J2EE Best Practices Architecture Guidelines
J2EE Coding Standards

ITC Architectural Guidelines and J2EE Coding Standards

TABLE OF CONTENTS

ITC Architectural Guidelines and J2EE Coding Standards

1 INTRODUCTION

This document describes the J2EE architecture components, elements, design patterns and Java coding standards used in building Web/J2EE applications. The architecture components and design patterns discussed in this document are based on the book Core J2EE Patterns by Sun Microsystems Press. The broadly acclaimed approach and definition has become the de facto standard for building J2EE applications. In addition, this document also addresses adopted Java coding standards (i.e., class and package naming conventions, and coding styles …).

2 WEB ARCHITECTURE LAYERS

The Web Application architecture uses four basic layers (tiers) to achieve the front to back execution of a web application. The design patterns used to implement the layers are:

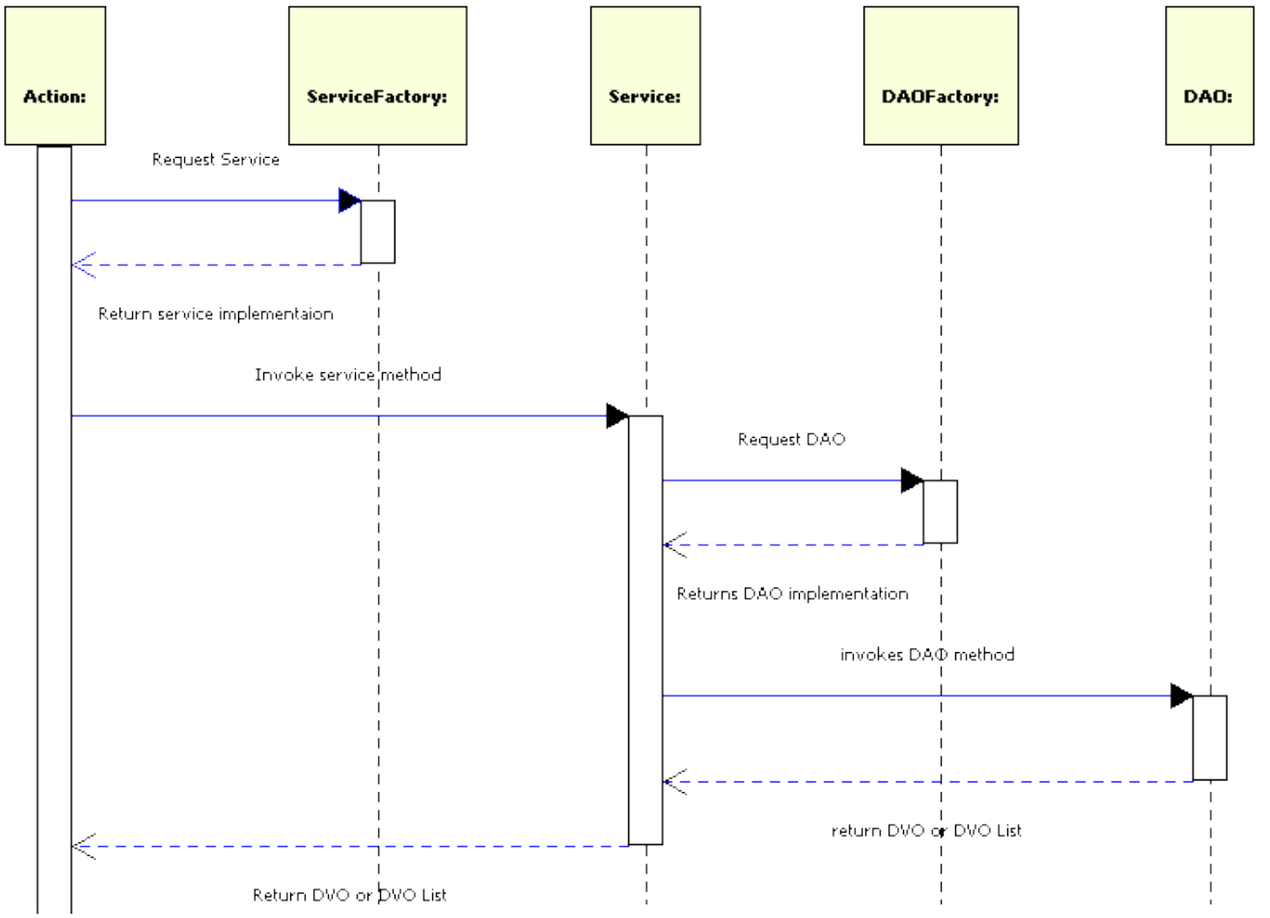
- Model View Controller II (MVC2)
- Presentation Bean Objects (PBO)
- The Service Layer – Business tier / Business Service Objects (BSO)
- Data Access Objects (DAO) - Persistence
- Data Value Objects (DVO) – As a transport container

The following rules apply to tier relationships:

- The Front Controller actions are cognizant of the Service Layer, but the services themselves are not cognizant of the Front Controller.
- The Service Layer is cognizant of the DAO Layer, but the DAO Layer is not cognizant of the Service Layer.
- The Controller Actions, Server, and DAO Layers are cognizant of the Value Objects, but Value Objects are not cognizant of any other layers.

Strictly adhering to these rules provides maximum reuse of these components.

The following sequence diagram depicts the execution flow through the architecture layers:



3 DETAIL VIEW OF ARCHITECTIURE LAYERS

3.1 Model View Controller II (MVC2)

The current MVC2 implementation for eBusiness is Struts. The Struts framework implements the Front End controller design pattern as defined in the Core J2EE Patterns. It should be noted however, that this front end view implementation can easily be substituted by many other front end input sources such as Web Services (SOA), rich graphical interfaces like Java Swing, or Eclipses' Standard Widget Toolkit (SWT). The front end actions are responsible for assembling the required object(s) required by the service method interface and placing or serializing any return objects in the format that the front end destination expects. In the case of Struts (or any Java Server Page (JSP) implementation [JSF]), objects for input to the service layer are assembled from the HttpServletRequest, HttpSession, or PageContext objects and return objects are placed in the above mentioned objects where Java Server Pages expect to find them. The following code snippet demonstrates the interaction between a Struts action and an Admin business service to retrieve a UserProfile object.

```
public void execute( ActionMapping am, ActionForm af,
                    HttpServletRequest req, HttpServletResponset resp ) throws Exception

{

  HttpSession session = req.getSession();

  AdminService adminService = ServiceFactory.getAdminService();

  // get user id from submitted form

  String strUserId = req.getParameter( "userId");

  // Get user profile from admin business service

  UserProfile userProfile = adminService.getUserProfile( strUserId );

  // Put object where jsp tablibs can find it

  session.setAttribute( "userProfile", userProfile );
```

3.2 Presentation Beans

In some cases, the DVO(s) passed back from the Service Layer have the entire set of individual data items present, but not in the format that the requesting user interface needs them in. A web application, for example, might want numerical values formatted with dollar sign and commas, but an IVR application may care about the numeric values only as they come from the service layer, or again, one web page displays a person's First and Last name as one field, and another wants needs to display it as Last name, First name. These types of

issues are not for a Business Service or DAO to decide. The Action which knows about its client is aware of this, but writing this code in the Action class breaks the rule of mixing presentation and application. A Presentation Bean is the way to solve this problem. Presentation Beans are decorators around existing DVO object(s). A presentation bean's constructor takes the DVO(s) from the Service. If a web application wants a persons First and Last names concatenated, for example, it could add a method called getName() where internally it gets the First name and Last name properties from the original DVO and returns the concatenated string of the two. It would do just the opposite when taking the same value from the web service action, say in a Save operation, and decompose the First and Last name string into the firstName and lastName properties that the business service expects them in. Presentation beans are not always necessary and should only be created when the DVO and the view of the DVO are not identical.

3.3 The Service Layer – Business tier / Business Service Objects (BSO)

The Service Layer, also known as Business Service Objects (BSO's), act as the model component in the MVC2 design pattern. The Service Layer is responsible for:

- Data validation - where rules apply to a data element's value or its null state Note: data elements that are unconditionally required, those that must be valid numeric, or date, are specified up front in java script (auto emitted by the ITC JSP tag libraries)
- Business logic processing
- Logging
- Propagate errors back to the Action
- Manage data retrieval and persistence through the Data Access Object layer (DAO)

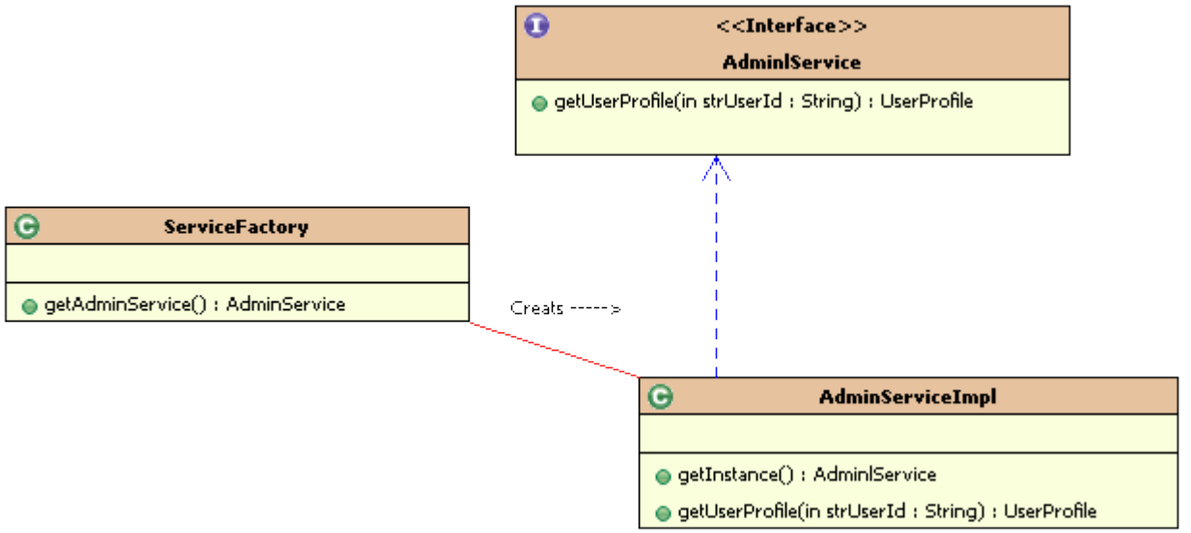
As previously stated, the Service Layer must not be cognizant of the Action Layer. A properly designed service should be able to be reused by many different client front ends. The code snippet above shows a Struts action using the Member Service to retrieve a Client object given a client number identifier. If the service were passed a servlet object like the HttpServletRequest (or had references to other HttpServletXXX objects), then this service could not be used by a different client front ends (without constant need of modifying the code each time a different front-end client was needed). Supposing this same service were to be used by an Interactive Voice Response (IVR) front end: by defining the service method to only take Value objects or Strings, we can use the same service to provide a WEB application, IVR application, and a WEB service application etc., without the need for heavy modification.

3.4 Business Service Design Patterns

All business service objects should implement an interface class that defines the contract for the service implementation class. Business services should always be obtained from a factory object that knows the specific implementation class to instantiate. The service factory always returns the interface class to the requesting client. This allows different service implementations to be used without having to re-factor code.

Unless there is a very compelling need to maintain state on the service object, services should use the singleton design pattern to avoid unnecessary object pollution. The service implementation class should define a public static final synchronized method called getInstance (which may or may not have parameters).

The following UML diagram illustrates the relationships between the Factory, interface and implementation objects.



Notice that the diagram illustrates the standard naming conventions for the factory, interface and implementation objects. The interface name semantically names the object. That it represents. The implementation object always starts with the interface name and adds the suffix “Impl”. If there are (or could be) multiple implementations for the interface then the implementation class should incorporate in its class name an additional qualifier. Let’s say for example that a service could handle both local and remote operations. The naming example would then change to something like AdminServiceImpl . The ServiceFactory in this case would contain the knowledge to know which implementation class to create when the getAdminService() method is invoked. All service methods should throw an Exception based object if the service encounters any error that prevents the request from successfully completing. The following code snippet illustrates the AdminService UML example above; the code snippet also demonstrates the singleton design pattern.

```
public class AdminServiceImpl implemnts AdminlService

{

  private static Adminlervice s_instance = null;


  // private constructor to insure singleton integrity


  private AdminService()
```

```
  {
    ; // constructor code (if needed)
  }
  public static final synchronized AdminService getInstance()
  {
    if ( s_instance == null )
      s_instance = new AdninService();
    return s_instance;
  } // end getInstance()
  // implements the getFoo Method here
  public UserProfile getUserProfile( String strUserId ) throws Exception
  {
       // Get Foo object from DAO
     AdminDAO dao = DAOFactory.getAdinDAO();
     return dao.getUserProfile( strUserId );
  }
}
```

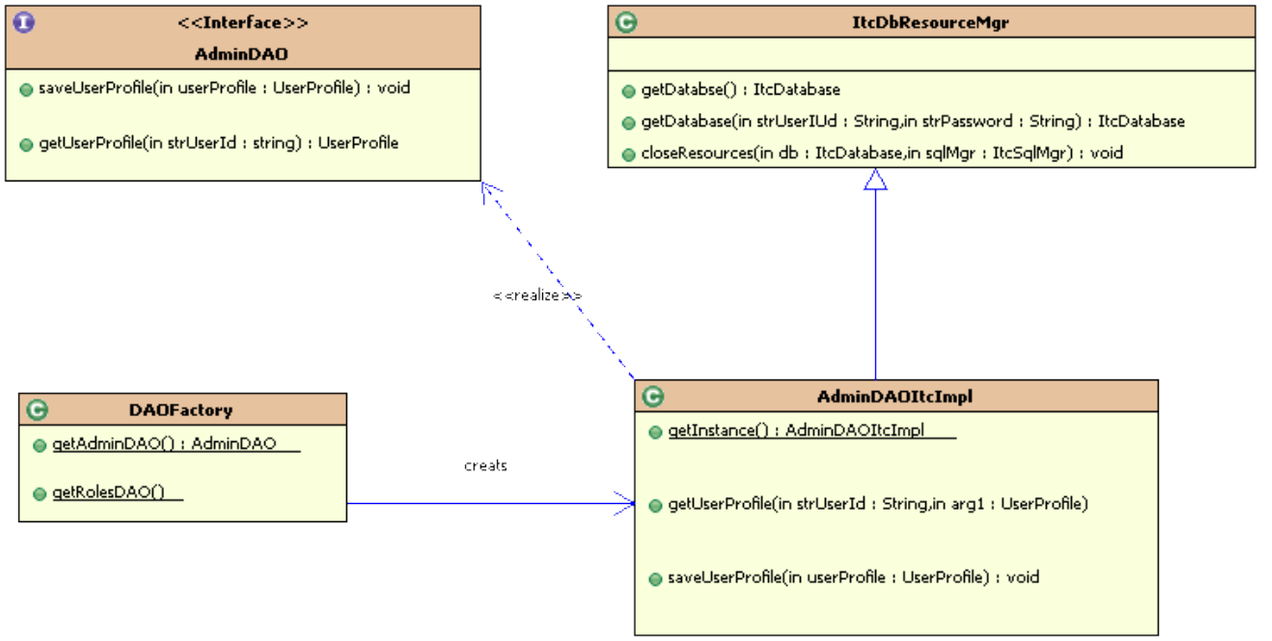
3.5 The Data Access Layer – Data Access Objects (DAO)

The Data Access Layer manages all persistence. The Service Layer should not know how objects are retrieved, stored, or where the persistence lives. Data can be stored in many different forms and should not assume relational database model. XML documents, flat files, property files, and other web sites are often other sources of data. To summarize, the DAO layer is responsible for:

- Handling all retrieval and storage requests from a service
- Providing a high level object interface to the service layer
- Propagating IO errors back to the service layer
- Provide journaling requests where needed
- Manage lower level transactions where needed

DAO classes like the BSO classes are typically implemented as singletons. DAO request methods should be atomic and not maintain state where possible as these resources tend to be more expensive in both memory and connection availability (for database DAO's). DAO methods should provide the same high level object interface to the business services as the services do for action handlers. What do we mean by this? Let's take for example the UserProfile object used in the GetLoginAction code example shown above. The service retrieved a Client object which is really an object graph composed of Member, ClientAddress and Address objects. This is because the DAO that handled the getClient request built the objects from the Client, member, ClientAddress, and Address objects. Likewise, when a save request is needed, the DAO is capable of breaking down the Client object graph and inserting or updating the tables associated with this object. (This was actually accomplished with the ***itcworks*** open source library which among many things handles object relational mapping and sql code/DVO generation from reverse engineering database tables) In addition, if journaling is needed, the DAO would add entries to the journaling table(s) inside a transaction bracketed around the saving of the Client and associated tables.

All DAO methods must throw Exception based objects when persistence fails. This allows the service layer to pass errors back to the action layer which in turn can present the client user with the appropriate feedback. The UML diagram below shows the DAO design pattern:



The following code snippet demonstrates a DAO implementation that retrieves the Client object graph for the member number requested. It also illustrates the high level interface to relational databases using the itcworks open source library. The AdminDAO.xml contains all of the SQL mappings for inserting, updating, deleting and

fetching objects from database tables and stored procedures. This document is generated by the ItcObjectSQLMapper utility (itworks open source) and is used by the ItcSqlMgr to complete its database requests. For more detailed information on itcworks, visit the site at www.i-techcorp.com or download at http://sourceforge.net/projects/itcworks.

The code snippet below demonstrates a sample DAO implementation extending from the ItcDbResourceMgr.

```
public class AdminDAOItcImpl extends ItcDbResourceMgr implements AdminDAO
{
  private static AdminDAOItcImpl s_instance;

  /**
   * Constructs an instance of this class.
   * <p>
   * This constructor is marked private to
   * prevent instantiation outside the confines
   * of this class.
   *
   * @throws  Exception if an error occurs on construction
   */
  public AdminDAOItcImpl( ItcResourceMgr mgr ) throws Exception
  {
    super("UDB", mgr.getString( "dao.urlId" ), "AdminDAO.xml");
  }


  /**
   * Retrieves an instance of this class.
   *
   * @return  an instance of this class
   * @throws  Exception if an error occurs retrieving the instance
   */
  public static AdminDAOItcImpl getInstance() throws Exception
  {
    if ( s_instance == null)
    {
      s_instance = new AdminDAOItcImpl();
    }

    return s_instance;
  }









  /**
   * Gets a UserProfile object for the requested user id
   *
   * @param strUserId The user id for the UserProfile to retrieve
   *
   * @return a UserProfile object for the user id or null if the user Id is not valid
   */
  public UserProfile getUserProfile( String strUserId ) throws Exception
  {

    try
    {
```

```
      ItcDatabase db = getDatabase();        // Get a database instance from the db resource manager
      ItcSqlMgr sqlMgr = getSqlMgr(db);      // Get ItcSqlMgr used to carry out database requests
      // Fetch the client object graph
      UserProfile userProfile = sqlMgr.findByPrimaryKey( UserProfile.class, strMemberNbr );
    }
    finally
    {
      // standard resource cleanup
      closeResources(db,sqlMgr);
    }

    return userProfile;
  }
```

4 JAVA CODING STANDARDS

For the most part, the Sun java coding standards "*Code Conventions for the Java Programming Language" found* at http://java.sun.com/docs/codeconv/ are used. The additional naming conventions are defined below.

4.1 Java package naming conventions

All java packages should start with a unique identifier followed by the application acronym/ abbreviation followed by the semantic category. The application uses the acronym so all of its package names start with these unique acronyms (i.e. com.mycompany.application). A list of the currently defined categories follows:

- ***actions*** All of the action classes for the application
- ***bso*** business service objects
- ***dao*** Data access objects
- ***dvo*** Data Value objects
- ***pbo*** presentation bean objects
- ***util*** common utility objects

Examples : My company's payroll application

com.mycompany.pr.actions Struts defined action classes

com.mycompany.pr.dvo Data Value Objects used by DAO, business services and JSP pages

4.2 Variable Naming Conventions

Class member variables should all start with the m_ prefix to denote these variables as instance variables as opposed to local variables. The s_ prefix should be used to denote all static defined variables.

4.3 Standard variable notation prefixes

In the early days of Microsoft Windows, a very popular self describing prefix notation was developed and was named Hungarian notation after the Microsoft engineer (who was Hungarian) published code recommendations. This style was promoted by the well known author Charles Petzold in his books Programming Windows and Programming OS/2. This notation greatly aids in the readability of code, especially during the maintenance phase when many different developers work with many different source files. The prefixes to declare the basic data types are summarized below:

- ***str*** String Data Type
- ***b*** byte data type
- ***f*** Boolean or flags
- ***ch*** char data type
- ***n*** int data type
- ***l*** long data type
- ***flt*** float data type
- ***dbl*** double data type
- ***dt*** date data type
- ***map*** map object
- ***list*** List object
- ***ht*** HashTable object
- ***vec*** Vector object

To declare an array of these base types, a lower case ‘a’ is placed in front of the base prefix.

Examples:

```
m_nCounter          // class variable of type int
m_strFirstName      // class variable of type String
s_mapProps          // static class variable that represents a Map
adblBalanace        //  local or parameter variable that is an array of doubles
afValid             //  local or parameter variable that is an array of Boolean
m_listClient        //  Class instance variable List of Client objects
```

4.3.1 4.4 Unit testing

JUnit is the standard for unit testing Java classes. At a minimum, all business service and data access classes are required to have JUnit test suites defined. The package naming convention is the same as those listed in section 4.1 with the exception that JUnit packages start with test. instead of org. The class names should also start with Test followed by the actual call name the test case is for. Using the DAO example shown above, a JUnit test for the AdminDAOItcImpl class would be named TestAdminDAOItcImpl and would reside in the package test.mycomany.pr.dao. All methods defined in the BSO and DAO classes MUST have corresponding test methods. The following code snippet shows a JUnit tester for the AdminDAOItcImpl class.

```
public TestAdminDAOItcImpl extends TestCase

{

  static AdminDAOItcImpl      s_dao = null;

  // tester for the getMember DAO defined method

  public void testGetMember()

  {

    Client client = s_dao.getMember( "123456678" );

    assertNotNull( "Did not expect client to be null for number 12345678", client );

    assertTrue('Expected last name to be Jones but got" + client.getLastName(),

               client.equals( "Jones" ));

 }

  // Define the static suite method. Every tester needs to have one of these

 public static Test suite()

 { return new TestSuite( TestAdminDAOItcImpl.class ); }

  public void static main( String[] astrArgs )

  {

     // create the DAO instance from factory

     s_dao = DAOFactory.getAdminDAO();

     // Run the test suite

     TestRunner.run( suite() );

  }

}
```

4.3.2 Java server pages (JSP) standards

General coding rules

All JSP's start with the file include jspbegin.jsp and end with the file include jspend.jsp. The jspbegin.jsp page must be the first line in the jsp page (except for comments) and is included the following way:

<%@include file="jspbegin.jsp" %>

jsp content goes here ….

<%@include file="jspend.jsp" %>

The jspbegin.jsp defines the taglibrary uri's needed for the itc and struts taglib extensions. The jspbegin.jsp also starts the first part of the try catch block that is used to trap unchecked exceptions so that they may be logged and properly displayed. The jspend.jsp closes the try block with the catch block so failure to include this page will result in runtime compile errors by the servlet container.

Inline java code through the use of scriptlet tags <% %> is prohibited. All logic to handle content should be done in the action classes and not in the java server page. This maintains the proper coding practice of separating application from presentation. Simply put, a java server page displays bean properties from beans created by the action and uses the logic and iteration tags provided by the taglib extensions to govern the content sent back to the browser. A properly written java server page should look like a well formed XML document.

The ***itc*** and ***jstl/structs*** tablib extensions provide the functionality for logic testing, iteration, internationalization and display bean properties. The use of Java script should be kept to a minimum. The itc taglibs auto emit java script to provide for required field checks, numeric and valid date content checks.

Internationalization

To comply with the internationalization needs of applications, no static text should be placed in the JSP page. All static text representing screen prompts and messages must be placed in resource bundles with the proper language extensions. The <itc:rb> displays the resource bundle keys. The following snippet shows the wrong and right way to code for internationalization:

Wrong way:

Last Name <itc:write name="member" property="lastName"/>

Right way:

<itc:rb key="prompt.lastName"/> <itc:write name="member" property="lastName"/>**