it.hasNext() ) {
                servObs = ( ServiceObserver )it.next();
                System.out.println("service observers added in loop");
                servObs.update(this, messageHandler );
        }
    }
}
```

ServiceLookup Class:

```
package util.test;
import java.util.HashMap;

public class ServiceLookup {
      public HashMap lookupSP(BrokerObserver brokObs){
        HashMap srvProvider = new HashMap();
        //look up code for service providers
        return srvProvider;
    }


        public HashMap lookupSP(ServiceObserver servObs){
        HashMap srvImpl = new HashMap();
        //look up code for service implementers
        return srvImpl;
        }
}
```

ServiceObserver Class:

```
package util.test;

import java.util.Observer;
import java.util.Observable;
import java.util.HashMap;
import java.util.Iterator;


public  class ServiceObserver extends Observable implements Observer {
   MessageHandler messageHandler ;
   HashMap servImpl ;

private HashMap lookUPServImpl(){
      String message = decryptServMessage();
      //based on message lookup service Impl
      servImpl = new ServiceLookup().lookupSP(this);
      return servImpl;
   }


    private String decryptServMessage(String encryptedMessage){
      //decrypt message and route to service provider
      return KeyStore.getKey(this);
    }

//add test service implementers

   public void update(Observable obs,Object obj){
      lookUPServImpl();
      servImpl.put(new ServImpl_1(),new ServImpl_1());
      //get list of serviceimplementers
      Iterator it = servImpl.keySet().iterator();

   if(obs instanceof BrokerObserver){
      //get broker and client VO
      this.setClientInfo(((BrokerObserver)obs).getClientVO());
      //execute relevant service methods
      while(it.hasNext()){
      ServiceImplInterface sp =(ServiceImplInterface)it.next();
         System.out.println("called the update method in ServiceObserver");
         sp.execute(messageHandler.getMessage());
      }
   }
   }

private void setMessageHandler(MessageHandler messageHandler)
{this.messageHandler=messageHandler;   }
   private MessageHandler getMessageHandler() {return messageHandler; }
}
```

ServImplInterface Interface:

```
package util.test;

public interface ServiceImplInterface {
    public void execute(MessageHandler messageHandler) ;
}
```

ServImpl Class:

```
package util.test;

public class ServImpl implements ServiceImplInterface  {
    public void execute(MessageHandler msgHandler){
    System.out.println("called the service implementer method for client");
    //implementation of actual service here
}
}
```

KeyStore Utility Class:

```
package util.test;

public class KeyStore   {
    private static String servObsKey;
    private static String brokerObsKey;
    private static String servImplKey;


    public static String getKey(ServiceObserver sobs){
        //some implementation
        return servObsKey;
    }
     public static String getKey(BrokerObserver bobs){
        //some implementation
        return brokerObsKey;
    }
}
```