95-702 Distributed Systems
Project 3

Assigned: Friday, February 19, 2016

Due: Friday, March 4, 11:59 PM

Web Service Design Styles

Web services are used to provide interoperable client and server interaction. The implementation language of the client and service may differ. The client may be a stand-alone program or a program running within a browser. Web services are an important foundation for the construction of service-oriented architectures. Web services may be deployed within the same enterprise as the client or web services may be deployed on the cloud.

Software architecture is about the decisions that are made that influence many of the non-functional characteristics of a software system.

In this project, we will build three small systems that illustrate various ways to design web services. These different design approaches will have the same functional characteristics but different non-functional quality attributes.

Before working on this project, view the videos on Service Design Styles on the course schedule.

Your first task will be to build, deploy, and test a simple JAX-WS web service and web service client. Your solution will employ SOAP documents holding the procedure name and a list of typed parameters. As was discussed in class, this is a tightly coupled approach. It has the benefit of being very simple to build and deploy.

Your second task will be to build a JAX-WS web service using a single message argument. This is a less tightly coupled approach – allowing clients and services to more easily change – but it is more of a challenge to build. You will have to take care of the XML messages that are placed within the SOAP documents.

For the first two tasks, be sure to see the short video on the schedule showing the construction of a JAX-WS service and JAX-WS service client.

Your third task will be to build a REST style web service using a simple JEE servlet and a client. The servlet will provide doGet, doPost, doDelete and doPut methods. To communicate with the servlet, the client will use the java.net.HttpURLConnection and java.net.URL classes.

In each of these assignments we will be making use of the two classes provided below. One class (Spy.java) holds information on a single spy. The class (SpyList.java) is a singleton (may only be created once) and maintains a list of spies. Each spy has a unique user id. Be sure to modify these classes when appropriate. For example, it would do well to add a toString method to Spy.java.

Some terminology

We will use the following terminology in the remainder of this document. By simple string we will mean a string with no markup. For example, the string "DS is the coolest!" is a simple string. The string "<a>Hello James Bond</a>" is not simple. An XML string is a string containing XML. "<a>Hello James Bond</a>" is an XML string.

In the discussion that follows, the name of the spy should be treated as a unique key. That is, a spy 'name' is really the spy's user ID.

Task 1 A Tightly Coupled JAX-WS Web Service With Typed Parameters

In this task, you will create two Netbeans projects named WebServiceDesignStyles1ProjectServer and WebServiceDesignStyles1ProjectClient.

The first project will contain the service. The second project will contain the client and will be a Java application.

Along the way, be sure to take screen shots showing the testing platform. This platform runs in the browser and shows the request and response SOAP documents.

1) Write a JAX-WS web services (using SOAP) that will allow a client to add new spy, update an existing spy, delete a spy, show a specific spy and list all of the spies in a spy list. We will allow the client to display the spy list in two different formats – simple string and XML.

   The web service methods have the following signatures and pre- and post-conditions:

   Web Service Operation: addSpy

   Pre: A name, title, location, and password are provided as input.

   Post: A new spy is added to the list of spies. The value returned

   is a simple String representation of the spy.

   If the spy already exists in the spy list, no change is made to the spy list

   and the string that is returned informs the user that no update was

   made.

```
@WebMethod(operationName = "addSpy")
public String addSpy(@WebParam(name = "name") String name,
                                   @WebParam(name = "title") String title,
                                   @WebParam(name = "location") String location,
                                   @WebParam(name = "password") String password);
```

Web Service Operation: updateSpy

Pre: A name, title, location, and password are provided as input.

Post: An existing spy is updated. The value returned

is a simple String representation of the updated spy.

If the spy does not already exists in the spy list, no change is made to the spy list and the string that is returned informs the user that no update was made.

```
    @WebMethod(operationName = "updateSpy")
    public String updateSpy(@WebParam(name = "name") String name,
                            @WebParam(name = "title") String title,
                            @WebParam(name = "location") String location,
                            @WebParam(name = "password") String password);
```

Web Service Operation: getSpy

Pre: A name is provided as input.

Post: An existing spy is returned as a string. If no such spy exists

then the message “No such spy” is returned.

```
    @WebMethod(operationName = "getSpy")
    public String getSpy(@WebParam(name = "name") String name) {
```

Operation: deleteSpy

Pre: A spy name is provided as input.

Post: The spy is deleted from the list of spies. The value returned is a simple String that says “Spy” <name> “was deleted from the list.” If no spy with that name exists in the list then deleteSpy() returns a string message stating that fact.

```
@WebMethod(operationName = "deleteSpy")
public String deleteSpy(@WebParam(name = "name") String name)
```

Operation: getList

Pre: None

Post: A simple string is returned that contains all of the spy data on the spy list. If the list is empty then an empty string is returned.

```
@WebMethod(operationName = "getList")
public String getList()
```

Operation: getListAsXML

Pre: None

Post: An XML simple string is returned that contains all of the spy data on the list. If the list is empty then an empty XML string is returned. The XML string has a start and an end tag but no spies.

```
@WebMethod(operationName = "getListAsXML")
public String getListAsXML()
```

2) Write a JAX-WS web service client (using SOAP) with a main routine that tests the web service. The main routine must test each operation of the service. There does not have to be any user interaction. The main routine of your web service client will perform the following actions. Be sure to also include your favorite actor's name as a spy. Insert his or her name and check to make sure it is displayed after a call to getList() and getListAsXML(). Include that insertion and display it in your main routine. In addition, be sure to use the testing page to test edge cases.

```
System.out.println(getList());
System.out.println(getListAsXML());
addSpy("mikem","spy","Pittsburgh","sesame");
addSpy("joem","spy","North Hills","xyz");
addSpy("seanb","spy commander","South Hills","abcdefg");
addSpy("jamesb","spy","Adelaide","sydney");
addSpy("adekunle","spy","Pittsburgh","secret");
System.out.println(getList());
System.out.println(getListAsXML());
updateSpy("mikem", "super spy", "Pittsburgh","sesame");
System.out.println(getListAsXML());
String result = getSpy("jamesb");
System.out.println(result);
deleteSpy("jamesb");
result = getSpy("jamesb");
System.out.println(result);
```

Task 2 A Less Tightly Coupled JAX-WS Web Service Using a Single Message Argument

With a single message argument design, we are more focused on the message itself – rather than the operations. In practice, the message design might come from a standards body or an industry consortium. In this small exercise, we will focus on a message that carries information about spies.

In this task, you will create two Netbeans projects named WebServiceDesignStyles2ProjectServer and WebServiceDesignStyles2ProjectClient.

Be sure to take screen shots showing the testing platform running in the browser – showing the SOAP documents.

1) In general, this is the same assignment as in Task 1 and the overall functionality will be the same. For each operation, the semantics are the same as in Task 1. But, rather than using several different operations, there will be only one web service operation. We will name this operation spyOperation and it will take an argument of type String. There will always be an XML document within the string. The XML document describes what operation needs to be performed as well as information about a spy (when appropriate). For example, here is an XML document that will be used to add a spy (spaces and indentation are not present in the real message):

   ```
   <spyMessage>
       <operation>addSpy</operation>
       <spy>
           <name>joem</name>
           <spyTitle>spy</spyTitle>
           <location>Pittsburgh</location>
           <password>joe</password>
       </spy>
   </spyMessage>
   ```

   Since we are using JAX-WS, this message will automatically be wrapped in a SOAP document.

Here is an XML document that will be used to update an existing spy:

```
<spyMessage>
        <operation>updateSpy</operation>
        <spy>
                <name>joem</name>
                <spyTitle>spy</spyTitle>
                <location>Pittsburgh</location>
                <password>joe</password>
        </spy>
</spyMessage>
```

Here is an XML document request message that will be used to get a spy (in XML):

```
<spyMessage>
        <operation>getSpyAsXML</operation>
        <spy>
                <name>mikem</name>
                <spyTitle></spyTitle>
                <location></location>
                <password></password>
        </spy>
</spyMessage>
```

Here is an XML document request message that will be used to delete a spy:

```
<spyMessage>
    <operation>deleteSpy</operation>
    <spy>
        <name>mikem</name>
        <spyTitle></spyTitle>
        <location></location>
        <password></password>
    </spy>
</spyMessage>
```

Here is an XML document request message that will be used to retrieve the spy list as a simple string:

```
<spyMessage>
    <operation>getList</operation>
    <spy>
        <name></name>
        <spyTitle></spyTitle>
        <location></location>
        <password></password>
    </spy>
</spyMessage>
```

Here is an XML document that will be used to list the spy list in XML:

```
<spyMessage>
    <operation>getListAsXML</operation>
    <spy>
            <name></name>
            <spyTitle></spyTitle>
            <location></location>
            <password></password>
    </spy>
</spyMessage>
```

In the above example, we are making two simplifications. First, we are passing the <spy>...</spy> data in each message even though, in some cases, we are not using these data. If you would like, you may remove this simplification and improve he system. A getListAsXML operation, for example, would look like this:

```
<spyMessage>
    <operation>getListAsXML</operation>
</spyMessage>
```

Second, we are not bothering to build a standard response message. Instead, in some cases, we are returning simple strings. If you would like, you may remove this simplification and design an XML response message for this service. Each response would have a particular XML structure – similar to the way we have handled the request messages. The XML format will be of your own design.

A five point bonus is available for removing these two simplifications.

2) Write a JAX-WS web service client (using SOAP) with a main routine that tests the web service. The main routine must test each operation of the service. There does not have to be any user interaction. The main routine of your web service client will perform the following actions. Be sure to also include your favorite actor's name as a spy. Insert his or her name and check to make sure it is displayed after a request to getList and getListAsXML. Include that insertion and display in your main routine.

```
// Note: There is NO communication code or XML handling code in the main
// routine. That important work is done in spyOperation. We are separating
// concerns with a proxy design.
String result = "";
System.out.println("Adding spy jamesb");
// create a spy
Spy bond = new Spy("jamesb", "spy", "London","james");
// create a message
SpyMessage sb = new SpyMessage(bond,"addSpy");
// make a call on the web service
result = spyOperation(sb.toXML());
System.out.println(result);

System.out.println("Adding spy seanb");
Spy beggs = new Spy("seanb", "spy master", "Pittsburgh","sean");
SpyMessage ss = new SpyMessage(beggs,"addSpy");
result = spyOperation(ss.toXML());
System.out.println(result);

System.out.println("Adding spy joem");
Spy mertz = new Spy("joem", "spy", "Los Angeles","joe");
SpyMessage sj = new SpyMessage(mertz,"addSpy");
result = spyOperation(sj.toXML());
System.out.println(result);

System.out.println("Adding spy mikem");
Spy mccarthy = new Spy("mikem", "spy", "Ocean City Maryland","sesame");
SpyMessage sm = new SpyMessage(mccarthy,"addSpy");
result = spyOperation(sm.toXML());
```

```
System.out.println(result);

System.out.println("Displaying spy list");
SpyMessage list = new SpyMessage(new Spy(),"getList");
result = spyOperation(list.toXML());
System.out.println(result);

System.out.println("Displaying spy list as XML");
SpyMessage listXML = new SpyMessage(new Spy(),"getListAsXML");
result = spyOperation(listXML.toXML());
System.out.println(result);

System.out.println("Updating spy jamesb");
Spy newJames = new Spy("jamesb","Cool Spy","New Jersey","sesame");
SpyMessage um = new SpyMessage(newJames,"updateSpy");
result = spyOperation(um.toXML());
System.out.println(result);

System.out.println("Displaying spy list");
list = new SpyMessage(new Spy(),"getList");
result = spyOperation(list.toXML());
System.out.println(result);

System.out.println("Deleting spy jamesb");
Spy james = new Spy("jamesb");
SpyMessage dm = new SpyMessage(james,"deleteSpy");
result = spyOperation(dm.toXML());
System.out.println(result);
System.out.println("Displaying spy list");
list = new SpyMessage(new Spy(),"getList");
result = spyOperation(list.toXML());
System.out.println(result);

System.out.println("Displaying spy list as XML");
listXML = new SpyMessage(new Spy(),"getListAsXML");
result = spyOperation(listXML.toXML());
System.out.println(result);



System.out.println("Deleting spy Amos");
Spy amos = new Spy("amos");
SpyMessage am = new SpyMessage(amos,"deleteSpy");
result = spyOperation(am.toXML());
System.out.println(result);
```

In my solution to Task 2, I wrote a class called SpyMessage. I used the SpyMessage class (as well as the Spy class) on the server and on the client. Note: the SpyList class only resides on the server.

On the server side, SpyMessage objects are always constructed with an XML string (coming in over a network from the client). Then, the server extracts the operation and the spy information from the SpyMessage object. The web service has one operation – spyOperation. The operation acts as a dispatcher – reading what is requested and then calling the appropriate method in the SpyList class. The SpyMessage class handles all of the parsing from the XML string. It constructs a DOM tree and reads the operation (and perhaps spy information) from the tree.

On the client side, the SpyMessage objects are constructed as you see in the test client above – with a spy object and an operation. The SpyMessage is used to easily generate the XML that needs to be transferred to the server.

The SpyMessage class is used to create XML (that’s easy to do) and to read and parse XML (a bit more challenging.) In order to parse the XML, I used the following code within the SpyMessage class:

```
private Document getDocument(String xmlString) {
  DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
  DocumentBuilder builder;
  Document spyDoc = null;
  try  {
     builder = factory.newDocumentBuilder();
     spyDoc = builder.parse( new InputSource( new StringReader( xmlString ) ) );

  } catch (Exception e) {
      e.printStackTrace();
 }
 return spyDoc;
}
```

Once the xmlString is converted to a Document object, we can read data from the Document object with code like the following:

```
spyDoc.getDocumentElement().normalize();

System.out.println("Root element :" +
spyDoc.getDocumentElement().getNodeName());
// the root element should be “spyMessage”
NodeList nl = spyDoc.getElementsByTagName("name");
Node n = nl.item(0);
String name = n.getTextContent();
temp.setName(name);
```

You will make use of this same idea when reading XML data (e.g. in doPut) arriving from the client in Task 3. In that case, you are are not reading a SpyMessage, but just a spy.

Task 3 This is an approach using the principles of Representational State Transfer (REST)

In this task, you will create two Netbeans projects named WebServiceDesignStyles3ProjectServer and WebServiceDesignStyles3ProjectClient. In this task, it is important that WebServiceDesignStyles3ProjectServer be a standard Java Web project and that WebServiceDesignStyles3ProjectClient be a standard Java SE project. That is, we will not be using JAX-WS or JAX-RS or any other frameworks. Simply put, we will be working with a standard JEE servlet and a standard Java SE client.

In this task, you will implement a simple REST protocol. The service will allow for CRUD (Create, Read, Update and Delete) operations on the SplyListCollection. The contract the service will expose is described in the following table:

HTTP Verb/Body	URI	Use
POST The body of the request holds a representation of an existing spy in XML.	/SpyListCollection	Update an existing spy with the representation in the body of this request
GET Use Accept text/xml or Use Accept test/plain	/SpyListCollection/{name}	Request an XML or plain text representation of the spy named {name}
GET Use Accept text/xml or Use Accept test/plain	/SpyListCollection	Request an XML or plain text representation of the SpyListCollection
PUT The body of the request holds a representation of a new spy in XML.	/SpyListCollection/{name}	Create a new spy and add the spy to the SpyListCollection
DELETE	/SpyListCollection/{name}	Delete the spy with name {name}

Implement the service interface with a standard Java servlet called SpyListCollection. Note: delete the processRequest method and the getServletInfo method. We are not using either one.

In order to allow the client to add a name to a URL, change the line

@WebServlet(name = "SpyListCollection", urlPatterns = {"/SpyListCollection"})

to include a wild card in the URL pattern. This line should now read:

@WebServlet(name = "SpyListCollection", urlPatterns = {"/SpyListCollection/*"}).

The HTTP status codes (return messages) of the service are shown here:

HTTP Verb	URI	Status Code
POST	/SpyListCollection	200 OK or 404 Not found 404 if {name} is not in the SpyListCollection. The value of {name} is found within the XML message and is not passed in the URL.
GET	/SpyListCollection/{name}	200 OK or 404 Not found 404 if {name} is not in the SpyListCollection
GET	/SpyListCollection	200 OK (The SpyListCollection is always available, even when empty)
PUT	/SpyListCollection/{name}	201 Created or 405 Method not allowed. If a spy with the provided name already exists in the collection then return 405.
DELETE	/SpyListCollection/{name}	200 OK or 404 Not found 404 if {name} is not in the SpyListCollection

The HTTP status codes may be set on the server side by using the setStatus() method of the response object. Your servlet will have doGet(), doPost(), doDelete() and doPut() methods defined. You will only have one doGet even though there are two URL's associated with the GET request (with two options each.) To learn what representation the client requires, use the getHeader("Accept") method of the request object.

The client side code (standard Java SE) will perform communication using the URL class and the HttpURLConnection class. For example, in order to make a call to the get method, your code might look like this:

```
URL url = new
URL(http://localhost:8080/WebServiceDesignStyles3ProjectServer/" +
"SpyListCollection");
HttpURLConnection con = (HttpURLConnection) url.openConnection();
con.setRequestMethod("GET");
con.setRequestProperty("Accept", "text/xml");
```

The client side code can use the con.getResponseCode() method to examine the HTTP response code. It can also use the con.getInputStream() method to read the server's response body.

The client side code will use a proxy design. In other words, all of the communications related code will be isolated within proxies. Your client should provide a demonstration of each operation.

Hint: You may use StackOverflow for examples on how to use the HttpURLConnection. If you use any code from StackOverflow, be sure to cite it in your comments.

Here is the main routine of my client. Your main routine should be similar. The output shown in comments is correct for the first run. On subsequent runs, the output differs since the state on the server changes.

```
System.out.println("Begin main");
Spy spy1 = new Spy("mikem","spy", "Pittsburgh","sesame");
Spy spy2 = new Spy("joem","spy", "Philadelphia","obama");
Spy spy3 = new Spy("seanb","spy commander", "Adelaide","pirates");
Spy spy4 = new Spy("jamesb","007", "Boston","queen");

System.out.println(doPut(spy1));     // 201
System.out.println(doPut(spy2));     // 201
System.out.println(doPut(spy3));     // 201
System.out.println(doPut(spy4));     // 201

System.out.println(doDelete("joem")); // 200
spy1.setPassword("Doris");
System.out.println(doPost(spy1));    // 200

System.out.println(doGetListAsXML()); // display xml
System.out.println(doGetListAsText()); // display text

System.out.println(doGetSpyAsXML("mikem"));  // display xml
System.out.println(doGetSpyAsText("joem"));  // 404

System.out.println(doGetSpyAsXML("mikem")); // display xml
System.out.println(doPut(spy2));   // 201
System.out.println(doGetSpyAsText("joem"));  // display text
System.out.println("End main");
```

Project 3 Summary

Be sure to review the grading rubric on the schedule. We will use that rubric in evaluating this project. Documentation is always required. The documentation may be written using the JavaDoc style - but that is not required. Normal Java commenting syntax is fine.

There will be 6 projects in Netbeans:

- WebServiceDesignStyles1ProjectServer
- WebServiceDesignStyles1ProjectClient
- WebServiceDesignStyles2ProjectServer
- WebServiceDesignStyles2ProjectClient
- WebServiceDesignStyles3ProjectServer
- WebServiceDesignStyles3ProjectClient

You should also have three screen shot folders:

- Project3Task1ScreenShots
- Project3Task2ScreenShots
- Project3Task3ScreenShots

For each Netbeans project, File->Export Project->To Zip...each. You must export in this way and NOT just zip the Netbeans project folders. In addition, zip all the Netbeans export zips and the screenshot folders into a folder named with your andrew id.

Zip that folder and submit it to Blackboard.

The submission should be a single zip file.

Project 3 Code Spy.java

Spy.java

```
package edu.cmu.andrew.mm6;
class Spy {
    // instance data for spies
    private String name;
    private String title;
    private String location;
    private String password;




    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public String getLocation() {
        return location;
    }

    public String getName() {
        return name;
    }

    public String getTitle() {
        return title;
    }
```

```
public void setLocation(String location) {
    this.location = location;
}

public void setName(String name) {
    this.name = name;
}

public void setTitle(String title) {
    this.title = title;
}

public Spy(String name, String title, String location, String password) {
    this.name = name;
    this.title = title;
    this.location = location;
    this.password = password;
}

public Spy(String name, String title, String location) {
    this.name = name;
    this.title = title;
    this.location = location;
    this.password = "";
}
public Spy(String name) {
    this.name = name;
    this.title = "";
    this.location = "";
    this.password = "";
}
```

```
    public Spy() {
        this.name = "";
        this.title = "";
        this.location = "";
        this.password = "";

    }

    public String toXML() {
        StringBuffer xml = new StringBuffer();

        xml.append("<spy>");
        xml.append("<name>" + name + "</name>");
        xml.append("<spyTitle>" + title + "</spyTitle>");
        xml.append("<location>" + location + "</location>");
        xml.append("<password>" + password + "</password>");
        xml.append("</spy>");
        return xml.toString();

    }

    public static void main(String args[]) {
        Spy s = new Spy("james","spy", "Pittsburgh", "james");
        System.out.println(s);
    }
}
```

Project 3 Code SpyList.java

```
package edu.cmu.andrew.mm6;
import java.util.Collection;
import java.util.Iterator;
import java.util.Map;
import java.util.TreeMap;

public class SpyList {

  private Map tree = new TreeMap();

    private static SpyList spyList = new SpyList();

    private SpyList() {
    }

    public static SpyList getInstance() {
        return spyList;
    }
    public void add(Spy s) {
        tree.put(s.getName(), s);
    }
    public Object delete(Spy s) {
        return tree.remove(s.getName());
    }
    public Spy get(String userID) {
        return (Spy) tree.get(userID);
    }

    public Collection getList() {
        return tree.values();
    }

    public String toString() {

        StringBuffer representation = new StringBuffer();
        Collection c = getList();
        Iterator sl = c.iterator();
        while(sl.hasNext()) {
            Spy spy = (Spy)sl.next();
            representation.append("Name: " + spy.getName()+" Title: " + spy.getTitle()+
                                    " Location: " + spy.getLocation());
        }
        return representation.toString();
    }

    public String toXML() {
        StringBuffer xml = new StringBuffer();
        xml.append("<spylist>\n");

        Collection c = getList();
        Iterator sl = c.iterator();
        while(sl.hasNext()) {
            Spy spy = (Spy)sl.next();
            xml.append(spy.toXML());
        }
        // Now, close
        xml.append("</spylist>");

        System.out.println("Spy list: " + xml.toString());
        return xml.toString();
    }
}
```