Advertisement: Support JavaWorld, click here!

Join the Java Community Process sm program
to vote on Executive Committee members and influence Java t
Join the JCP sm program today.

Design a simple service-oriented J2EE application framework

Leverage Struts, Spring, Hibernate, and Axis

Summary
Often, a J2EE Web application framework—Struts, for example—doesn't address the Web-tier object references between `Action`/servlet and other layers, such as a plain old Java object (POJO) business manager, Enterprise JavaBeans (EJB), Web services, and a data access object (DAO), or between a DAO and JDBC (Java Database Connectivity) stored procedures. Thus, Java developers end up with messy code in the Web tier `Action`/servlet. This article describes in detail the steps for developing a custom framework that addresses those issues. (*3,000 words;* **October 4, 2004**)

By Fangjian Wu

Today, developers are inundated with open source frameworks that help with J2EE programming: Struts, Spring, Hibernate, Tiles, Avalon, WebWorks, Tapestry, or Oracle ADF, to name a few. Many developers find that these frameworks are not the panacea to their problems. Just because they are open source doesn't mean they are easy to change and improve. When a framework falls short in a key area, addresses only a specific domain, or is just bloated and too expensive, you might need to build your own framework on top of it. Building a framework like Struts is a nontrivial task. But incrementally developing a framework that leverages Struts and other frameworks doesn't have to be.

In this article, I show you how to develop **X18p** (Xiangnong 18 Palm, named for a legendary powerful kung fu fighter), a sample framework that addresses two common issues ignored by most J2EE frameworks: tight coupling and bloated DAO (data access object) code. As you'll see later, X18p leverages Struts, Spring, Axis, Hibernate, and other frameworks at various layers. Hopefully, with similar steps, you can roll your own framework with ease and grow it from project to project.

The approach I take in developing this framework uses concepts from IBM's Rational Unified Process (RUP). I follow these steps:

1. Set simple goals initially
2. Analyze the existing J2EE application architecture and identify the issues
3. Compare alternative frameworks and select the one that is simplest to build with
4. Develop code incrementally and refactor often
5. Meet with framework's end-user and collect feedback regularly
6. Test, test, test

Step 1. Set simple goals

It is tempting to set ambitious goals and implement a cutting-edge framework that solves all problems. If you have sufficient resources, that is not a bad idea. Generally, developing a framework upfront for your project is considered overhead that fails to provide tangible business value. Starting smaller helps you lower the unforeseen risks, enjoy less development time, lower the learning curve, and get project stakeholders' buy-in. For X18p, I set only two goals based on my past encounters with J2EE code:

1. Reduce J2EE `Action` code coupling
2. Reduce code repetition at J2EE DAO layer

Overall, I want to provide better quality code and reduce the total cost of development and maintenance by increasing my productivity. With that, we go through two iterations of Steps 2 through 6 to meet those goals.

Reduce code coupling

Step 2. Analyze previous J2EE application architecture

If a J2EE application framework is in place, we first must see how it can be improved. Obviously, starting from scratch doesn't make sense. For X18p, let's look at a typical J2EE Struts application example, shown in Figure 1.

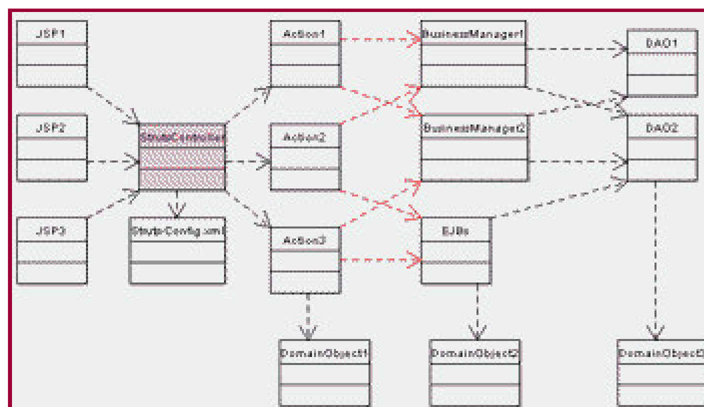


Figure 1. J2EE Struts application architecture. Click on thumbnail to view full-sized image.

`Action` calls `XXXManager`, and `XXXManager` calls `XXXDAO`s. In a typical J2EE design that incorporates Struts, we have the following items:

- `HttpServlet` or a Struts `Action` layer that handles `HttpRequest` and `HttpResponse`
- Business logic layer
- Data access layer
- Domain layer that maps to the domain entities

What's wrong with the above architecture? The answer: tight coupling. The architecture works just fine if the logic in `Action` is simple. But what if you need to access many EJB (Enterprise JavaBeans) components? What if you need to access Web services from various sources? What if you need to access JMX (Java Management Extensions)? Does Struts have a tool that helps you look up those resources from the `struts-config.xml` file? The answer is no. Struts is meant to be a Web-tier-only framework. It is possible to code `Action`s as various clients and call the back end via the Service Locator pattern. However, doing so will mix two different types of code in `Action`'s `execute()` method.

The first type of code relates to the Web-tier `HttpRequest`/`HttpResponse`. For instance, code retrieves

HTTP form data from `ActionForm` or `HttpRequest`. You also have code that sets data in an HTTP request or HTTP session and forwards it to a JSP (JavaServer Pages) page to display.

The second code type, however, relates to the business tier. In `Action`, you also invoke backend code such as `EJBObject`, a JMS (Java Message Service) topic, or even JDBC (Java Database Connectivity) datasources and retrieve the result data from the JDBC datasources. You may use the Service Locator pattern in `Action` to help you do the lookup. It's also possible for `Action` to reference only a local POJO (plain old Java object) `xxxManager`. Nevertheless, a backend object or `xxxManager`'s method-level signatures are exposed to `Action`.

That's how `Action` works, right? The nature of `Action` is a servlet that is supposed to care about how to take data in from HTML and set data out to HTML with an HTTP request/session. It also interfaces to the business-logic layer to get or update data from that layer, but in what form or protocol, `Action` could care less.

As you can imagine, when a Struts application grows, you could end up with tight references between `Action`s (Web tier) and business managers (business tier) (see the red lines and arrows in Figure 1).

To solve this problem, we can consider the open frameworks in the market—let them inspire our own thinking before we make an impact. Spring Framework comes on my radar screen.

Step 3. Compare alternative frameworks

The core of Spring Framework is a concept called `BeanFactory`, which is a good lookup factory implementation. It differs from the Service Locator pattern in that it has an Inversion-of-Control (IoC) feature previously called *Injection Dependency*. The idea is to get an object by calling your `ApplicationContext`'s `getBean()` method. This method looks up the Spring configuration file for object definitions, creates the object, and returns a `java.lang.Object` object. `getBean()` is good for object lookups. It appears that only one object reference, `ApplicationContext`, must be referenced in the `Action`. However, that is not the case if we use it directly in the `Action`, because we must cast `getBean()`'s return object type back to the EJB/JMX/JMS/Web service client. `Action` still must be aware of the backend object at the method level. Tight coupling still exists.

If we want to avoid an object-method-level reference, what else we can use? Naturally, **service**, comes to mind. Service is a ubiquitous but neutral concept. Anything can be a service, not necessarily just the so-called Web services. `Action` can treat a stateless session bean's method as a service as well. It can treat calling a JMS topic as consuming a service too. The way we design to consume a service can be very generic.

With strategy formulated, danger spotted, and risk mitigated from the above analysis and comparison, we can spur our creativity and add a thin service broker layer to demonstrate the service-oriented concept.

Step 4. Develop and refactor

To implement the service-oriented concept thinking into code, we must consider the following:

- The service broker layer will be added between the Web tier and the business tier.
- Conceptually, an `Action` calls a business service request only, which passes the request to a service router. The service router knows how to hook up business service requests to different service provider controllers or adapters by looking up a service mapping XML file, `X18p-config.xml`.
- The service provider controller has specific knowledge of finding and invoking the underlying business services. Here, business services could be anything from POJO, LDAP (lightweight directory access protocol), EJB, JMX, COM, and Web services to COTS (commercial off the shelf) product APIs. `X18p-config.xml` should supply sufficient data to help the service provider controller get the job done.

- Leverage Spring for X18p's internal object lookup and references.
- Build service provider controllers incrementally. As you will see, the more service provider controllers implemented, the more integration power X18p has.
- Protect existing knowledge such as Struts, but keep eyes open for new things coming up.

Now, we compare the `Action` code before and after applying the service-oriented X18p framework:

Struts Action without X18p

```
   public ActionForward execute(ActionMapping mapping, ActionForm form, HttpServletRequest
request, HttpServletResponse response)throws IOException, ServletException {
      ...

      UserManager userManager = new UserManager();

      String userIDRetured = userManager.addUser("John Smith")
      ...
}
```

Struts Action with X18p

```
public ActionForward execute(ActionMapping mapping, ActionForm form, HttpServletRequest
request, HttpServletResponse response)
      throws IOException, ServletException {
      ...
      ServiceRequest bsr = this.getApplicationContext().getBean("businessServiceRequest");

      bsr.setServiceName("User Services");
      bsr.setOperation("addUser");
      bsr.addRequestInput("param1", "addUser");

      String userIDRetured = (String) bsr.service();
      ...
}
```

Spring supports lookups to the business service request and other objects, including POJO managers, if any.

Figure 2 shows how the Spring configuration file, `applicationContext.xml`, supports the lookup of `businessServiceRequest` and `serviceRouter`.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
   <!-- ============================== FOR X18p ============================== -->
   <bean id="businessServiceRequest" class="x18p.framework.service.ServiceRequest" singleton="false"/>
   <bean id="serviceRouter" class="x18p.framework.service.ServiceRouter" singleton="true"/>

</beans>
```

Figure 2. Spring framework configuration. Click on thumbnail to view full-sized image.

In `ServiceRequest.java`, the `service()` method simply calls Spring to find the service router and passes itself to the router:

```
public Object service() {
  return ((ServiceRouter) this.serviceContext.getBean("service router")).route(this);
}
```

The service router in X18p routes user services to the business logic layer with `X18p-config.xml`'s help. The key point is that the `Action` code doesn't need to know where or how user services are implemented. It only needs to be aware of the rules for consuming the service, such as pushing the parameters in the correct order and casting the right return type.

Figure 3 shows the segment of `X18p-config.xml` that provides the service mapping information, which `ServiceRouter` will look up in X18p.

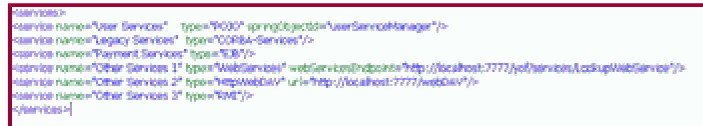


Figure 3. X18p service mapping configuration. Click on thumbnail to view full-sized image.

For user services, the service type is POJO. `ServiceRouter` creates a POJO service provider controller to handle the service request. This POJO's `springObjectId` is `userServiceManager`. The POJO service provider controller uses Spring to look up this POJO with `springObjectId`. Since `userServiceManager` points to class type `X18p.framework.UserPOJOManager`, the `UserPOJOManager` class is the application-specific logic code.

Examine `ServiceRouter.java`:

```
public Object route(ServiceRequest serviceRequest) throws Exception {


  //      /1. Read all the mapping from XML file or retrieve it from Factory
  //       Config config = xxxx;

  //      2. Get service's type from config.
  String businessServiceType =
Config.getBusinessServiceType(serviceRequest.getServiceName());

  //      3. Select the corresponding Router/Handler/Controller to deal with it.

  if (businessServiceType.equalsIgnoreCase("LOCAL-POJO")) {

    POJOController pojoController = (POJOController) Config.getBean("POJOController");

    pojoController.process(serviceRequest);
  }
  else if (businessServiceType.equalsIgnoreCase("WebServices")) {

    String endpoint = Config.getWebServiceEndpoint(serviceRequest.getServiceName());

    WebServicesController ws = (WebServicesController)
Config.getBean("WebServicesController");

    ws.setEndpointUrl(endpoint);
    ws.process(serviceRequest);
  }
```

```
    else if (businessServiceType.equalsIgnoreCase("EJB")) {
      EJBController ejbController = (EJBController) Config.getBean("EJBController");
      ejbController.process(serviceRequest);

    }
    else {
      //TODO
      System.out.println("Unknown types, it's up to you how to handle it in the framework");
    }
    //      That's it, it is your framework, you can add any new ServiceProvider for your next
project.

    return null;

  }
```

The above routing if-else block could be refactored into a Command pattern. The `Config` object provides the Spring and X18p XML configuration lookup. As long as valid data can be retrieved, it's up to you how to implement the lookup mechanism.

Assuming a POJO manager, `TestPOJOBusinessManager`, is implemented, the POJO service provider controller (`POJOServiceController.java`) then looks for the `addUser()` method from the `TestPOJOBusinessManager` and invokes it with reflection (see the code available from Resources).

By introducing three classes (`BusinessServiceRequester`, `ServiceRouter`, and `ServiceProviderController`) plus one XML configuration file, we have a service-oriented framework as a proof-of-concept. Here `Action` has no knowledge regarding how a service is implemented. It cares about only input and output.

The complexity of using various APIs and programming models to integrate various service providers is shielded from Struts developers working on the Web tier. If `x18p-config.xml` is designed upfront as a service contract, Struts and backend developers can work concurrently by contract.

Figure 4 shows the architecture's new look.

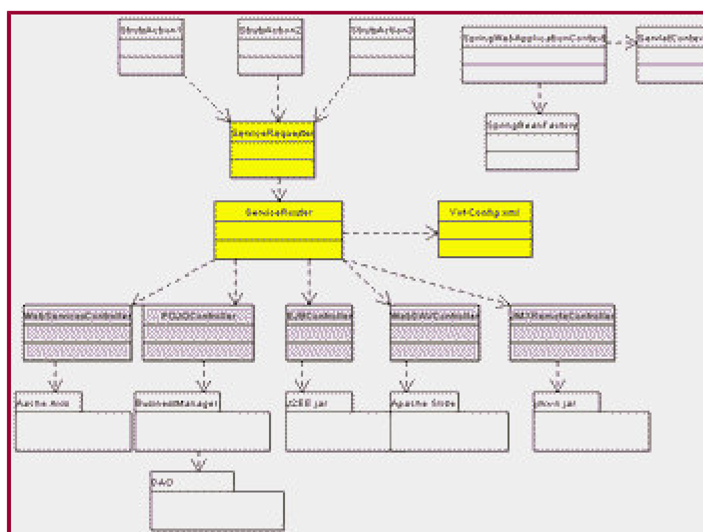


Figure 4. X18p service-oriented architecture. Click on thumbnail to view full-sized image.

I summed up the common service provider controllers and implementation strategies in Table 1. You can easily add more.

Table 1. Implementation strategies for common service provider controllers

Service type	Service provider controller	Packages
POJO	`POJOController`	J2SE
Web services	`WebServiceController`	Apache Axis
EJB	`EJBController`	J2EE
JMX	`JMXController`	M4JX

To give you an example, here's how to implement the `WebServiceController` strategy using Apache Axis:

`WebServiceController` creates an Axis `Service` object and binds all the parameters to it, invokes it, and returns. To keep it simple, it supports only type string:

```
  public Object process(ServiceRequest requester) throws Exception {
    String ret = null;

    try {
      Service service = new Service();
      Call call = (Call) service.createCall();

      String methodName = requester.getOperationName();

      call.setTargetEndpointAddress(new java.net.URL(endpointUrl));
      call.setOperationName(methodName);

      List parameters = (List) requester.getServiceInputs();
      //
      int sizeOfParameters = parameters.size();
      Object[] args = new Object[sizeOfParameters ];

      log.debug("REQUESTING Web Service:  [" + methodName + "], Inputs   ["
+  sizeOfParameters + "]");
      //TODO
      boolean isMethodFound = false;

      for (int i = 0; i < sizeOfParameters; i++) {
        int currentIndex = i;

        call.addParameter("op" + (currentIndex + 1), XMLType.XSD_STRING,
ParameterMode.IN);

        args[currentIndex] = parameters.get(currentIndex);

        log.debug("SET [" + currentIndex + "], VALUE [" +  args[currentIndex] + "]");

      }

      call.setReturnType(XMLType.XSD_STRING);
      ret = (String) call.invoke(args);

      log.debug(" Web Service: " + methodName + ", Got result : " + ret);
    }
```

```
    catch (java.net.MalformedURLException ue) {
      ue.printStackTrace();
    }
    catch (java.rmi.RemoteException re) {
      re.printStackTrace();
    }
    catch (javax.xml.rpc.ServiceException e) {
      e.printStackTrace();
    }

    return ret;
  }
```

As a matter of fact, Axis supports many other types that can be further leveraged. As can be seen above, provided you have existing knowledge of the programming model and APIs, the service provider controller should not be hard to implement.

Steps 5 and 6. Meet with user and test
Since the above X18p code is for demonstration purposes only, it is not meant to be used directly. More work must be done, but how do we find out what requirements are needed? Ask around. X18p's end-user will let you know. Adding more robust code to X18p is your responsibility. Once you know the simple architecture, discerning where to address user concerns shouldn't be difficult, right? Nevertheless, testing is indispensable and critical. X18p deserves more care.

Other improvement considerations:

- Caching at the service broker layer
- Transaction support with JDOM

DAO

Now, we turn our attention to another common area in J2EE development, DAO, and start another iteration from Steps 2 to 6.

Step 2. Analyze previous J2EE application architecture
As J2EE developers, we hate writing tedious and repetitious code. Usually we duplicate code by copying and pasting, which seems a time saver. However, chances are, something should have been changed after the copy/paste process, but wasn't. The time we saved copying and pasting is then wasted later while trying to find our error an fixing it. For a large J2EE application, without a neat framework, we spend a lot of time on the DAO code because of its creation/read/update/delete (CRUD) operations on numerous domain objects and its repetitious programming model.

To start analyzing the DAO layer, let's look closer at some JDBC code in DAO:

Stored procedure call

```
public List getReport(String s) {
  Connection connection = null;
  CallableStatement proc = null;
  ResultSet ret = null;

  try {

    Context ic = new InitialContext();
    DataSource ds = (DataSource) ic.lookup("jdbc/mydatasource");

    connection = ds.getConnection();
    connection.setAutoCommit(false);
```

```
        proc = connection.prepareCall("{ call run_monthly_report_2 (?,?,?,?,?) }");

        proc.setString(1, cdNumber);

        proc.registerOutParameter(2, Types.VARCHAR);
        proc.registerOutParameter(3, Types.VARCHAR);
        proc.registerOutParameter(4, Types.VARCHAR);
        proc.registerOutParameter(5, Types.VARCHAR);

        proc.execute();


        ret = proc.getObject(2);

        // Print the results
        while (rs.next()) {
          System.out.println(rs.getString(1) + "\t" +
            rs.getFloat(2) + "\t" +
            rs.getDate(3).toString());
        }

      }
      catch (NamingException ne) {
        ne.printStackTrace();
      }
      catch (SQLException e) {
        e.printStackTrace();
      }
      finally {
        try {
          if (proc != null) {
            proc.close();
            proc = null;
          }

          if (connection != null && !connection.isClosed()) {
            connection.close();

            connection = null;
          }
        }
        catch (Exception e) {
        }

      }
      return ret;
    }
```

A standard SQL statement call:

SQL call

```
try {
    String url = "jdbc:oracle:thin:@192.168.0.1:1521:orcl";
    Connection conn = DriverManager.getConnection(url,"","");
    Statement stmt = conn.createStatement();
    ResultSet rs;

    rs = stmt.executeQuery("SELECT u.name FROM user WHERE age = ? and sex =?");
    while ( rs.next() ) {
```

```
  String lastName = rs.getString("Lname");
  System.out.println(lastName);
   }
   conn.close();
} catch (Exception e) {
   System.err.println("Got an exception! ");
   e.printStackTrace();

}
```

Other third-party integration code follows a programming model similar to the code above, which opens a connection, gets data from the connection, and processes the data. I picked out some sample code from webMethods (a B2B server), Livelink (a document management COTS product), LDAP (standard J2SE Java Naming and Directory Interface API), Documentum (another document management COTS product), and Hibernate (an O/R (object/relational) mapping tool):

webMethods B2B server

```
iimport com.wm.util.Table;
import com.wm.data.*;
import com.wm.util.coder.IDataCodable;
import com.wm.app.b2b.util.GenUtil;
import com.wm.app.b2b.client.Context;
import com.wm.app.b2b.client.ServiceException;

public class WebMethodsCall {
  public static void main(String[] args) {
    String server = "Fangjian:5555";
    Context context = new Context();
    String username = "user";
    String password = "manage";

    try {
      context.connect(server, username, password);
    }
    catch (ServiceException e) {
      System.out.println("\n\tCannot connect to server \"" + server + "\"");
      System.exit(0);
    }

    try {
      callBuilnInService(context);
      context.disconnect();
    }
    catch (IOException e) {
      System.err.println(e);
      e.printStackTrace();
    }
    catch (ServiceException se) {
      System.err.println(se);
      se.printStackTrace();
    }
    System.exit(0);
  }

  public static final void callBuilnInService2(Context context) throws IOException,
ServiceException {
    IData in = IDataFactory.create();
```

```
      IDataCursor idc = in.getCursor();

      idc.insertAfter("$dbAlias", "SAMPLEDevp");
      idc.insertAfter("$dbSchemaPattern", "SAMPLEDEVP");
      idc.insertAfter("$dbTable", "XREFSOMETHING");
      idc.insertAfter("$dbAlias", "SAMPLEDevp");
      idc.insertAfter("$dbSQL", "select * from xcompanycode");

      IData criteria = IDataFactory.create();
      idc.insertAfter("$data", criteria);
      idc.destroy();

      IData outputRecord = context.invoke( in, "servername", "inputRecord");
      IDataCursor odc = outputRecord.getCursor();

      if (odc.next("results")) {
        com.wm.util.Table t = (com.wm.util.Table) odc.getValue();

        IData ii = t.getIData();
        GenUtil.printRec(ii, "Output Table's IData");

        IData i = t.getRow(0);
        IDataCursor idc3 = i.getCursor();

        if (idc3.first("abc")) {
          String iata = (String) idc3.getValue();
          System.out.println(">>> OK, got data : >> " + iata);
        }
      }
      else {
        System.out.println(">>> OK, resutls not found >>> \n");
      }

      if (odc.next("$dbMessage")) {
        String s = (String) odc.getValue();
        System.out.println(">>> OK, dbmssage >> " + s);
      }
      else {
        System.out.println(">>> OK, dbmssage not found >> ");
      }
    }

}
```

LDAP/JNDI API call

```
import java.util.*;
import javax.naming.*;
import javax.naming.directory.*;

public class LDAPSearch {
  public static String INITCTX = "com.sun.jndi.ldap.LdapCtxFactory";
  public static String MY_HOST = "ldap://server:389";
  public static String MGR_DN = "cn=abc,cn=users, dc=company1,dc=org1";
  public static String MGR_PW = "password";
  public static String MY_SEARCHBASE = "dc=company1,dc=company1";

  public static void main(String args[]) {
    search("abc");
  }
```

```
  public static List search(String filter) {
    filter = "cn="+filter;
    boolean isFound=false;
    try {
      Hashtable env = new Hashtable();
      env.put(Context.INITIAL_CONTEXT_FACTORY, INITCTX);
      env.put(Context.PROVIDER_URL, MY_HOST);
      env.put(Context.SECURITY_AUTHENTICATION, "simple");
      env.put(Context.SECURITY_PRINCIPAL, MGR_DN);
      env.put(Context.SECURITY_CREDENTIALS, MGR_PW);

      DirContext ctx = new InitialDirContext(env);

      SearchControls constraints = new SearchControls();

      constraints.setSearchScope(SearchControls.SUBTREE_SCOPE);

      NamingEnumeration results = ctx.search(MY_SEARCHBASE, filter, constraints);

      List returnList = new ArrayList();

      while (results != null && results.hasMore()) {
        SearchResult sr = (SearchResult) results.next();
        String dn = sr.getName();
        System.out.println("Distinguished Name is " + dn);

        Attributes attrs = sr.getAttributes();


        for (NamingEnumeration ne = attrs.getAll(); ne.hasMoreElements();) {
          Attribute attr = (Attribute) ne.next();
          String attrID = attr.getID();

          returnList.add(attrID);

          System.out.println(attrID + ":");
          for (Enumeration vals = attr.getAll(); vals.hasMoreElements();) {
            System.out.println("\t" + vals.nextElement());
          }
        }
      } // End while loop displaying list of attributes

      return returnList;
    }catch (Exception e) {
      e.printStackTrace();
      System.exit(1);
    }
    return null;
  }
}
```

Livelink call

```
import com.opentext.api.*;
import java.io.*;
import java.util.*;

public class LiveLinkDemo {

  public static void main(String[] args) {
```

```
      try {
        LLSession session;
        int volumeID;
        int nodeID;

        LAPI_DOCUMENTS documents;

        LLValue value = (new LLValue()).setAssocNotSet();
        LLValue info1 = (new LLValue()).setAssocNotSet();
        LLValue info2 = (new LLValue()).setAssocNotSet();


        //Initialize session

        session = new LLSession("fangjian", 2099, "livelink", "Admin", "password");

        documents = new LAPI_DOCUMENTS(session);

        if (documents.AccessEnterpriseWS(value) == 0) {
          volumeID = value.toInteger("VolumeID");
          nodeID = value.toInteger("ID");

          LLValue value = (new LLValue()).setAssocNotSet();

          documents.ListObjects(vID, nID, "", "", documents.PERM_SEE, value);

          LLOutputStream stream = new LLOutputStream(System.out);
          LLValueEnumeration rows = value.enumerateValues();

          while (rows.hasMoreElements()) {
            LLNameEnumeration columns = value.enumerateNames();
            LLValueEnumeration cols = rows.nextValue().enumerateValues();

            while (columns.hasMoreElements() && cols.hasMoreElements()) {

              stream.writeString(columns.nextName() + "-->");
              stream.writeValue(cols.nextValue());
              stream.writeString("\n");

            }

            int childvID = value.toInteger(i, "volumeID");
            int childID = value.toInteger(i, "ID");
          }

        }
      }
      catch (Throwable e) {
        System.err.println(e.getMessage());
        e.printStackTrace(System.err);

      }
    }
}
```

Documentum call

```
import com.documentum.fc.client.*;
import com.documentum.fc.common.*;
import com.documentum.operations.IDfFile;
```

```
import com.documentum.operations.IDfImportNode;
import com.documentum.operations.IDfImportOperation;
import com.documentum.com.*;

public class DocumentumSearch {

  public List executeSearchL(String dqlstring) {
    IDfSession session = null;
    IDfCollection idfCollection = null;

    DfClientX clientx = new DfClientX();
    IDfQuery dqlQuery = clientx.getQuery();

    IDfSessionManager mgr = clientx.getLocalClient().newSessionManager();
    IDfLoginInfo loginInfoObj = clientx.getLoginInfo();
    loginInfoObj.setUser("john");
    loginInfoObj.setPassword("password");
    mgr.setIdentity("dobcasename", loginInfoObj);

    try {

      session = mgr.getSession("dobcasename");


      dqlQuery.setDQL(dqlstring);

      idfCollection = dqlQuery.execute(session, IDfQuery.READ_QUERY);
      List ret = new ArrayList();

      while (idfCollection.next() == true) {
        for (int i = 0; i < collection.getAttrCount(); i++) {
          IDfAttr attr = collection.getAttr(i);

          ret.add(collection.getString("r_object_id"));
        }
      }
      return ret;
    }
    catch (DfException ed) {
      ed.printStackTrace();
    }
    catch (Throwable e) {
      e.printStackTrace();

    }
    finally {
      if (idfCollection != null) {
        try {
          idfCollection.close();
        }
        catch (DfException e1) {
          e1.printStackTrace();
        }
      }

      mgr.release(session);
    }
  }
}
```

Hibernate

```
import net.sf.hibernate.*;
import java.util.*;

public class HibernateDemo {
  net.sf.hibernate.Session session = null;

  public HibernateDemo() {
  }

  public void someMethod() {
    try {
      session = HibernateUtil.currentSession();
      Transaction tx = session.beginTransaction();

      MyObject object = new MyObject();
      object.setter1("abc");

      session.save(object);

      tx.commit();

      HibernateUtil.closeSession();

    }
    catch (Throwable e) {
      e.printStackTrace();
    }
  }

  public List getObjectList() {

    List ret = new ArrayList();

    try {
      session = HibernateUtil.currentSession();
      Query query = session.createQuery("select a from Acount as a");

      for (Iterator it = query.iterate(); it.hasNext();) {
        Account account = (Account) it.next();
        ret.add(account);
      }
    }
    catch (HibernateException hbe) {
      hbe.printStackTrace();
    }
    return ret;
  }
}
```

What do these examples have in common? They follow a strikingly similar pattern. First, a connection object is created. Then, input parameters are passed in to execute an operation. Lastly, the raw return data is processed and a more generic domain object or Java collection object returns to the caller. They also have typical try/catch blocks. As an O/R mapping tool, Hibernate reads/writes Java objects directly and executes an HQL (Hibernate Query Language) statement.

Due to the nature of CRUD at the DAO layer, we could face numerous CRUD operations in many DAO objects depending on the relation model. For instance, a financial application might have

`AccountDAO`, `QuoteDAO`, and `PricingDAO`. An enterprise content management application might have `DocumentDAO`, `FolderDAO`, and `ReportDAO`— not to mention commonly used `UserDAO`, `GroupDAO`, `RoleDAO`, and legacy stored procedures.

The above code clearly shows that a more desirable approach would extract the common code such as the open/close connection, parameter binding, and try/catch block to a single place in X18p. We leave the processing of raw return data to application code as it contains specific logic. We can provide a Java interface (`Contract`) for that step.

Step 3. Compare alternative frameworks
Keeping in mind that open source frameworks are the basis of our inspiration, I found three interesting items:

- A JDBC framework outlined by Ryan Daigle in *JavaWorld*
- Spring's `JDBCTemplate` and stored procedure
- Apache Cocoon's SQLProcessor

I quickly analyze these technologies in Table 2.

Table 2. Comparison of alternative frameworks

Source	Pros	Cons
Daigle's JDBC framework	Very easy to follow	Not declarative
Spring	Supports SQL and stored procedures	Not declarative, a little complicated
Apache Cocoon's SQLProcessor	Declarative	Limited to SQL statement; unfortunately deprecated and no longer supported

Due to the fact these three technologies are from disparate sources, they either lack a declarative approach or fail to consistently deal with the programming model that opens a server connection, uses it to get data, and processes data. However, we can leverage their concepts in developing a DAO support module for X18p.

Daigle's JDBC framework lays good foundation for us to extend. As you will see, the X18p `JdbcSQLProcessor` resembles that framework's `SQLProcessor`.

Step 4. Develop and refactor
Now we add some handy framework code to X18p that helps most backend developers. We can create `StoredProcedureProcessor`, `SQLStatementProcessor`, `HibernateProcessor`, or `DQLProcessor` as the single place to hold common code for X18p. With the processors and handlers implemented, we call a stored procedure like this:

```
List result = null;
StoredProcedureProcessor p = JdbcProcessor.getStoredProcedureProcessor();
List inputList = new ArrayList();
inputList.add("123");
try


{
   result = p.execute("get_complex_time_consuming_report_calucation", inputList);
}
catch(Throwable e)
{
   e.printStackTrace();
```

```
}
return result;
}
```

The `StoredProcedureProcessor` in X18p needs to be coded only once to complete the following tasks:

1. Automatically find stored procedure information from the configuration file (could be `X18p-config.xml`) by ID.
2. Get connection from configuration's datasource and create a callable statement and related object.
3. Take the input list from the caller, bind the input parameters, and register the output parameters by the configuration information.
4. Execute the procedure in a try/catch block.
5. Get a `ResultSetMapHandler`, which should be implemented by the application code, and use it to process the returned output value, which could be a string, number, date, or cursor. They are stored as a map.
6. Close all the resources in the end.

Figure 5 illustrates the referenced `X18p-config.xml` segment.

Figure 5. Stored procedure mapping configuration. Click on thumbnail to view full-sized image.

In addition to the stored procedure call, a SQL statement call can also be simplified. After `JdbcSQLProcessor` is implemented, calling a SQL statement becomes similar to calling a stored procedure:

```
JdbcSQLProcessor p = JdbcProcessor.getSqlProcessor();

List inputList = new ArrayList();
inputList.add(userName);

try {
  p.executeUpdate("select_user", inputList);
}
catch (Throwable e) {
  e.printStackTrace();
```

```
}
```

`JdbcSQLProcessor` in X18p is coded to do slightly different tasks:

1. Automatically find SQL statement information from the configuration file by ID.
2. Get connection from configuration's datasource and create a statement and related object.
3. Take the input list from the caller and bind the input parameters to the question mark (`?`) by the configuration information.
4. Execute the statement in a try/catch block.
5. Get `ResultSetHandler`, which should be implemented by the application code, and use it to process the returned data, which is a cursor. It is stored as a result set.
6. Close all the resources at the end.

Figure 6 illustrates the referenced `X18p-config.xml` segment.

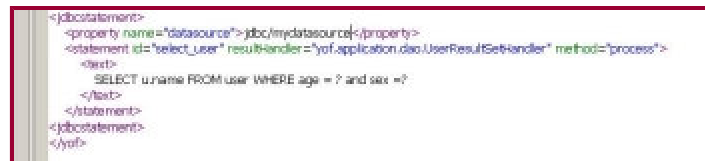


Figure 6. SQL statements mapping configuration. Click on thumbnail to view full-sized image.

If we use Hibernate, we may also implement `HibernateProcessor.java`, which maps HQL in `X18p-config.xml`. `HibernateProcessor` is consistent with `JdbcSQLProcessor` and `StoredProcedureProcessor` in terms of XML mapping.

`HibernateProcessor` can be coded to do the following tasks:

1. Automatically find HQL statement information from the configuration file by ID.
2. Get Hibernate session from configuration's datasource and create a statement and related object.
3. Take the input list from the caller.
4. With the Hibernate statement type, either bind an input parameter to a Hibernate method, such as `save()`, or bind parameter to `?` in the HQL query.
5. Execute the statement in a try/catch block.
6. Get `ResultHandler`, which should be implemented by the application code, and use it to process the returned object which could be `Query` or any name you prefer.
7. Close all the Hibernate sessions at the end.

Figure 7 illustrates the referenced `X18p-config.xml` segment for `HibernateProcessor`.

```
<hibernate-statements>
    <property name="datasource">jdbc/mydatasource</property>
    <statement id="get_account" resultHandler="x18p.application.dao.GetAccountResultHandler" method="process">
        <hql>
            select a from Acount as a :n?
        </hql>
    </statement>
    <statement id="transactions_1" type="transaction" resultHandler="x18p.application.dao.GetAccountResultHandler" method="process">
        <operation id="1">
            save(?)
        </operation>
        <operation id="2">
            saveOrUpdate(?)
        </operation>
    </statement>
</hibernate-statements>
```

Figure 7. Hibernate statement mapping configuration. Click on thumbnail to view full-sized image.

In Figure 7, other than the generic HQL mapping, one item worth noting is the transaction control configuration that `HibernateProcessor` uses to simplify the transaction code. If a transaction sequence can be predetermined, which normally is the case, we can use `X18p-config.xml` to help `HibernateProcessor` do the transaction work. We then only write the following simple code for a transaction that requires both `o1` to be saved and `o2` to be saved or updated:

```
List result = null;
HibernateProcessor p = getHibernateProcessor();

List inputList = new ArrayList();

MyObject o1= new MyObject();
MyObject o2= new MyObject();
inputList.add( o1);
inputList.add( o2);

try
{
  result = p.execute("transactions_1", inputList);
}
catch(Throwable e)
{
  e.printStackTrace();
}
return result;
}
```

The final object model, as shown in Figure 8, is simple and straightforward.

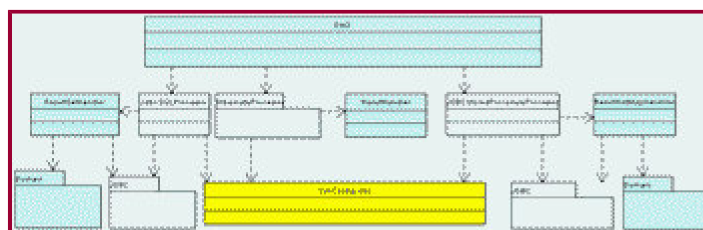


Figure 8. DAO framework architecture. Click on thumbnail to view full-sized image.

Next, as you have probably already discovered, implementing other processors and handlers on

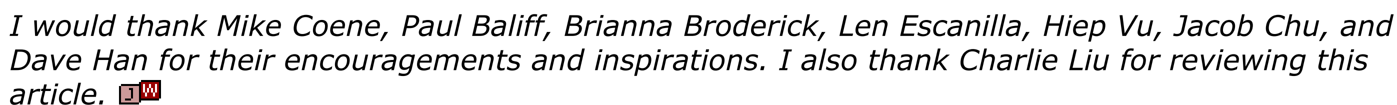
your own is not difficult. This article's code is used only for illustration purposes and is not complete for deployment. The key point is to roll your own.

Steps 5 and 6. Meet with user and test
Complete the final steps the same way we did previously.

Conclusion

In this article, you have learned a service-oriented approach for decoupling layers. Also, you discovered many ways to reduce DAO code. However, Java technology continues to evolve. One thing is for certain, there are always ways to greatly improve our productivity.

I would thank Mike Coene, Paul Baliff, Brianna Broderick, Len Escanilla, Hiep Vu, Jacob Chu, and Dave Han for their encouragements and inspirations. I also thank Charlie Liu for reviewing this article.



About the author
Fangjian Wu, a technical architect with Booz Allen Hamilton, a global consulting firm, has been developing J2EE applications during the last five years. He is currently architecting Documentum and electronic-submission J2EE applications. He has a BS in microelectronics and an MS in computer information systems. He is based in Rockville, Maryland.

Resources

- Download the source code that accompanies this article:
  http://www.javaworld.com/javaworld/jw-10-2004/soa/jw-1004-soa.zip
- Struts:
  http://struts.apache.org/
- Service Locator pattern:
  http://java.sun.com/blueprints/corej2eepatterns/Patterns/ServiceLocator.html
- Hibernate:
  http://www.hibernate.org/
- "Eliminate JDBC Overhead," Ryan Daigle (*JavaWorld,* May 2002):
  http://www.javaworld.com/javaworld/jw-05-2002/jw-0524-sql.html
- Spring Framework:
  http://www.springframework.org/
- Axis:
  http://ws.apache.org/axis/
- JDBC technology:
  http://java.sun.com/products/jdbc/
- Documentum, an enterprise content management technology:
  http://www.documentum.com/
- webMethods, a B2B integration server:
  http://www.WebMethods.com/
- Livelink, a knowledge management system:
  http://www.OpenText.com/
- Cocoon:
  http://cocoon.apache.org/
- For an introduction to Axis, read "Axis : The Next Generation of Apache SOAP," Tarak Modi (*JavaWorld,* January 2002):
  http://www.javaworld.com/javaworld/jw-01-2002/jw-0125-axis.html
- *JavaWorld* has published numerous articles on Struts, including the following:
  - "Struts Best Practices," Puneet Agarwal (September 2004)
  - "Jump the Hurdles of Struts Development," Michael Coen and Amarnath Nanduri (April 2003)
  - "Boost Struts with XSLT and XML," Julien Mercay and Gilbert Bouzeid (February 2002)
- For more articles on J2EE development, browse the **Java 2 Platform, Enterprise Edition (J2EE)** section of *JavaWorld*'s Topical Index:

http://www.javaworld.com/channel_content/jw-j2ee-index.shtml?

- For more articles on JDBC, browse the **Java Database Connectivity (JDBC)** section of *JavaWorld*'s Topical Index:
  http://www.javaworld.com/channel_content/jw-jdbc-index.shtml

Advertisement: Support JavaWorld, click here!

Copyright © 2004 JavaWorld.com, an IDG company