Crosscutting Concerns in J2EE Applications

Ali Mesbah, Arie van Deursen*
Centrum voor Wiskunde en Informatica
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
{Ali.Mesbah,Arie.van.Deursen}@cwi.nl

Abstract

We explore the evolution benefits of adopting aspects in a J2EE setting by studying crosscutting concerns in a typical J2EE application. To identify these concerns, we take a top-down *as well as a* bottom-up *approach. In the top-down view we focus on typical concerns that are known to be crosscutting (e.g., persistence), the way they are currently implemented and the possible gains and benefits if solved by aspects. In the bottom-up approach we take a look at the application's source code, and apply* aspect mining *techniques in order to find crosscutting concerns. We discuss how such concerns can be represented in an aspect-oriented language (viz. AspectJ), and reflect on the results in terms of maintainability and evolvability of the affected system.*

1. Introduction

From its inception, Java 2, Enterprise Edition (J2EE)[1] has established a new model for developing distributed applications. It is based on well-defined components to provide server-side and client-side support for developing multi-tier applications. The J2EE architecture defines a client tier, a middle tier and a back-end tier. The client tier provides support for a variety of client types (e.g. HTML pages generated by JavaServer Pages (JSP), Java applets, Java Web Start-enabled clients). The middle tier supports client services through Web containers (e.g. Servlets, JSP) and supports business logic component services through Enterprise JavaBeans (EJB) containers. The middle tier is often subdivided into the Web tier and EJB tier. On the back-end tier , the enterprise information systems are accessible by the way of standard APIs (e.g. JDBC).

Aspect-oriented software development (AOSD) [9] aims at improving the modularity of software systems, by capturing inherently scattered functionality, often called *crosscutting concerns*, in a well-modularized way, making the evolution of such systems easier and manageable. In order to achieve this, aspect-oriented programming languages add an extra abstraction mechanism, called an *aspect*, on top of existing modularization mechanisms such as functions, classes and methods. Aspects allow developers to tackle the problems of scattering and tangling by reducing the spread of code belonging to a certain concern over different components.

J2EE already provides a transparent component-based framework in which many crosscutting concerns such as security and authentication are dealt with in the container without requiring the developer to know the corresponding implementation. However, introducing AOSD into J2EE application development could provide us with more flexible, modularized applications thus empowering the long-term evolution of applications developed in the framework.

The goal of this paper is to explore the evolution benefits of adopting aspects in a J2EE setting. To that end, we study crosscutting concerns in a typical J2EE application, Sun's Pet Store demonstration. To identify these concerns, we take a *top-down* as well as a *bottom-up* approach. In the top-down view we focus on typical concerns that are known to be crosscutting (e.g., persistence), the way they are currently implemented and the possible gains and benefits if solved by aspects. In the bottom-up approach we will take a look at the application's source code, and apply *aspect mining* techniques in order to find crosscutting concerns. We discuss how such concerns can be represented in an aspect-oriented language (viz. AspectJ), and reflect on the results in terms of maintainability and evolvability of the affected system.

This paper is laid out as follows. Section 2 discusses related work. In section 3 a short overview of the Pet Store web application is outlined which is the case study used in this paper. Section 4 discusses the top-down concerns and the ways they could be implemented by aspects. The bottom-up approach is presented in section 5. In section 6 we discuss the achieved results. Finally, Section 7 presents our conclusions and future work.

We assume the reader has basic knowledge of both J2EE and AspectJ.

*Also affiliated at Delft University of Technology, Faculty of Electrical Engineering, Mathematics and Computer Science, Mekelweg 4, 2628 CD Delft, The Netherlands.

[1] http://java.sun.com/j2ee/

2. Related Work

There are a number of publications reporting the possible applications of aspects to J2EE applications.

Soares et al. [18] discuss their experiences in applying the paradigm to a web-based information system's distribution, persistence and transaction concerns. Kerston et al. [8] share their experiences with a web-based educational system, separating network context concerns in a distributed environment. In [16], a component-based web-crawling system was developed with and without aspects to be able to evaluate the differences of both approaches. The results favor the aspect-oriented paradigm. Zhang and Jacobsen [20] apply aspect-oriented programming to middleware (Object Request Broker) demonstrating a reduction of the complexity of the architecture. Kim and Clark [10] present a case study in which they show that using the EJB framework modularizes and minimizes crosscutting concerns such as security, persistence and transaction management. On the other hand, Choi [2] discusses the deficiencies of the EJB's architecture and tries to present an aspect-based server implementation. A comparison of J2EE container managed and aspect-oriented security is presented in [17]. Fabry [4] discusses the shortcomings of declarative transaction management in EJB with respect to a clean separation of concerns and proposes an aspect solution which can detect the need for transactional methods and integrate all transactional handling in one unit.

Han and Hofmeister [6] concentrate on navigation concerns of J2EE web applications and present an aspect-oriented approach to separate these concerns. An analysis of the crosscutting nature of the J2EE patterns is illustrated by [15]. Finally, Cohen and Gil [3] propose a promising new language, AspectJ2EE, geared towards the implementation of J2EE application servers and applications. They use a deploy-time mechanism to bind services to user applications in the application server. This would enable the EJB developers to extend, enhance and replace the standard services provided by the container.

None of these works however, cover how aspects influence evolution in J2EE environments, a gap we try to bridge in the present paper. Our work differs from these approaches in that: first, we adopt two different aspect mining techniques to identify candidate aspects; second, we provide full details of the resulting aspect-oriented code in order to provide a clear picture of the advantages and disadvantages; and third, we explicitly evaluate the evolution benefits in terms of change scenarios.

3. Pet Store

Pet Store[2] is an example of a J2EE web application developed by Sun in order to illustrate all sorts of J2EE features. The main function of the application is to allow customers to search and purchase pets through a web browser. The application provides the customers with an interface (Web site) to search through catalogs of products and order items. The Admin client has an interface to view sales statistics and manually accept or reject orders.

There are a number of reasons why we have chosen Pet Store as our case study. First of all, Pet Store is a well-designed web application using the latest J2EE specifications. There are numerous design patterns used in its architecture such as MVC, ServiceLocator, Data Access Object, etc. It is a well-known open-source application. Further, even though the functionality of the system seems simple, the actual implementation is quite complex. The complexity is mainly due to the fact that it is a demo presenting the best and newest practices of J2EE application development. The complexity is also a price to pay for maintainability, portability and scalability of the application. Finally, keeping all this in mind, our reasoning is based on the assumption that if we are able to identify crosscutting concerns and show benefits of aspectizing those concerns in such a well-designed system, then it should also be possible to find crosscutting concerns in real life J2EE applications, many of which are in practice not half as well-designed.

4. Top-down concerns

In this section we will consider Transaction management which is known to be crosscutting from a top-down perspective. The Persistence and Security concerns in Pet Store are thoroughly disscussed in our technical report [14], addressing how how each of them is implemented in J2EE environments and whether the implementation can be improved if aspects are applied. Other relevant top-down concerns such as Tier-cutting concerns (e.g. compression, encryption [3]), Logging and Exception Handling, Persistence and Security are left out of this paper because of space limitations. The crosscutting nature of some of the J2EE patterns [15] can also be examined from a top-down perspective.

4.1. Transaction management

In J2EE applications, transaction management is handled in either a *declarative* or *programmatic* form. EJB provides an elegant system for handling transaction management by enabling the developers to declare the transactional functionality in the EJB's deployment descriptor. It is then the container that has to take care of the actual implementation. This approach separates the transaction management from the core business logic of the application. As Fabry [4] discusses, one could manage to produce tangling and scattering code even when using the declarative possibilities of EJB if the functionality provided by the container is not satisfactory.

[2] http://java.sun.com/blueprints/, Java Pet Store Demo 1.3.2

```
private void insertTemplate(HttpServletRequest request,
 HttpServletResponse response, String templateName)
 throws IOException, ServletException {
  try {
    UserTransaction ut = null;
    InitialContext ic = new InitialContext();
    UserTransaction ut = (UserTransaction)
      ic.lookup("java:comp/UserTransaction");
    ut.begin();
    context.getRequestDispatcher(templateName)
     .forward(request, response);
    ut.commit();}
  ...
}
```

Figure 1. Method using Transaction management

Transaction management is also used in other parts of the J2EE platform (e.g. Servlets) where no EJB is present. J2EE does not provide any support for non-EJB parts regarding the declarative transactional management.

We analyzed the Pet Store application code looking for possible crosscutting transactional concerns. The application makes exclusive use of the declarative transaction control in the EJB tier without producing any tangling code. The EJB deployment descriptor defines any call to methods that need transactional support by setting the transaction attribute to `Required`.

In the Web tier a programmatic approach is used in two classes to manage transaction. The `TemplateServlet` begins a `UserTransaction` before it forwards a request to the template JSP and ends the transaction after the forward has accomplished. See Figure 1. The `doPost` method of the `RcvrRequestProcessor` Servlet uses the same transactional code to update supplier inventories. In order to be able to aspectize the transactional code, the `doPost` method had first to be refactored because it was used in a nested `if` statement. We moved the transactional handling into a new method called `updateAndSend`. This refactored method used the transactional code in a similar way as in Figure 1.

We decided to aspectize the transactional management in the following way. First we defined an *abstract* aspect [11], which has an `around` advice while keeping the `pointcut` abstract . See Figure 2. Then we created a concrete transaction aspect which inherits from the abstract aspect, providing concrete definitions for the abstract `pointcut`. See Figure 3.

With this aspectized approach, the transactional control code is modularized and is reusable. Further, the classes using the functionality are now oblivious of the transactional code.

5. Bottom-up concerns

Aspect mining is the quest for identifying candidate aspects in existing object-oriented systems and isolating them into aspects, improving the comprehensibility of the system, and thereby improving it's maintainability and evolvability.

```
public abstract aspect AbstractTransAspect {
  abstract pointcut transactionOperations();
  void around() : transactionOperations() {
    try {
      InitialContext ic = new InitialContext();
      UserTransaction ut = (UserTransaction)
        ic.lookup("java:comp/UserTransaction");
      ut.begin();
      proceed();
      ut.commit();}
  ...
}
```

Figure 2. Abstract Transaction Aspect

```
public aspect PetStoreTransactionAspect
 extends AbstractTransAspect {
  pointcut transactionOperations() :
    execution(* com..TemplateServlet.insertTemplate(..)
     throws IOException, ServletException) ||
    execution(*
      com..RcvrRequestProcessor.updateAndSend(..)
     throws IOException, ServletException);
}
```

Figure 3. Pet Store Transaction Aspect

Aspect mining techniques can take a bottom-up approach in which they try to examine the current implementation of a software system, looking for patterns of tangling and scattering code which would make good aspect candidates. This bottom-up approach could also be applied to J2EE applications to find and extract crosscutting concerns.

In this section we will look at two aspect mining techniques and see how each technique could be used in J2EE environments discussing the possible benefits of applying each approach.

5.1. Fan-In Analysis

Fan-in analysis [13] takes a bottom-up approach, analyzing the code for certain crosscutting symptoms. This technique defines the number of distinct method calls to a given method as the fan-in number. Methods having a high fan-in value are likely to be crosscutting methods and should be further examined to see whether they could be considered as candidate aspects.

The Pet Store application code was analyzed by [13], identifying a number of candidate aspects for methods with a fan-in of 8 and above.

Next we will present an aspect-oriented implementation of the identified candidate aspects in order to discuss the effects the refactoring would have on the maintainability and evolvability of the application.

Exception Wrapping Exception wrapping is a crosscutting concern that affects a number of classes in

```
public EJBLocalHome getLocalHome(String jndiHomeName)
    throws ServiceLocatorException {
  EJBLocalHome home = null;
  try {
    if (cache.containsKey(jndiHomeName)) {
      home = (EJBLocalHome) cache.get(jndiHomeName);}
    else {
      home = (EJBLocalHome) ic.lookup(jndiHomeName);
      cache.put(jndiHomeName, home);}
  } catch (NamingException ne) {
    throw new ServiceLocatorException(ne);}

  return home;
}
```

Figure 4. Exception wrapping in Pet Store

```
public aspect ExceptionWrappingAspect {
  declare soft : NamingException : call(* *.*(..)
    throws NamingException)
    || call(*.new(..) throws NamingException)
    && within(*.ServiceLocator);

  after() throwing(SoftException ex) throws
   ServiceLocatorException
    : execution(* *.*(..) throws
        ServiceLocatorException)
     && within(*.ServiceLocator) {
      throw new ServiceLocatorException(
        ex.getWrappedThrowable());}
}
```

Figure 5. Exception wrapping in AspectJ

Pet Store. The `ServiceLocator` classes for instance catch the `NamingException` and re-throw `ServiceLocatorException`. This is a typical instance of the Business Delegate J2EE pattern [1]. Every method in these classes catches exceptions thrown by the underlying implementation and re-throws an application-specific exception requiring a try/catch block in each method. Normal refactoring of these identical try/catch blocks is not possible in object-oriented languages. This *logic duplication* is a result of a language limitation. Aspects can however be used to refactor this pattern of scattering code [12, 11].

To refactor the exception wrapping logic, we implement an aspect in which the checked-exceptions are declared soft. This way the exception will be wrapped in an unchecked-exception (`SoftException`) when thrown. An after throwing advice is then used to catch any `SoftException` thrown and throw a new `ServiceLocatorException` wrapping the original exception obtained.

Figure 4 shows one of the web `ServiceLocator` methods in its original form. Applying the aspect presented in Figure 5 would allow us to refactor the method (and all the other methods in the class). Figure 6 shows the refactored method.

As can be seen, the refactoring of the exception wrapping concern results in a simple aspect defining the wrapping strategy, and cleaner business logic that is not tangled with wrapping anymore. This not only leads to a reduction in code size of up to 20% in the refactored classes, it also improves evolvability of both the business logic and the wrapping strategy.

```
public EJBLocalHome getLocalHome(String jndiHomeName)
 throws ServiceLocatorException {
  EJBLocalHome home = null;
  if (cache.containsKey(jndiHomeName)) {
    home = (EJBLocalHome) cache.get(jndiHomeName);}
  else {
    home = (EJBLocalHome) ic.lookup(jndiHomeName);
    cache.put(jndiHomeName, home);}
  return home;
}
```

Figure 6. Aspectized by ExceptionWrappingAspect

ServiceLocator & Singleton A ServiceLocator J2EE pattern [1] acts as a central point for obtaining and caching a service. This pattern implemented in classes `ServiceLocator` was also identified as a candidate aspect in Pet Store. There are two `ServiceLocator` classes in Pet Store one of which (JMS) is implemented as a Singleton and the other one (EJB) is (mistakenly) not.

The ServiceLocator can be approached in two ways. One could argue that because the `getInstance` method has a high fan-in, the methods calling the method should be refactored in a way that no explicit calling is needed. There are techniques which make this approach possible. For instance, Spring's[3] inversion of control allows us to pass the `ServiceLocator` instance reference to our classes in a subtle manner. However, because of the non-systematic nature of the methods, it is not possible to create a unified aspect which would allow us to refactor all the calls to the `getInstance` method.

The second approach, presented by [7, 15], tries to refactor the Singleton nature of the `ServiceLocator` class itself. This means removing the private constructor and the `getInstance` of the class and instead providing a public constructor which is implemented as an `around` advice, initializing the instance on the first call and returning it on all constructor calls.

The Singleton class was refactored using the aspect shown in Figure 7. As can be seen, the aspect captures all calls to the constructor using the `around` advice. The advice then creates the static instance if not already created and initializes the object; otherwise the instance is returned without manipulation.

Now the question is what the consequences are of this refactoring step. Hiding the Singleton nature of the class could confuse J2EE developers. This could also lead to violations of the singleton nature, if, for example, a subclass provides cloning functionality. [15,] mention a workaround for this possible problem.

On the other hand one could argue that hiding the Singleton nature of the class makes the application oblivious in the sense that the clients using the `ServiceLocator` class do not and should not care about the implementation details of the service.

[3] http://www.springframework.org/

```
privileged public aspect LocatorAspect {
  private static ServiceLocator service;
  pointcut serviceLocator():
    call(*.ServiceLocator.new())
    && !within(LocatorAspect);

  Object around()
   throws ServiceLocatorException: serviceLocator() {
    synchronized (service) {
      if (service == null) {
        service = new ServiceLocator();
        try {
          service.ic = new InitialContext();
          service.cache =
            Collections.synchronizedMap(new HashMap());
        } catch (NamingException e) {
          service = null;
          throw new ServiceLocatorException(e);
    } } }
    return service;
} }
```

Figure 7. Locator aspect

Precondition Checking Precondition checking often requires duplicated code if the conditions are common to many methods. In Pet Store many EJB classes implementing `EntityBean` use a Plain Old Java Object (POJO) of their own to hold the actual data. All these classes have a static `fromDOM` method which expects a DOM node as input parameter and returns an instance of the corresponding class made from the node. The node has to have a certain structure for the method to be able to parse it to the right class. The main precondition states that the name of the first element of the DOM has to coincide with the value of a static variable defined in the class. The `Address` class, for instance, has a static variable called `XML_ADDRESS` which has a string value of "Address". This means the first element of the DOM has to be called "Address", otherwise the check will through an `XMLDocumentException`.

The parameter checks occur at the beginning of the methods and are similar to:

```
public static $OBJECT$ fromDOM(Node node)
 throws XMLDocumentException {
  if (node.getNodeType() == Node.ELEMENT_NODE
    && ((Element) node).getTagName().equals($PARAMETER$)) {
    //OK proceed
    ...
  }
  else {
    throw new XMLDocumentException($PARAMETER$
      + " element expected.");
  }}
```

where `$OBJECT$` is the corresponding class type (e.g. Address) and `$PARAMETER$` is the name of the static variable holding the expected name of the first element.

Conventional implementations of this precondition require adding identical code conditional checks into many methods. With aspect-oriented techniques, we can refactor such contract checks into a separate aspect. Figure 8 shows the aspect which handles the precondition checks. We define the pointcut as calls to the `fromDOM` methods and use a `before` advice to check the precondition. The

```
public aspect XMLPreCheck {
  pointcut fromDOM(Node node) :
    call(* com.sun.j2ee.blueprints.*.ejb.*.fromDOM(Node)
      throws XMLDocumentException) && args(node);

  before(Node node) throws XMLDocumentException :
      fromDOM(node) {
    if (!isPreChecked(node, thisJoinPointStaticPart)) {
      throw new XMLDocumentException(
        getParameterValue(thisJoinPointStaticPart)
        + " element expected.");}}

  private boolean isPreChecked(Node node,
      JoinPoint.StaticPart joinPoint) {
    return (node.getNodeType() == Node.ELEMENT_NODE
        && ((Element) node).getTagName().equals(
          getParameterValue(joinPoint)));}

  private String getParameterValue(
      JoinPoint.StaticPart joinPoint) {
    String className =
   joinPoint.getSignature().getDeclaringType().getName();

    return className.substring(
     (className.lastIndexOf('.') + 1);}
}
```

Figure 8. Precondition Check Aspect

tricky part in this advice is to find the value of the expected variable on which the check has to be carried out. The `thisJoinPointStaticPart` gives us the corresponding class information and using reflection on the class, we can find the required parameter. By analyzing the code, a pattern can be seen in the value of the required variable; i.e., the value of the static variable is the same as the class-name. This means we can use the class-name to check our precondition and that is exactly what we have done. This, however, implicitly defines a contract by which all future classes of the same type have to abide.

Using this precondition aspect we are able to refactor out the precondition checks from nine classes in Pet Store.

5.2. Interface Concerns

Interfaces and their implementations can be crosscutting. When the implementation of an interface is distributed across many classes in a system, aspect-oriented programming enables us to extract and locate it in an aspect, increasing the modularity of the system. In order to investigate the behavior of this crosscutting concern in J2EE applications, we analyzed the Pet Store application, looking for possible aspect candidates. The results are discussed here.

Extracting Interface Implementations The first concern that was identified as a possible aspect candidate was found because four interfaces (`Event`, `EJBAction`, `HTMLAction` and `EventResponse`) had a default implementation (`EventSupport`, `EJBActionSupport`, `HTMLActionSupport` and `EventResponseSupport`). Other classes then extended these *support* classes instead of implementing the interfaces. While Pet Store avoids duplicated code using this technique, this approach fails for the classes

```
public interface Event {
  public void setEJBActionClassName(
    String ejbActionClassName);
  public String getEJBActionClassName();
  ...
  static abstract aspect EventAspect {
    private String Event.ejbActionClassName = null;
    public String Event.getEJBActionClassName() {
           return ejbActionClassName;}
    public void Event.setEJBActionClassName(
      String _ejbActionClassName) {
           ejbActionClassName = _ejbActionClassName;}
    ...
} }
```

Figure 9. Aspectized Event interface

that are already extending another class or for classes that need to extend another class.

Using the refactoring technique suggested by [11], we are able to make use of the inter-type declaration mechanism to write aspects which introduce the default implementation into the identified interfaces.

Figure 9 shows the aspectized `Event` interface. The inner aspect functions as an implementation of the interface. This way the `EventSupport` class becomes obsolete. The main advantage of this refactoring is that all the classes which extended the `EventSupport` class can now implement the `Event` interface without the need to implement the declared methods in the interface (if they are satisfied with the default implementation of course) and more importantly, the classes can now extend another class if needed.

Interface Migrating Interface migration tries to mine interface implementations which can be regarded as crosscutting concerns and refactor them to aspects. Tonnela et al. [19] represent a mining technique based on external package identification, string matching for interfaces names, clustering and unpluggability of the methods.

Further analysis of the application code, while keeping these indicators in mind, reveals that there are many classes/interfaces which implement/extend the `Serializable` interface. This is a common character of J2EE applications because of their distributed nature. From a logical point of view, this character does not really belong to the principal decomposition of the application [19] and therefore it could be treated as an aspect. An aspect capturing the serializable concern is the following:

```
declare parents : ProfileInfo implements Serializable;
declare parents : HTMLAction extends Serializable;
...
```

where `ProfileInfo` is a class and `HTMLAction` is an interface. In Pet Store the purpose of using `Serializable` is simply to identify classes whose objects are serializable. This means the private `writeObject` and `readObject` methods are not customized and no field is marked as `transient`. Private methods cannot be introduced into target classes in AspectJ which means we would not be able to fully aspectize the serialization if Pet Store did customize the mentioned methods or used `transient` fields.

A total of 29 classes and 6 interfaces were declared to implement/extend the `Serializable` interface within this aspect. The explicit `implements` and `extends` declarations in the classes and interfaces were removed afterwards.

Further, EJB rules demand the extension of `EJBLocalObject` for the component interface and `EJBLocalHome` for the home interface. Pet Store has 13 component interfaces with a name ending in *Local*. There are also 13 home interfaces with a name ending in *LocalHome*. We have aspectized these two interface concerns as follows:

```
declare parents : com.sun.j2ee.blueprints.*.ejb.*.*Local
  implements javax.ejb.EJBLocalObject;

declare parents : com.sun.j2ee.blueprints.*.ejb.*.*LocalHome
  implements javax.ejb.EJBLocalHome
```

In this aspectized version, the two local EJB interfaces are modularized in a single aspect. The serialization concern is also handled in a single separate unit, resulting in an increased degree of modularization. This enables developers to get a better overall view of the classes implementing these interfaces.

6. Discussion

This section reflects on the results we obtained by studying the crosscutting nature of the top-down and bottom-up concerns. Table 1 shows an overview of the studied concerns in Pet Store. Each concern is examined against a set of properties:

Crosscutting Has the concern a crosscutting nature in the application? As it can be seen all the concerns are crosscutting in some way.

Aspectizable From the crosscutting concerns, which ones are aspectizable? A concern is aspectizable when its crosscutting nature can be resolved by applying aspect-oriented refactoring. It is interesting to see that all the crosscutting concerns are aspectizable which shows that the aspect-oriented paradigm is suited for tackling crosscutting problems.

Code Reduction Logic duplication is one of the manifestations of crosscutting concerns and aspect-oriented techniques are experts in minimizing that. Aspectizing the *Exception Wrapping* concern resulted in a 20% code reduction in the affected classes. Aspectizing the *Precondition Checking* concern allowed us to completely remove the check code from 9 classes, each class having one affected method. It is worth noting that only three of the aspectized concerns showed a reduction in code size. This implies that aspect-oriented programming has a much broader purpose than only code reduction.

Obliviousness Obliviousness states that neither the existence nor the execution of the aspect code is apparent by examining the body of the base code. Obliviousness is desirable because it allows greater separation of concerns in the application development process [5]. Five of the aspectized concerns resulted in obliviousness.

Reliability In certain cases, the use of an aspect makes it harder to make a particular kind of mistake. For example, the generic pointcut for the *Exception Wrapping* aspect ensures that new methods automatically have the naming exception wrapped into a ServiceLocator exception. For the same reason, faults are less likely for the *Transaction* and *Precondition Checking* concerns.

Modularity Aspectized modularity enables us to reduce *tangling*, multiple concerns intermixed, and *scattering*, spread of code for a concern, which in turn simplifies maintenance and evolution. Modularity obtained in four of the concerns in this case study has resulted in code that is more localized, less coupled, and has better cohesion. *Transaction*, for instance, was aspectized into a new module, enabling the removal of corresponding code from classes that were merely using *Transaction*, i.e., transaction management was not their primary function. *Interface Migration* is another example of achieving better modularity. The serialization concern was moved to an aspect, enabling us to remove the scattering code in all classes that need to be serializable. It has made the interface implementation transparent to the classes that were using it explicitly before the aspectization process. This allows us to add the serialization concern to new classes easily and also gives us a clear overview of all the classes using this particular concern, increasing design and code understandability.

Evolvability Software evolution is a process that either introduces new requirements into an existing system, or modifies the system if the requirements change or were not correctly implemented. We consider *evolvability* improvements by presenting possible change scenarios in the life-cycle of the application:

Transaction: Imagine we decide to use a different transaction management API. In the conventional implementation all classes using the transaction code had to be modified. In the new aspectized version, however, the change will take place in only one aspect. Further adding the functionality to a new class is a matter of expanding the `pointcut` in the transactional aspect.

Exception Wrapping: Being able to modularize the Exception Wrapping concern allows us to have a uniform way of dealing with all the affected methods; i.e., there will be no inconsistencies in the type of wrapped exception thrown or the way exceptions are logged. The base code is separated from the aspect code which enables us to alter one independent of the other.

Concern	Crosscutting	Aspectizable	Code Reduction	Obliviousness	Reliability	Modularity	Evolvability
Transaction	✓	✓	✓	✓	✓	✓	✓
Exception Wrapping	✓	✓	✓	✓	✓	✓	✓
ServiceLocator	✓	✓		✓			✓
Precondition Checking	✓	✓	✓	✓	✓	✓	✓
Interface Extraction	✓	✓				✓	✓
Interface Migration	✓	✓		✓		✓	✓

Table 1. Studied concerns in Pet Store

Precondition Checking: Precondition checking is one of those concerns with a volatile requirement that can easily change over time. If the requirement change states that, for instance, the check has to be extended on more parameters, it is now much easier to adapt the implementation.

ServiceLocator: A possible scenario is that we decide to refactor our `ServiceLocator` class not to be a *Singleton* anymore because Singleton makes it very hard to write unit tests, i.e, Singleton makes it very difficult to follow the testing independence rule. Because the application is now oblivious of the Singleton nature, this change will not affect the clients using the `ServiceLocator` class.

Interface Concern: We have a central module defining all the classes that implement a certain interface using aspects. We can also have a default implementation of the corresponding interface through aspects. Imagine we write a new class that needs to implement the interface. We then simply add the name of the new class to the list of the declared classes that implement the interface in the aspect. *Interface Extraction* has also given us the ability to make use of multiple inheritance if needed in the future.

Costs Utilizing aspect-oriented programming is not without costs. While aspects improve modularity, they can increase a system's complexity. For instance, the transaction concern is aspectized using an abstract aspect and a concrete aspect. Generally speaking, this is more complex to comprehend than the original simple implementation. Understandability is also at stake when aspects do not achieve full obliviousness. For example, a pointcut may rely on a particular naming pattern, which the developer, not aware of the aspect code, may break by applying a method renaming.

7. Concluding Remarks

Evaluation The key lessons learned from our experiments are the following. First, many concerns that are by nature crosscutting are well addressed by J2EE's container mechanism. Second, in those cases where the container mechanism is not sufficiently powerful or cannot be utilized, such as for the transaction mechanism adopted, an aspect-oriented implementation of these concerns does offer benefits. Third,

crosscutting concerns in J2EE systems include generic ones as well, such as logging (not discussed), or precondition checking and interface extraction. Fourth, while adopting aspects in J2EE applications does have clear benefits, these are relatively small (as suggested by our example concerns and scenarios) and certainly not of an order of magnitude. Extrapolating some of the numbers we found, we suspect that there are approximately 25 reasonable aspect opportunities in Pet Store, on a total of 283 classes/interfaces. We do expect, however, that in real life J2EE applications the need to circumvent J2EE's container mechanism will be much stronger (for example due to performance limitations), leading to significant benefits from the use of aspect-orientation in such cases.

Contributions This paper has presented the results we have obtained by aspectizing a number of top-down and bottom-up crosscutting concerns in a J2EE case study. We have provided the full details of the resulting aspect-oriented code in order to outline a clear view of the advantages and disadvantages. The results show that even though proper use of design patterns and container-managed services helps encapsulation in many situations, we still find the repetition and concern-mixing phenomena even in well-designed J2EE systems. We have shown how aspect-oriented techniques provide a framework in which these crosscutting concerns can easily be modularized, allowing greater separation of concerns which in turn increases the evolvability of the system.

Future Work Our future work will focus on studying the crosscuttingness of concerns in a number of real world J2EE applications. We will try to apply a wider range of aspect mining techniques in order to identify aspect candidates. Our research will also examine the convenience of using other aspect-oriented tools (besides AspectJ) such as AspectWerkz[4] in J2EE environments. We will also provide access to APetStore, an aspect-oriented refactoring of the Pet Store application, through our Web site, which will be further used to experiment with aspects.

Acknowledgements We would like to thank Marius Marin, Magiel Bruntink and Gertjan van Oosten for their reviews, advice and feedback. This paper received partial support from SenterNovem, project Single Page Computer Interaction (SPCI).

References

[1] D. Alur, J. Crupi, and D. Malks. *Core J2EE Patterns*. Sun Microsystems, Inc., USA, 2003.

[2] Jung Pil Choi. Aspect-Oriented Programming with Enterprise JavaBeans. In *EDOC '00: Proceedings of the 4th International conference on Enterprise Distributed Object Computing*, page 252. IEEE Computer Society, 2000.

[3] Tal Cohen and Joseph (Yossi) Gil. AspectJ2EE = AOP + J2EE: Towards an Aspect Based, Programmable and Extensible Middleware Framework. In *ECOOP - Object-Oriented Programming, LNCS 3086*, pages 219–243. Springer-Verlag, 2004.

[4] J. Fabry. Transaction management in EJBs: Better separation of concerns with AOP. In Y. Coady and D. Lorenz, editors, *Proc. of the 3rd AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*, Victoria, Canada, 2004. University of Victoria.

[5] Robert E. Filman and Daniel P. Friedman. Aspect-Oriented Programming is Quantifi cation and Obliviousness. In *OOPSLA Workshop on Advanced Separation of Concerns*,, 2000.

[6] Minmin Han and Christine Hofmeister. Separation of Navigation Routing Code in J2EE Web Applications. In *ICWE*, pages 221–231, 2005.

[7] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In *Proceedings of the 17th Annual ACM conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 161–173, Boston, MA, 2002. ACM Press.

[8] Mik Kersten and Gail C. Murphy. Atlas: a case study in building a web-based learning environment using aspect-oriented programming. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 340–352. ACM Press, 1999.

[9] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *11th Europeen Conf. Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.

[10] H. Kim and S. Clarke. The relevance of AOP to an applications programmer in an EJB environment. In *First International Conference on Aspect-Oriented Software Development (AOSD) Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*, 2002.

[11] R. Laddad. Aspect-oriented refactoring. `www.theserverside.com`, December 2003.

[12] M. Lippert and C.V. Lopes. A study on exception detection and handling using aspect-oriented programming. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE)*, pages 418–427, Boston, MA, 2000. ACM Press.

[13] M. Marin, A. van Deursen, and L. Moonen. Identifying aspects using fan-in analysis. In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE2004)*., pages 132–141. IEEE Computer Society, 2004.

[14] A. Mesbah and A. Deursen. Crosscutting Concerns in J2EE applications. Technical Report SEN-R05xy, CWI, 2005.

[15] T. Murali, R. Pawlak, and H. Younessi. Applying aspect orientation to J2EE business tier patterns. In Y. Coady and D. Lorenz, editors, *Proc. of the 3rd AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*, Victoria, Canada, 2004. University of Victoria.

[16] Odysseas Papapetrou and George A. Papadopoulos. Aspect oriented programming for a component-based real life application: a case study. In *SAC '04: Proceedings of the 2004 ACM Symposium on Applied Computing*, pages 1554–1558. ACM Press, 2004.

[17] P. Slowikowski and K. Zielinski. Comparison study of aspect-oriented and container managed security. In *AAOS2003: Analysis of Aspect Oriented Software. Workshop held in conjunction with ECOOP*, 2003.

[18] Sergio Soares, Eduardo Laureano, and Paulo Borba. Implementing distribution and persistence aspects with AspectJ. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 174–190. ACM Press, 2002.

[19] Paolo Tonella and Mariano Ceccato. Migrating interface implementation to aspects. In *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM'04)*, pages 220–229. IEEE Computer Society, 2004.

[20] C. Zhang and H-A. Jacobsen. Quantifying aspects in middleware platforms. In *Proc. 2nd Int. Conf. on Aspect-Oriented Software Development (AOSD-2003)*, pages 130–139. ACM Press, March 2003.

[4] http://aspectwerkz.codehaus.org/