Applying Aspect Orientation to J2EE Business Tier Patterns

Therthala Murali
murali_therthala@yahoo.com

Renaud Pawlak
pawlakr@rh.edu

Houman Younessi
houman@rh.edu

Rensselaer Polytechnic Institute- Hartford Graduate Campus
275 Windsor St, Hartford, CT 06120 USA

ABSTRACT

J2EE Design Patterns [1] offer flexible solutions to common software problems encountered in the design and construction of distributed systems for the J2EE platform. A number of J2EE patterns involve crosscutting structures in the relationship between the roles in the pattern and classes in each instance of the pattern, thus making the resulting components increasingly complex. This complexity is at odds with one of patterns' key goals - to make it easier to build simple, elegant and high-quality systems that work. This paper analyzes the problem of crosscutting within the implementation of J2EE patterns in the Business Tier and demonstrates how Aspect-Oriented techniques can be used to generate improvements within the business layer components from the perspective of better code locality, reusability, composability and (un)pluggability.

1. INTRODUCTION

Software Patterns are designed to communicate expert knowledge about system construction. Useful Patterns address structural problems and are carefully written to be readable.

Prior research [2] shows that aspect-based implementations of the GoF design patterns showed modularity improvements in 17 of 23 cases. These improvements were in terms of better code locality, reusability, composability and (un)pluggability. These results suggest that it would be worthwhile to undertake the experiment of applying aspect-oriented techniques to J2EE pattern implementations.

Constructing an application under the J2EE platform involves the assembly/composition of prefabricated, reusable and independent components. The J2EE design patterns [1] offer flexible solutions to construct high-quality, reusable, evolvable components for the J2EE platform. While a lot of the J2EE pattern literature is focused on highlighting the benefits of the J2EE applications constructed using the patterns, there is hardly any discussion of how the patterns have introduced code tangling and code scattering within the core functionality of the J2EE components.

In our study, we highlight the problems caused by code scattering and code tangling within the J2EE business tier due to the implementation of the patterns. We develop and compare Object-Oriented and Aspect -Oriented implementations of the J2EE patterns for this tier. We retain the purpose, intent and applicability of the J2EE patterns but only allow the solution structure and solution implementation to change.

The rest of the paper is organized as follows. Section 2 highlights some of the problems created within the business tier due to the implementation of J2EE design patterns and present them. Section 3 introduces the format of study that we have undertaken. In section 4, we present our AspectJ implementations for some patterns and highlight the improvements we observed. Section 5 summarizes our work.

2. CHALLENGES

2.1 Established Challenges

The three major problems [2] in systems realized using patterns are related to pattern implementation, pattern documentation and pattern composition.

Implementations of patterns are often governed by the instance of use and context owing to the fact that pattern implementations heavily influence system structures and vice versa [3]. This makes it hard to distinguish between the pattern, its concrete instance and the pertinent object model [4]. Changes to a pattern within a system are often invasive and tedious. Consequently, while the design pattern is reusable, its implementations usually are not [2].

As stated in [2], the impacts of design patterns on programs are of two different natures. In the first case, they can *superimpose roles*. An initial functional class could be enhanced to define a role in the design pattern. In the second case, they could add new classes to the program that are independent from the initial functional program and *define new roles*. In both the cases, the design patterns are not completely modular. In the first case, the design pattern implementation is invasive since it modifies a class of the initial program. In the second case, the newly created role has to be used eventually by a class of the functional program and this reflects the existence of an associated superimposed role.

Thus, non-modularization in the business layer of the J2EE applications due to patterns introduces code scattering and code tangling within the program. Code Scattering is caused because several instances of the patterns or of a given role will be used within several classes of the program. Code tangling occurs when several pattern or role instances overlap in a single class. This last effect is particularly troublesome because, when a particular class is involved in more than one pattern, it becomes difficult to compose the patterns together because the structure of the application becomes less straightforward. Moreover, documenting the patterns and their participants within the application also becomes cumbersome.

2.2. Crosscutting in J2EE Patterns

This section presents the standard J2EE design patterns and discusses the problems caused by their implementation in terms of crosscutting.

2.2.1 Business Delegate

The Business Delegate hides the underlying implementation details of the business service, such as lookup and access details of the EJB Container and JNDI Directory Services and thereby reduces the coupling between presentation-tier clients and business services. However, interface methods in the Business Delegate may still require modification if the underlying business service API changes. The reference to the Business Delegate layer, within every client that accesses the business services layer, is a crosscutting concern. While location transparency is one of the benefits of this pattern, a different problem may arise due to the developer treating a remote service as if

it was a local one. This may happen if the client developer does not understand that the Business Delegate is a client side proxy to a remote service. Typically, a method invocation on the Business Delegate results in a remote method invocation under the wraps. Ignoring this, the developer may tend to make numerous method invocations to perform a single task, thus increasing the network traffic.

It would be worth exploring if there exists a way to leverage the advantages offered by the business delegate layer, without implementing the business delegates and eliminating the coupling with the client.

2.2.2 Service Locator

The Service Locator pattern reduces the client complexity that results from the client's need to perform lookup of distributed services and their creation, which are resource-intensive. However clients that use the Service Locator are faced with a plethora of crosscutting problems.

A client of the Service Locator such as a Business Delegate has to explicitly reference the interfaces (`javax.ejb.EJBHome` and `javax.ejb.EJBLocalHome`) within the javax.ejb package and the exceptions within the `javax.ejb`, `java.rmi` and the `javax.naming` packages. The clients ought to capture these exceptions and handle them appropriately. The references to the interfaces and classes are a crosscutting concern.

A reference to the Service Locator within the client that needs to lookup services is in itself a crosscutting concern. The Service Locator is an implementation of the GoF Singleton pattern and has a private constructor. Hanneman et al [2] have demonstrated how a plain old java object can be turned into a Singleton by weaving into it, the Singleton Protocol via an aspect. It would be worth applying this idea to the Service Locator to see if it offers any advantages within the J2EE world.

2.2.3 Transfer Object

When clients require more than one value from the business services layer, it is possible to reduce the number of remote calls to the Session Façade and to avoid overhead by using Transfer Objects to transport the data from the enterprise bean to its client.

In order to be transportable over the wire via Java's Remote Method Invocation (RMI), the Transfer Objects have to implement the `java.io.Serializable` interface. If a client that is located within the same virtual machine as the Session Facade, desires to invoke the same business service,the client need not invoke the service via RMI and hence the implementation of `java.io.Serializable` by the Transfer Object becomes redundant.

The Client and the Session Facade that use Transfer Objects reference these objects within their implementations. Thus it would be worth investigating whether the benefits offered by Transfer Objects can be obtained, without them implementing the `java.io.Serializable` and also not cross-cutting the client and Session Facade implementations.

2.2.4 Session Facade

The Session Facade in a J2EE application is usually a Session Enterprise Bean that manages the business objects, and provides a uniform coarse-grained service access layer to the clients. The benefits of a facade have been highlighted in the GoF literature and also in Core J2EE Patterns [1]. The Session Facade bean ought to implement the `javax.ejb.SessionBean` interface.

It would be worth exploring whether the Session Facade can be made to leverage the features of the EJB Container by realizing it as a plain old java object (POJO) and without implementing the `javax.ejb.SessionBean`.

2.2.5 Transfer Object Assembler

The Transfer Object Assembler can be a POJO or a Session Facade. If the Transfer Object Assembler is implemented as a Session Facade, then the problems discussed in section 2.2.4 for the Session Facade would apply.

2.2.6 Value List Handler

The Value List Handler can be a POJO or a Stateful Enterprise Session Bean. In either of the implementations, the Value List Handler is coupled to the Value List Iterator interface. If a Session Bean, the Value List Handler becomes tied to the `javax.ejb.SessionBean` interface and the problems discussed in section 2.2.4 for the Session Façade would apply.

2.2.7 Composite Entity

The Composite Entity's implementation of the Entity Bean interface is a crosscutting concern and is not beneficial from a system adaptability standpoint. It would be worth pursuing the realization of the Composite Entity as a POJO, without implementing the `javax.ejb.EntityBean` but still leveraging the container managed persistence features.

2.2.8 Application Service

The Application Service is usually a POJO and is implemented either as a Command pattern or as a Strategy pattern. The problems of Command and Strategy have been highlighted [2] and we shall implement the Application Services using the AspectJ versions of Command and Strategy as demonstrated in [2].

2.2.9 Business Objects

The Business Objects are usually implemented either as POJOs or as Enterprise Entity Beans. When realized as POJOs, they are implemented by composing any of the GoF patterns depending on the problem domain. In such a scenario, their AspectJ implementations could be realized as outlined in [2]. When realized as Enterprise Entity Beans, the BusinessObject has to implement the `javax.ejb.EntityBean` interface. Hence if the BusinessObject is to be reused in another J2EE application that does not use Entity Beans, the BusinessObject becomes useless and needs to be converted to a POJO.The implementation of the javax.ejb.`EntityBean` interface by the BusinessObject is a crosscutting concern and does not facilitate seamless component adaptation. It would be worth pursuing the realization of the BusinessObject as a POJO, without implementing the `javax.ejb.EntityBean` interface and yet leveraging the EJB's container-managed persistence features.

3. STUDY FORMAT

The methodology for study involved the design and implementation of a contrived distributed application in accordance with the J2EE specification on a J2EE platform using J2EE patterns, first using the classical Object-Oriented approach and later employing aspects using AspectJ 1.1.4. The core business model of the application provides a Currency component that performs conversion between currency values as shown in the interface listing below.

```
public interface ICurrency {
  public double dollarToPound(
    double aDollarValue) throws
      TooLargeValueException,RemoteException;
  public double dollarToEuro(
    double aDollarValue) throws
      TooLargeValueException,RemoteException;
  public CurrencyTO getCurrencyTable()
    throws RemoteException;
  public CurrencyTO getCurrencyByCountry()
```

```
    throws RemoteException;
  public String getUsCurrency()
    throws RemoteException;
  public String getUkCurrency()
    throws RemoteException;
  public String getFranceCurrency()
    throws RemoteException;
  public String getPolandCurrency()
    throws RemoteException;
}
```

The application's business tier is fronted by EJB Session facades while the client tier consists of java application clients. The application was packaged and deployed on Sun ONE Application Server. The Java implementations correspond to the samples presented in the Core J2EE Patterns book [1]. Each J2EE pattern has a number of implementation variants and alternatives. If a pattern offered more than one possible implementation, we picked the one that seemed the most widely used. Our modularization goals in implementation of J2EE patterns using AspectJ were consistent with those in [2]. In this paper, we will mainly focus on the aspectized implementation of the Business Delegate, the Service Locator, and the Transfer Object patterns.

4. RESULTS

This section presents a comparison of the aspect-oriented and pure object-oriented implementations of concrete instances of the J2EE Business Tier patterns. We focus on the Business Delegate, the Service Locator, and the Transfer Object patterns.

4.1 Business Delegate and Service Locator

In the classical implementation, the Business Delegate pattern manages the complexity of distributed component lookup and exception handling for the calling client, yet the reference to the delegate within the client's implementation is a crosscutting concern. The delegate's presence is truly valuable only when invoking a remote service.

The following code shows the implementation of a typical client. It explicitly uses the Business Delegate that uses the Service Locator pattern.

```
public class TestClient {
  public static void main(String[] args) {
    try  {
      // delegate is used here
       CurrencyDelegate delegate =
        new CurrencyDelegate();
      logger.debug(
        delegate.dollarToPound(10.0) + "GBP");
      // a transfer object is used here
      // it reduces network traffic
      CurrencyTO to =
        delegate.getCurrencyByCountry();
      logger.debug(" US Currency -> " +
        to.getUsCurrency());
    } catch(Throwable t) {
        [...]
```

The following code sample shows the implementation of Currency Delegate using regular Object-Orientation. It has to lookup distributed services by using the Service Locator and deal with the exceptions that can be thrown by the invocation of remote services. In a real application, several delegates are created, usually one per facade. This introduces crosscutting within the clients and a dependence on the JNDI (Java Naming and Directory Interface) and EJB technologies that reduces the adaptability of the application.

```
public class CurrencyDelegate {
  private static ServiceLocator locator;

  private void init() throws SystemException {
    try {
      locator =  ServiceLocator.getInstance();
    } catch(NamingException ne) {
      throw new SystemException(
        ne.getMessage());
    }
  }

  private Currency getServiceFacade()
    throws SystemException {
    Currency currency = null;
    try {
      CurrencyHome home = (CurrencyHome)locator
        .lookupHome(Currency.class);
      currency = home.create();
    } catch(ClassNotFoundException cne){
      throw new SystemException(
        cne.getMessage());
    } catch(NamingException ne) {
       throw new SystemException(
         ne.getMessage());
    } catch(CreateException ce) {
       throw new SystemException(
         ce.getMessage());
    } catch(RemoteException re) {
      throw new SystemException(
        re.getMessage());
    }
    return currency;
  }

  public CurrencyDelegate()
    throws SystemException {
    if(locator == null)
    init();
  }

  public double dollarToPound(double aValue)
    throws SystemException {
    Currency currency = getServiceFacade();
    try {
      return currency.dollarToPound(aValue);
    } catch(RemoteException re) {
      throw new SystemException(
        re.getMessage());
    } catch(TooLargeValueException te) {
      throw new SystemException(
        te.getMessage());
    }
  }
  // same principle with other delegating
  // methods
  [...]
```

The code snippet below shows how the `ClientAspect` and the `LocatorAspect` combine to make the Business Delegate obsolete. The pointcuts and their corresponding advices provide the necessary J2EE plumbing that enables a plain java client to invoke the business services offered by components within the EJB container.

```
public class TestClient {
  public static void main(String[] args) {
    try {
      TestClient client = new TestClient();
      ICurrency currencyService =
        (ICurrency)client.getServiceFacade(
           Currency.class);
      logger.debug("GB POUNDS -> " +
        currencyService.dollarToPound(10.0));
      logger.debug("GB CURRENCY -> " +
        currencyService.getUkCurrency());
      logger.debug("US CURRENCY -> " +
        currencyService.getUsCurrency());
    } catch(Throwable t) {
      t.printStackTrace();
    }
  }

  public Object getServiceFacade(Class aClass)
    throws SystemException {
    // empty method that is automatically
    // implemented by the client aspect
    return null;
  } [...]
```

As is evident from the discussion, the Client does not need to (i) use a Business Delegate, (ii) provide an implementation of the `getServiceFacade` method. The `LocatorAspect` introduces into

the client the implementation for the `getServiceFacade` method which is used to lookup the Service Facade. This facade directly implements the business component's interface and appears to be co-located with the client. This technique offers two main advantages. Firstly, it simplifies the overall design by removing the delegate in most cases (usually, delegates have the same interfaces as the facade they delegate to – note that the use of a specific delegate is still possible). It makes the code more local because as seen in the implementation of the `LocatorAspect`, all the delegating code is confined to a unique aspect. Secondly, the code has no distributed semantics.

The final code is:

- less technology dependent – it can use the EJB component model or any other distributed computing technology or none at all.
- independent of deployment semantics– in case the client is finally deployed in the same virtual machine as the server, then the `getServiceFacade` implementation can be easily changed to return a direct reference to a local object and not the delegate.

The following sample code shows the main parts of the Locator aspect.

```
public aspect LocatorAspect {
  public static final String CURRENCY_SERVICE =
  "edu.rh.cs.j2ee.business.Currency";
  private EJBServiceLocator ejbLocator;
  private JDBCServiceLocator
    jdbcConnectionLocator;
  private JMSServiceLocator jmsObjectLocator;

  // pointcut to capture calls made to
  // getServiceFacade.
  pointcut ejbservice(Class aClass):
    call(* *.getServiceFacade (Class))
    && args(aClass);

  // same principle for databases
  pointcut connectionservice(
    String aDataSource):
    call(* *.getDatabaseConnection(String))
    && args(aDataSource);

  // same principle for JMS
  pointcut jmsservice(String aJMSObject):
    call(* *.getJMSObject(String))
    && args(aJMSObject);

  // EJB service locator -> EJBHome
  Object around(Class aClass)
    throws SystemException:
    ejbservice(aClass) {
      Object service =null;
      try {
        if(ejbLocator == null)
          ejbLocator = new EJBServiceLocator();
        Object home =
          ejbLocator.lookup(aClass);
        // all the lookups can be centralized
        // right here...
        if(aClass.getName()
            .equals(CURRENCY_SERVICE)) {
          CurrencyHome currencyhome =
            CurrencyHome)home;
            service = currencyhome.create();
        }
      } catch (NamingException ne) {
          throw new SystemException(
            ne.getMessage());
      } catch (ClassNotFoundException cne) {
          throw new SystemException(
            cne.getMessage());
      } catch(CreateException ce) {
          throw new SystemException(
            ce.getMessage());
      } catch (RemoteException re) {
          throw new SystemException(
            re.getMessage());
      } catch (Exception e) {
          throw new SystemException(
            e.getMessage());
      }
      return service;
    }

    // -> java.sql.Connection
    Object around(String aDataSource)
    throws SystemException:
    connectionservice(aDataSource) {
      [...]
    }

    // -> JMS Object
    Object around(String aName)
    throws SystemException:jmsservice(aName) {
      [...]
    }

    public pointcut exception():
      call(* edu.rh.cs.j2ee.business..*+.*(..)
           throws *Exception)
      && !within(LocatorAspect);

    // soften thrown exceptions
    declare soft:RemoteException: exception();
    declare soft:TooManyItemsException:
      exception();
    declare soft:TooLargeValueException:
      exception();

    Object around():exception()   {
      Object value = null;
      try {
        value = proceed();
      } catch(Exception e) {
        throw new RuntimeException(
          e.getMessage());
      }
      return value;
    }
}
```

The `LocatorAspect` removes the Client's need to reference the Service Locators explicitly to lookup objects and locate services. This task is seamlessly done within the advices for the pointcuts `ejbservice`, `connectionservice`, and `jmsservice`. The `Locator Aspect` also introduces into the client, the reference to the Service Locator and the references to the classes and interfaces within the `java.rmi` and `javax.naming` packages. The remote exceptions are captured and logged by the `LocatorAspect` (see the `exception()` pointcut). If the application requirement is such that a certain exception is to be handled consistently for all incoming client transactions, then the exception handling can be implemented within the `LocatorAspect` itself and a user friendly error message can be encapsulated within a generic runtime exception (`CompositeRuntimeException`) and passed onto the client. If the application requirement is such that a certain exception is to be handled differently depending on the type of incoming client transaction, then the `LocatorAspect` can pass the exception onto the client, by wrapping it within the `CompositeRuntimeException`. The client layer can then choose to handle the exception appropriately. Finally, the `LocatorAspect` also introduces into the client, the reference to the classes and interfaces within the `javax.ejb` package.

The actual remote invocation performed previously by the delegate is performed within the `ClientAspect`. The code below shows the parts of the `ClientAspect` responsible for the invocation of the `dollarToPound` method of the remote service. Note that it uses `getServiceFacade`, which is implemented by the `LocatorAspect`.

```
public aspect ClientAspect {
  [...]
  // should write a more general pointcut
  pointcut currencyConversion(double aValue):
  call(* edu.rh.cs.j2ee.business.ICurrency+
       .*(*))
  && args(aValue)
  && !within(ClientDataTransferAspect);
```

```
  [...]
  double around(double aValue)
  throws java.rmi.RemoteException:
  currencyConversion(aValue) {

    Signature sig =
      thisJoinPointStaticPart.getSignature();
    String name = sig.getName();
    ICurrency currency = null;
    if(name.equals("dollarToPound")) {
      try {
        currency = (ICurrency)
          getServiceFacade(Currency.class);
      } catch(SystemException se) {}
      return currency.dollarToPound(aValue);
    } else if [...] // other methods
  }
  [...]
```

The Client implementation is conscious only of the Business interfaces and knows nothing about any of the classes or interfaces within the `javax.ejb` package. The client layer is EJB technology agnostic. So if an existing EJB based J2EE solution is implemented using `the LocatorAspect` and `ClientAspect` and, if later there arises a need to convert the existing EJB based implementation to a non EJB solution, then the conversion can be accomplished effortlessly by simply not weaving the aspects into the client, during the compilation phase.

The Service Locator's implementation as a Singleton is based on the techniques outlined in [2]. The Service Locator can be instantiated like a POJO using the new constructor instead of using a factory method like getInstance. However this feature can lead to some confusion among J2EE developers. A factory method makes it clear that the Service Locator is a singleton but the new constructor does not. So it is conceivable that the Singleton might extend another class that can be cloned and developers might call the clone method of the Singleton. In order to prevent the cloning of the Singleton, the `SingleProtocol` aspect's Singleton interface has been modified as shown below.

```
public Object SingletonProtocol.Singleton.clone()
throws CloneNotSupportedException {
   throw new CloneNotSupportedException();
}
```

So if an attempt is made to clone the Singleton, a `CloneNotSupportedException` is thrown.

4.2 Transfer Object

The implementation of the `TestClient` in the typical J2EE application references the transfer object `CurrencyTO`. The Transfer Object reduces the network traffic by carrying multiple data items. The Aspect-Oriented version of the `TestClient` does not use the Transfer Object. As shown in the implementation below, the `Client Aspect` captures the join points of a logical set of remote calls made by the client to the business service within a pointcut. The advice to this pointcut allows the remote invocation during the first call and fetches all the data for the remainder of the invocations within a transfer object. The `ClientAspect` caches the transfer object locally to service subsequent client requests. Thus it eliminates the crosscutting within the client due to the Transfer Object pattern.

```
public aspect ClientAspect {
  public static final String CURRENCY =
    "CurrencyTO";
  private HashMap transferObjectMap =
    new HashMap();

  [...]
  pointcut currencytransfer():
    call(* edu.rh.cs.j2ee.business.ICurrency+
         .get*Currency())
    && !within(ClientAspect);

  Object around()
  throws java.rmi.RemoteException:
  currencytransfer() {

    Signature sig =
      thisJoinPointStaticPart.getSignature();
    String name = sig.getName();
    CurrencyTO to =
      (CurrencyTO)transferObjectMap
       .get(CURRENCY);

    // if the cached TO is null, fetch it
    if(to == null) {
      to = new CurrencyTO();
      ICurrency currency = null;
      try {
        currency = (ICurrency)
        getServiceFacade(Currency.class);
      } catch(SystemException se) {}
      CurrencyTO fetched =
        currency.getCurrencyByCountry()
      to.setUsCurrency(
        fetched.getUsCurrency());
      to.setUkCurrency(
        fetched.getUkCurrency());
      to.setFranceCurrency(
        fetched.getFranceCurrency());
      to.setPolandCurrency(
        fetched.getPolandCurrency());
      transferObjectMap.put(CURRENCY,to);
    }

    // get the data from the cache
    if(name.equals("getUsCurrency"))
      return to.getUsCurrency();
    else if(name.equals("getUkCurrency"))
      return to.getUkCurrency();
    else if(name.equals("getFranceCurrency"))
      return to.getFranceCurrency();
    else if(name.equals("getPolandCurrency"))
      return to.getPolandCurrency();
    return null;
  }
  [...]
```

The `ClientAspect` also provides a cache invalidation pointcut. For our simple case, it invalidates the cache (removes the transfer object from the hash map) when the program returns from the main method. The cache invalidation pointcut is application dependent and can be quite complex in real applications.

4.3 Session Facade

The Aspect version of the pattern uses the `SessionBeanProtocol` and the `FacadeAspect`, to introduce the `javax.ejb.SessionBean` interface within the session facade. The `CurrencyBean` session façade is a POJO that is business -functionality centric and is oblivious to the `javax.ejb package`. The facade is reusable and adaptable within a non EJB environment.

```
public interface SessionBeanProtocol
    extends javax.ejb.SessionBean {}

public aspect FacadeAspect {
  // ICurrency is due to the BusinessInterface
  //pattern and not the
  //Remote interface
  declare parents: CurrencyBean implements
     SessionBeanProtocol,ICurrency;
  public void SessionBeanProtocol.ejbCreate()
    throws CreateException {}
  public void SessionBeanProtocol.ejbRemove() {}
  public void SessionBeanProtocol.ejbActivate() {}
  public void SessionBeanProtocol.ejbPassivate() {}
  public void SessionBeanProtocol
        .setSessionContext(SessionContext sc) {}
}
```

4.4 Code Improvement Evaluation

4.4.1 Business Delegate, Service Locator, and Transfer Object patterns

The Aspect-Oriented implementation of the Business Delegate, the Service Locator and the Transfer Object patterns has the following closely related modularity properties:

Locality – All the code that implements the Business Delegate functionality and the associate service lookup is in the `ClientAspect` and the `LocatorAspect` and none of it is in the participating client classes. For each kind of service lookup, the code is within the advice of the `LocatorAspect`. The packaging of all related data for a transfer object is localized within the `ClientAspect`. The participant clients are entirely free of the Business Delegate and Transfer Object pattern contexts and as a consequence there is no coupling between the participants. Potential changes to the pattern instance are confined to one place. All the Singleton related code is within the `SingletonProtocol` and the ServiceLocator is a POJO.

Reusability – The core pattern code is abstracted and reusable. The implementation of the `getServiceFacade` method within the `Client` via the `LocatorAspect` generalizes the overall pattern behavior. The interface can be reused and shared across multiple pattern instances. The implementation of the Transfer Object is limited to the clients that need the same Transfer Object. The `SingletonProtocol` aspect can be reused to create several types of Singeltons.

Composition transparency – Since the Client implementation is not coupled to either of the patterns, it can participate in other kinds of pattern relationships and the resulting code does not become more complicated. Since the ServiceLocator is oblivious to the Singleton pattern's context, it could participate in another pattern context seamlessly.

(Un)pluggability – Since the Client need not be aware of its role in any of these pattern instances, it is possible to switch effortlessly between using the Business Delegate pattern and Transfer Object pattern, and not using them in the system. It is possible to add and remove the Singleton property to the ServiceLocator easily.

4.4.2 Session Facade pattern

The Aspect-Oriented implementation of the Session Facade pattern has the following closely related modularity properties:

Locality – All the code that implements the Session Facade functionality is within a POJO and the Session Bean contract is introduced via a protocol and an aspect. For each kind of Session Facade, we only need to extend the `SessionBeanProtocol` and supply an implementation for the Session Bean's methods via the aspect. The participating facade is entirely free of the pattern context, and as a consequence is EJB agnostic. Potential changes with the EJB container's contract are confined to the aspect.

Reusability – The core pattern code present within the Session Bean interface methods (`ejbCreate()`, `ejbActivate()`…) is abstracted and reusable.

Composition transparency – Since the facade implementation is not coupled to the pattern, it can participate in other kinds of pattern relationships and the resulting code does not become complicated.

(Un)pluggability – Since the facade need not be aware of its role in this pattern instance, it is possible to switch effortlessly between using the EJB Component Model and not using it in the J2EE application.

4.4.3 Transfer Object Assembler, Value List Handler, and Composite Entity

The modularity advantages discussed for a Session Facade are also applicable to the Transfer Object Assembler and the Value List Handler.

The `CompositeEntityBean` is implemented as a POJO and the `javax.ejb.EntityBean` interface is weaved into the Composite Entity via the `EntityAspect` and the `EntityBeanProtocol`. The Composite Entity implementation purely manages the inter-entity relationships and is unaware of the `javax.ejb` package. The POJO entity is reusable and adaptable within a non EJB environment.

4.4.4 Application Services and Business Objects

The Application Services and the Business Objects are usually implemented using any of the GoF patterns, and as discussed in [2], there may or may not be significant modularity benefits after applying aspects, depending on their implementation.

5. ANALYSIS AND CONCLUSIONS

In this paper we have presented and compared a J2EE application built using EJB Component Software Engineering techniques and using Aspect-Orientation. We have demonstrated and explained how a more flexible, adaptable and reusable component based J2EE system can be built using Aspect-Oriented techniques.

The improvements from using AspectJ in J2EE business tier pattern implementations are closely tied to the presence of crosscutting in the structure of the patterns. Crosscutting in pattern structure is caused by roles [2] and their collaboration with participant classes. We notice great improvements in those patterns where a single module of abstraction handles the original behavior and the pattern specific behavior. In such patterns, the roles cut across participant classes and conceptual operations crosscut methods and constructors. Patterns having shared participants can also crosscut each other. The improvements in the J2EE world are apparent as a set of properties associated leading with modularity. The J2EE pattern implementations are more localized and reusable and hence the system is more adaptable. Localization enhances the documentation. AspectJ implementations of J2EE business tier patterns are composable because there is a better alignment between the dependencies in the code with dependencies in the participant structure.

Our results suggest that Aspect-Orientation should strongly be considered in the design and implementation of J2EE applications.

REFERENCES

[1] Deepak Alur (Author), John Crupi (Author), Dan Malks. *Core J2EE Patterns: Best Practices and Design Strategies*, Prentice Hall PTR; 2nd edition (June 2003)

[2] Hannemann and Kiczales. *Design Pattern Implementation in Java and AspectJ*, Proceedings of the 17th Annual ACM conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pages 161-173, November 2002.

[3] Florijn, G., Meijers, M., Winsen, P. van. *Tool support for object-oriented patterns*. Proceedings of ECOOP 1997

[4] Soukup, J. *Implementing Patterns*. In: Coplien J. O., Schmidt, D. C. (eds.) Pattern Languages of Program Design. Addison Wesley 1995, pp. 395-412