SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2000; **00**:1–6 *Prepared using* speauth.cls *[Version: 2002/09/23 v2.2]*

Distribution and Persistence as Aspects

S. C. B. Soares[1,*] and P. H. M. Borba[2,†] and E. A. G. C. Laureano[2,‡§]

[1] *Pernambuco State University, Computing Systems Department, Madalena, Recife - PE, CEP 50720-001, Brazil.*[2] *Federal University of Pernambuco, Informatics Center, Cidade Universitária, Recife - PE, Caixa Postal 7851, CEP 50732-970, Brazil.* [1] *Microsoft Corporation, One Microsoft Way - 50/1315. Redmond, WA 98052, USA.*

SUMMARY

This paper reports our experience using AspectJ, a general-purpose aspect-oriented extension to Java, to implement distribution and persistence concerns in a web-based information system. This system was originally implemented in Java and restructured with AspectJ. Our main contribution is to show that AspectJ is useful for implementing several persistence and distribution concerns in the considered application, but also in similar applications. We have also identified interferences between the implemented aspects and a few drawbacks in the language, so we suggest some minor language modifications that could significantly improve similar implementations. Despite those problems, we argue that the AspectJ implementation is superior to the pure Java implementation. Some of the aspects implemented in our experiment are abstract and constitute a simple aspect framework. The other aspects are application specific but we suggest that different implementations might follow the same aspect patterns. The framework and the patterns allow us to propose architecture-specific guidelines that provide practical advice for both restructuring and implementing certain kinds of persistent and distributed applications with AspectJ. Copyright © 2000 John Wiley & Sons, Ltd.

KEY WORDS: Aspect-oriented programming; separation of concerns, framework, persistence, distribution

*Correspondence to: Escola Politécnica da Universidade de Pernambuco, Departamento de Sistemas Computacionais, Rua Benfica, No 455, Madalena, CEP 50720-001. Recife - PE - Brazil.
*E-mail: sergio@dsc.upe.br
†E-mail: phmb@cin.ufpe.br
‡E-mail: edlaure@microsoft.com
§This work was done while the first and third authors were at Informatics Center of the Federal University of Pernambuco.
Contract/grant sponsor: CNPq and CAPES.

INTRODUCTION

In this article we give an extended and revised presentation of our work [1] on using AspectJ [2], a general purpose aspect-oriented [3, 4] extension to Java [5], to implement distribution and persistence aspects in a simple but real and non trivial web-based information system, a health complaint system, which was originally implemented in Java. In fact, the aspects implementation we present here had considerable evolved since the system restructuring resulting in improved and more reusable aspects.

The distribution aspects implement basic remote access to system services using Java RMI *Remote Method Invocation* [6]. The persistence aspects implement basic persistence functionality using relational databases, and support the following main concerns: connection and transaction control, storage medium customization, and partial (shallow) object loading. Additionally there is an aspect for synchronizing object states with the corresponding database, when the software uses persistent data management, and/or with the corresponding remote server, when the software is distributed, in order to ensure consistency. During implementation of those aspects, it was necessary to define auxiliary exception handling aspects, which we also present here. We discuss the lessons learned implementing those aspects and justify our design decisions.

The main contribution of our restructuring experience is to show that AspectJ is useful for implementing several persistence and distribution concerns in the kind of application considered, but we have also identified a few drawbacks in the language and suggest some minor modifications that could significantly improve implementations similar to the ones discussed here. Moreover, we mention other development difficulties that could be minimized by proper tools and processes for aspect-oriented development. We also argue that the AspectJ implementation of the health complaint system is superior to the pure Java implementation.

Some of the aspects implemented in our experiment are abstract and constitute a simple aspect framework. These abstract aspects can be extended for implementing persistence and distribution in other applications that comply with the architecture of the health complaint system, a layer architecture used for developing web-based information systems. The other aspects are application specific and therefore have different implementations for different applications. Nevertheless, we suggest that different implementations might follow a common aspect pattern, having aspects with the same structure. Based on the framework and the pattern, we propose architecture specific guidelines that provide practical advice for both restructuring and implementing certain kinds of persistent and distributed applications with AspectJ. Those guidelines are actually a systematic description of a real world application implemented with AspectJ including a documentation of design rationale in the Health Watcher Aspects Section.

ASPECTJ OVERVIEW

AspectJ [2] is a general purpose aspect-oriented extension to Java. The aspect-oriented constructs support the separate definition of crosscutting concerns—concerns that affect several units of a system. This separation of concerns allows better modularity, avoiding

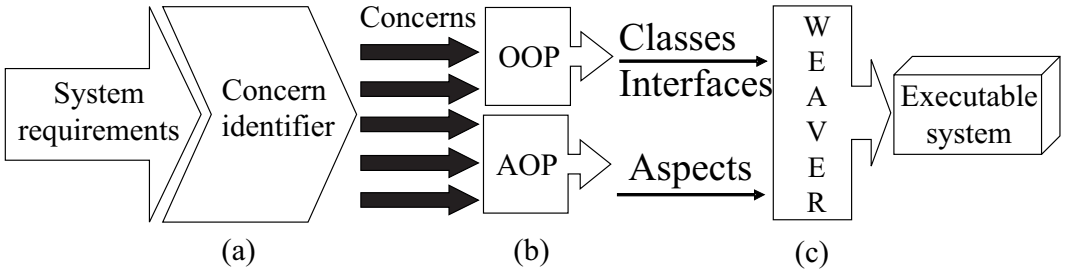

Figure 1. Aspect-Oriented development phases.

code tangling and code scattering over several units. Therefore, system maintainability is also increased.

Programming with AspectJ uses both objects and aspects to separate concerns. Concerns that are well modeled as objects are separated that way; concerns that crosscut the objects are separated using units called aspects, and those are composed with the objects of a system by a process called weaving. By weaving AspectJ aspects with standard Java code, we obtain a new AspectJ application.

Figure 1 illustrates aspect-oriented development phases. The first phase (Figure 1.a) is to identify the system's concerns necessary to implement the system requirements[†]. After that, the concerns that can be well implemented as objects, are implemented in this way, using object-oriented programming, and the crosscutting concerns are implemented using aspect-oriented programming, in order to increase the system's modularity (Figure 1.b). Finally, the aspects and the classes are composed, by a process called weaver (Figure 1.c), to obtain the final version of an application with the required functionality.

In fact, the phases illustrated by Figure 1 have to be accommodated in the context a software development process. Although requirements, analysis and design activities are crucial for any software development, we are focusing only on experience with AspectJ and not on experience of using AO across the life cycle. Despite not considering such activities, they should be placed in the Concern identifier phase. Considerations on that issue can be found elsewhere [7].

The main construct of the AspectJ [2] language is called *aspect*. Each aspect defines a functionality that crosscuts others, called crosscutting concerns, in a system. An aspect can declare attributes and methods, and can extend another aspect by defining concrete behavior for some abstract declarations.

An aspect can affect the static structure of Java programs, by using AspectJ's static crosscutting mechanism. This mechanism allows one to introduce new methods and fields

[†]It is possible to derive several concerns from a single system requirement.

```
 1: import java.sql.*;
 2:
 3: aspect DatabaseDebugging {
 4:
 5:   pointcut queryExecution(String sql):
 6:     call(* Statement.execute*(String)) &&
 7:     args(sql);
 8:
 9:   before(String sql): queryExecution(sql) {
10:     System.out.println(sql);
11:   }
12: }
```

Figure 2. AspectJ example.

to an existing class, convert checked exceptions into unchecked exceptions, and change the class hierarchy by, for example, making an existing class extend another one.

Aspects can also affect the dynamic structure of a program by changing the way a program executes. They can intercept certain points, called *join points*, of the program execution flow and add behavior *before*, *after*, or *around* the join point. Examples of join points are method calls, method executions, constructor executions, field references (get and set), exception handling, static initializations, and combinations of these using the logical `!`, `&&` and `||` operators. An interesting set of join points can match all the execution points of an execution flow, and it is usually used combined to others in order to better identify joint points to be affected

Usually, an aspect declares *pointcuts* that select sets of join points and context values at those join points. The aspect also declares *advice* in order to specify the piece of code that should be executed when a pointcut is matched during execution. The advice declaration indicates if the code should execute before, after, or around the pointcut. An aspect has access to reflective information of the join points through an attribute (`thisJoinPoint`).

Figure 2 shows a simple but powerful aspect definition[‡]. The aspect `DatabaseDebugging` defines a pointcut (line `5`) to identify calls to methods with names starting with `execute` of the JDBC interface `Statement` that receives a string parameter (line `6`) and expose the parameter (line `7`) of the method call. The `Statement` methods named `execute*` are responsible to execute SQLs statements into the database. The aspect also defines an advice (line `9`) that prints the SQL strings before the execution of the methods identified by the pointcut. Note that this 7

[‡]For simplicity and legibility reasons we omit the `public` visibility modifier in every piece of code throughout the paper

lines aspect will affect every point of the system that makes an access to the database using the `java.sql.Statement` interface.

THE HEALTH WATCHER SYSTEM

The Health Watcher, the information system used in our experiment, is a real health complaint system developed to improve the quality of the services provided by health care institutions. By allowing the public to register several kinds of health complaints, such as complaints against restaurants and food shops, health care institutions can promptly investigate the complaints and take the required actions. The system has a web-based user interface for registering complaints and performing several other associated operations.

In order to achieve modularity and extensibility, a layer architecture and associated design patterns [8, 9, 10] were used in the Java implementation of the system. This layer architecture helps to separate data management, business, communication (distribution), and presentation (user interface) concerns. This structure leads to less tangled code—such as when business code interlaces with distribution code—but does not completely avoid it. For example, the code for starting and terminating transactions, in general, cannot be easily untangled by using this architecture and an object-oriented language. Moreover, in the cases where it can be untangled, one has to pay a high price for that: adapters have to be written just to take care of the transaction functionality. Another example is the code for providing data access on demand, which cannot be untangled too.

The layer architecture of the Health Watcher system does not prevent scattering code too. This is the case of the code specifying which classes have to be serializable for allowing the remote communication of its objects. The exception handling code is also scattered throughout the system. The transactions code appears only in the facade class [8], the unique entry point to the system, but it is essentially replicated on all transactional methods of this class.

Despite not completely separating concerns, the layer architecture gives some support to adaptability. Figure 3 shows two possible system configurations, where a relational database is used as the persistence mechanism accessed through JDBC. In the one used in our restructuring experience, the system is used through an HTML [11] and Javascript [12] user interface, which interacts with Java servlets [13] running in a web server. In the other configuration, a Java user interface interacts directly with an application server using Java RMI. Instead of RMI, it would be possible to use EJB [14, 15] (Enterprise JavaBeans) or another distribution technology. Similarly, we could also have an object-oriented database as the persistence mechanism. Moreover, for making tests easier and allowing early functional requirements validation, we could not use a persistence mechanism at all, but test and validate the system using nonpersistent data structures. After the system is mostly validated, we could then implement the persistence code. This kind of flexibility was desirable for the Health Watcher system, and justified the use of the layer architecture and some of the design decisions that we discuss later.

Although some researches might think counter productive using nonpersistent data collection to test the software before plugging in a persistent collection [16], the use of nonpersistent data collections aims in early identifying requirements change. This increases productivity since the persistence code is only implemented after such validation [7], avoiding rewriting persistence

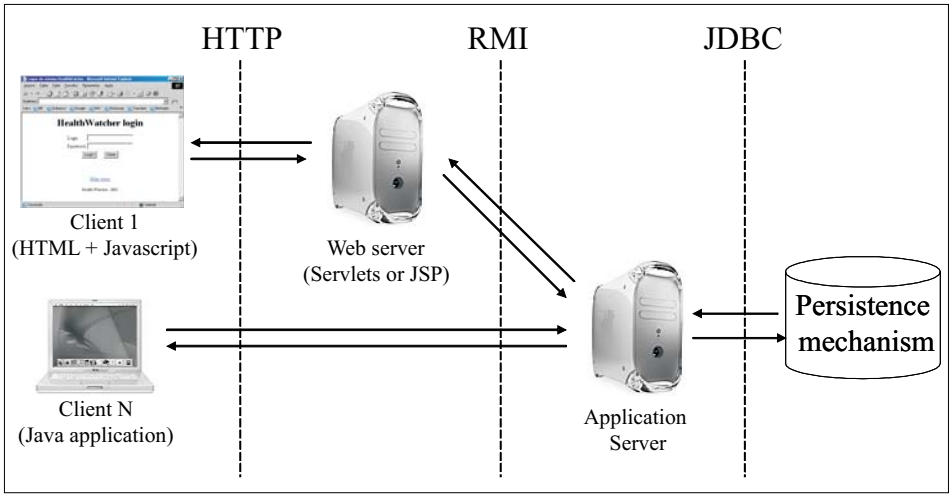

Figure 3. System configurations.

code if requirements change. In fact, the nonpersistent data collection should be automatically generated [7, 17], also increasing productivity.

Figure 4 presents part of the Health Watcher UML [18] class diagram. For simplification, it only shows the classes involved in the complaint processing services, the others essentially follow the same pattern [10]; we also omit the classes from the communication layer, which allow remote access to system services. Complaints are registered, updated, and queried through a web client implemented using Java servlets. Accesses to the Health Watcher services are made through its facade (`HWFacade`), which is composed of business collections. The interface `IPersistenceMechanism` abstracts which persistence mechanism is in use. Classes implementing this interface (`PersistenceMechanismRDBMS`) should handle database connections and transaction management. In fact, this concrete class can be reused in other developments that use the same kind of database. Persistent data collections (`ComplaintDataRDBMS`) are used to map persistent data into business basic objects (`Complaint`), and vice versa. Those collections are used by business collections (`ComplaintRecord`) through business-data interfaces (`ComplaintData`). These interfaces allow multiple implementations of the data collections, using different data storage and management mechanisms, including nonpersistent data structures (`ComplaintDataArray`).

To exemplify the tangling problem, the following piece of code shows some lines of the object-oriented version facade class, a business class. In the first line, there is a reference to a remote interface, which is code specific for distribution implementation. The class also has persistence specific code. The `IPersistenceMechanism` attribute (line 2) is responsible to allow implementing persistence transactions.

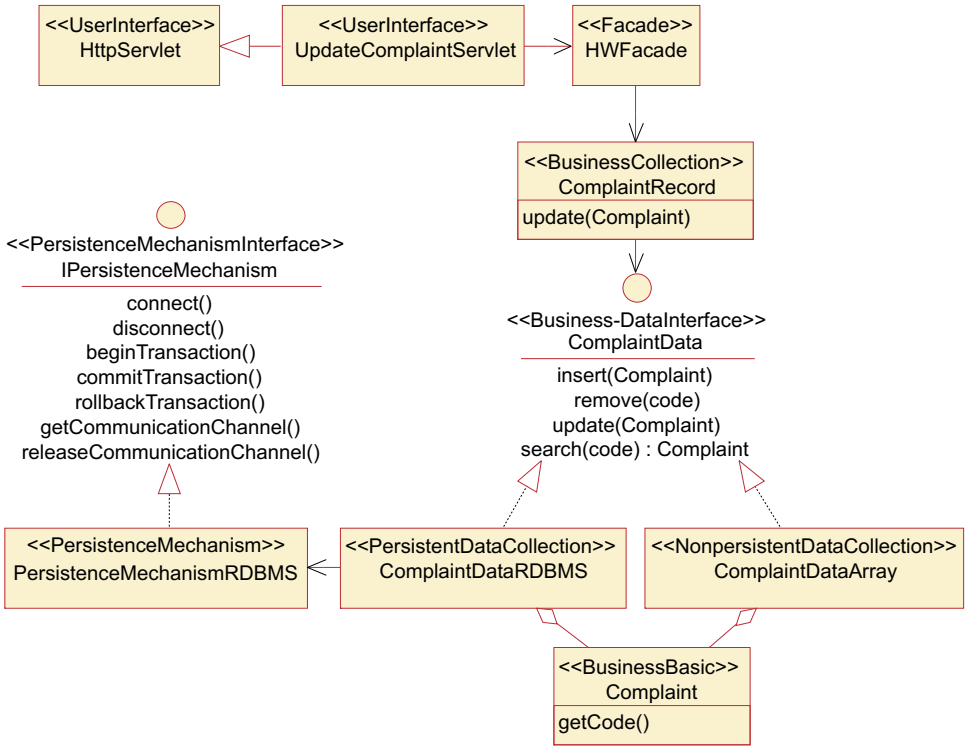

Figure 4. Partial Health Watcher class diagram.

```
1:   public class HWFacade implements IRemoteFacade {
2:     private IPersistenceMechanism pm;
3:     private ComplaintRecord complaintRecord;
4:     // ...
5:     public void update(Complaint complaint) {
6:       try {
7:         this.pm.beginTransaction();
8:         this.complaintRecord.update(complaint);
9:         this.pm.commitTransaction();
10:      } catch (ObjectNotFoundException e) {
11:        this.pm.rollbackTransaction();
12:        throw e;
13:      } // other catch blocks for each throwable exceptions
14:    } //...
15:  }
```

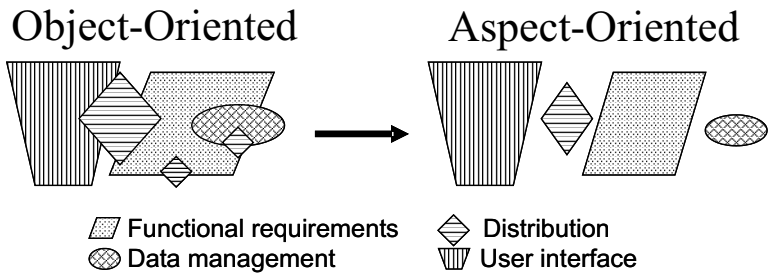

Figure 5. Aspect-oriented Health Watcher restructuring.

The transactions are implemented in every facade methods. In this example, the transactions are implemented from line `6` to `13`, plus other catch blocks to rollback the transaction if other exceptions are raised. In fact, if we do not consider transaction implementation, this method would be implemented only by line `8`. The aspect solution to this tangling is presented in the Health Watcher Aspects Section.

HEALTH WATCHER ASPECTS

In order to minimize the deficiencies of the pure Java implementation of the system considered in our experiment (see Section "The Health Watcher System"), AspectJ was chosen to restructure the system to implement distribution and persistence concerns. By doing this, we aimed to achieve better separation of concerns and avoid some tangling and scattering that cannot be avoided by simply using the layer architecture. Therefore, we hoped to obtain a more extensible system, supporting, without invasive changes, several different configurations required by the Health Watcher stakeholders.

Figure 5 depicts the restructuring we made in the object-oriented Health Watcher system resulting the aspect-oriented Health Watcher. In the object-oriented version, different concerns are tangled with each other and scattered over several places of the system. In the aspect-oriented version, the concerns are physically separated and well modularized, not scattered. As previously mentioned, these separated concerns should be woven to result the distributed and persistence version of the system.

DISTRIBUTION ASPECTS

In this section, we concentrate on the distribution aspects. Later we consider the persistence aspects and the complementary exception handling aspects. Those aspects are presented by discussing the steps we performed towards restructuring the pure Java version of the system and obtaining the AspectJ version. Although those steps are not generally applicable for all

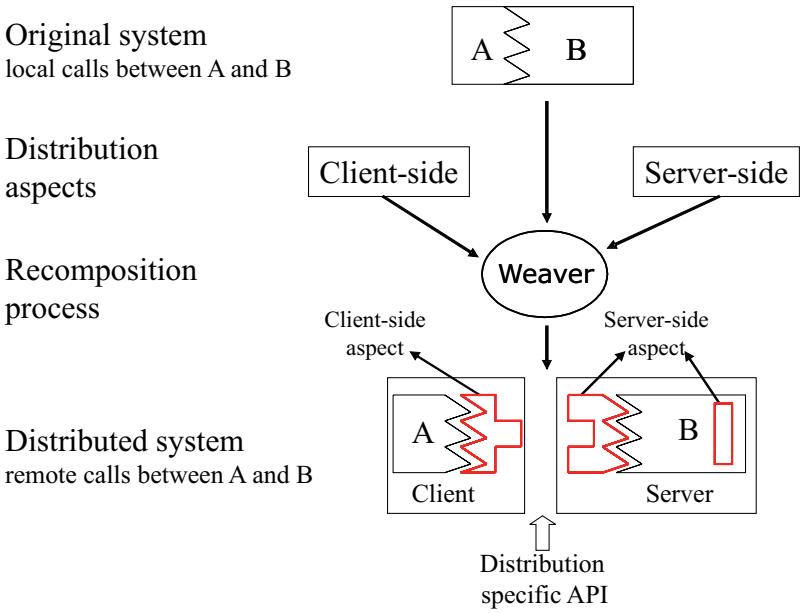

Figure 6. Distribution code weaving.

kinds of applications, we believe they can be used as specific guidelines for implementing distribution and persistence aspects in systems that comply with the architecture presented in Section "The Health Watcher System". They can, likewise, be used as guidelines for restructuring such systems. Actually, the aspects we present had evolved, since their first version [1], to a more reusable aspects definition.

In order to allow reuse we implemented an aspect hierarchy composed of abstract aspects, which are system-independent, and concrete aspects, which are specific to the Health Watcher system. In fact, the abstract aspects comply with the software architecture, like all others guidelines and abstract aspects we defined. Those abstract aspects define an aspect framework that provides aspect reuse, helping programmers to implement those concerns in other systems.

The first step of the restructuring process for separating the distribution code was to remove the RMI specific code from the pure Java version of the system. Roughly, in a system that complies with the architecture presented in Section "The Health Watcher System", the RMI distribution code is tangled in the facade class (server-side) and in the user interface classes (clients-side). Furthermore, the business basic classes also have some RMI code if their objects are arguments and return values of the facade's methods, which are remotely executed.

The RMI code was removed from the mentioned server and client classes and a similar functionality was separately implemented in associated server-side and client-side aspects, as explained by the following sections. Those distribution aspects are weaved (composed) to the

system resulting in the distribution version, as shown in Figure 6. This seems to be a common AspectJ pattern [19], where the aspects glue the functionality of their associated classes to the original system code. In fact, our distribution code consists of distribution aspects and auxiliary classes or interfaces. When this code is woven with the system code, it essentially affects the system facade and the user interface classes; the communication between them becomes remote by distributing the facade instance.

Server-side distribution aspect

The server-side distribution aspect is responsible for making the facade instance remotely available. It also ensures that the methods of the facade have serializable parameters and return types, since this is required by RMI.

Implementing a reusable aspect

RMI remote objects implement a so-called remote interface, which is used to access the remote services provided by those objects. Therefore, we can define an abstract aspect that uses this interface to generalize the server side behavior.

Additionally, remote objects are required to extend the RMI `UnicastRemoteObject` class, which defines the behavior of remote objects and makes their references remotely available. This approach, although recommended by RMI specification, would require the server-side aspect to add `RemoteException` to the `throws` clause of the facade's constructor. This would be necessary because the subclass (system facade) constructor calls the superclass (`UnicastRemoteObject` in this case) constructor, which declares that it might throw `RemoteException`. Unfortunately, the current version of AspectJ does not support that kind of static crosscutting. It can introduce, for example, methods, fields, and `implements` declarations, but not exceptions to a `throws` clause.

As we could not make the facade extend `UnicastRemoteObject`, we obtained a similar effect using an RMI alternative. The `exportObject` static method, declared in `UnicastRemoteObject`, was used to export the facade instance and make it remotely available. This method is called by the facade `main` method, which essentially starts up the remote Health Watcher server.

The abstract aspect defines an abstract pointcut to identify the execution of the facade `main` method. It also defines abstract methods to initialize the remote instance, to get the instance and the server names.

```
abstract aspect ServerSide {
  abstract pointcut facadeMainExecution();
  abstract Remote initFacadeInstance();
  abstract String getSystemName();
  abstract String getServerName();
```

Those abstract methods and pointcut are implemented by system specific aspects. The abstract aspect also defines and `around` advice that uses the abstract methods to export the facade instance and make it remotely available.

```
  void around(): facadeMainExecution() {
    try {
      Remote facade = initFacadeInstance();
      String systemName = getSystemName();
      UnicastRemoteObject.exportObject(facade);
      java.rmi.Naming.rebind("/"+ systemName, facade);
    } catch (RemoteException rmiEx) { ... }
    } catch (MalformedURLException rmiEx) { ... }
  }
}
```

In fact, this advice replaces the entire original `main` method, and therefore, the original behavior if despised. However, usually, the `main` method of a facade class might not even exist or it may only has tests over the system. It is very unusual a `main` method to be executed while the system is being executed.

Defining a concrete aspect

As previously mentioned, RMI remote objects must implement a remote interface. Hence, the concrete server-side aspect has to modify the facade class (`HWFacade`) to implement a corresponding remote interface (`IRemoteFacade`), which extends the RMI remote interface (`java.rmi.Remote`). Since the `IRemoteFacade` interface is only needed to implement distribution, its relationship with the system facade was implemented by the distribution aspect. This is done by using AspectJ's `declare parents` construct:

```
aspect HWServerSide extends ServerSide {
  declare parents: HWFacade implements IRemoteFacade;
```

The `IRemoteFacade` interface, which is distribution specific code, was part of the pure Java version of the system, so we did not have to implement it again. In the AspectJ version of the system, this interface is specific and auxiliary to the distribution aspects, so it was grouped with the other auxiliary types. Besides extending RMI's `Remote` interface, this interface contains the signatures of the facade public methods, however, adding the `java.rmi.RemoteException` in their `throws` clauses. This exception is used by RMI in order to indicate several kinds of configuration problems and remote communication failures.

The concrete aspect must implement the abstract members of the abstract aspect in order to specialize it to a specific system. The methods

```
  Remote initFacadeInstance() { return HWFacade.getInstance(); }
  String getSystemName()      { return "HealthWatcherSystem"; }
  String getServerName()      { return /* server IP or DNS address */ }
```

and the pointcut

```
  pointcut facadeMainExecution(): execution(static void HWFacade.main(..));
```

As previously mentioned, facade class might not have a `main` method, and when this happens, we should add an empty one in order to enable the super aspect. This can be done through the use of AspectJ inter-type declarations, which adds a `main` method into the `HWFacade` class.

Serializing types

As demanded by RMI, the concrete server-side aspect should also make serializable all parameters and return types of the facade methods. Exceptions are for parameter and return values that correspond to remote objects themselves.

In order to be serializable, a class has to implement the Java `Serializable` interface, which indicates that default object serialization should be available for its objects. So the aspect simply uses the `declare parents` construct for each parameter and return type that should be serializable:

```
    declare parents: healthGuide.HealthUnit || ... || complaint.Complaint
                     implements java.io.Serializable;
}
```

This might indeed be repetitive and tedious, suggesting that either AspectJ should have more powerful metaprogramming constructs [7, 20] or code analysis and generation tools [7, 17] would be helpful for better supporting this development step. Those tools would be even more useful for the pure Java implementation, where we have to write basically the same code, but in a tangled and scattered way.

Client-side distribution aspect

A simple implementation of the client-side aspect would make the client (user interface) classes refer to the remote facade instance. They all have a `HWFacade` field that should yield the remote instance when accessed. At first, it seems that this could be easily achieved with AspectJ by intercepting the accesses to those fields. However, due to RMI conventions, the type of the remote reference is actually `IRemoteFacade`. So the remote reference is not assignable to the `HWFacade` fields and, consequently, those cannot yield that reference when accessed. This problem could be avoided if the client classes had `IRemoteFacade` fields, but those classes would then depend on RMI code, decreasing system modularity.

If the remote reference had `HWFacade` type, another possibility would be to intercept calls to the facade methods, making sure that the calls are directed to the remote facade instance. This could be achieved by first defining a pointcut (line 1) to identify calls to the non-static `HWFacade` methods (lines 2 and 3), as long as they originate from the user interface classes (line 3), which in our case are Java servlets:

```
1:  pointcut facadeCalls(HWFacade local):
2:    target(local)          && call(* *(..))  &&
3:    !call(static * *(..)) && this(HttpServlet);
```

In this code, the pointcut parameter `local` indicates that we want to expose some value in the execution context of the associated join points. We use the `target` designator to bind

the `local` pointcut parameter to the target of the method calls, and the `this` designator to indicate that the currently executing object has type `HttpServlet`.

Besides identifying the join points of the facade method calls, we would define an `around` advice (line 4) to affect those join points by substituting the reference to the local facade instance (the target of the call) with the reference to the remote facade instance:

```
Object around(HWFacade local) throws /*...*/: facadeCalls(local) {
  return proceed(remoteHW);
}
```

This advice affects the facade calls, exposing the reference to the target of each call (line 4). It uses a reference to the remote instance (`remoteHW`, declared and initialized by the aspect) to proceed with the execution flow (line 5), but changing the execution context. This is done by changing the exposed reference to the target of the call: instead of the reference stored in `local` it becomes the one stored in `remoteHW`. This advice, however, would only be valid if the type of `remoteHW` were `HWFacade`, the type of the advice parameter, instead of `IRemoteFacade`.

Redirecting method calls

As the discussed solutions do not work with the current version of AspectJ, we had to write an advice for each facade method, essentially doing the same thing as the previous around advice, but in a specific way for each single facade method. For example, the advice for the `update(Complaint)` method is the following:

```
int around(Complaint c) throws /*...*/:
    facadeCalls() && call(void update(Complaint)) && args(c) {
  return remoteHW.update(c);
}
```

It redirects the `update` calls to the facade remote instance. However, this is not done by changing the value of the target of the call, as in the general around advice shown before. Here the around advice does not proceed with the execution of the original call, but executes a new call to the same method, with the same argument, but with a different target. Since we do not change the value of any variable, we avoid the typing problems, with `HWFacade` and `IRemoteFacade`, discussed before. The `facadeCalls` pointcut used in this advice would be essentially the same as the one we have shown before, but does not need to expose a reference to the target of the call.

The advices for the other facade methods are quite similar to this one. In fact, this solution works well but we loose generality and have to write much more tedious code. It is also not so good with respect to software maintenance: for every new facade method, we should write an associated advice, besides including a new method signature in the remote interface.

Defining an abstract aspect

In order to solve this problem and allow reuse, we define an abstract aspect that uses Java reflection and AspectJ features to redirect facade methods calls. The aspect defines an abstract

pointcut to identify facade calls, and abstract methods to retrieve the name bound to the remote instance and the name of the server where the remote instance is available.

```
abstract aspect ClientSide {
  private Remote facade;
  abstract String getSystemName();
  abstract String getServerName();
  abstract pointcut facadeLocalCalls();
```

These abstract methods are not specific to the Health Watcher system, making the aspect more general. The aspect also defines the lookup service to retrieve the remote instance, assigning it to the `facade` field.

```
  private synchronized Remote getRemoteFacade() {
    if (facade == null) {
      String systemName = getSystemName();
      String serverName = getServerName();
      try {
        facade = java.rmi.Naming.lookup("//" + serverName +
                                        "/"  + systemName);
      } catch (Exception ex) {
        throw new SoftException(ex);
      }
    }
    return facade;
  }
```

Note that the method wraps any exception raised by the `lookup` method execution into a `SoftException`. This is our approach to any exception raised by a concern implemented in the aspects, since the exception handling aspects will be responsible to handle them. The aspect also defines an `around` advice that uses the AspectJ API to access the arguments and name of the method being called, and then to call the corresponding method of the remote instance.

```
  Object around(): facadeLocalCalls()  {
    Object[] args = thisJoinPoint.getArgs();
    Signature signature = thisJoinPoint.getSignature();
    String methodName = signature.getName();
    return MethodExecution.invoke(getRemoteFacade(), methodName, args);
  }
}
```

The auxiliary class `MethodExecution` defines the static method `invoke` to call method of objects using the Java reflection API. This approach simplifies and generalizes facade method redirection. Despite simplifying the number of lines of code and generalizing the method redirection, the use of reflection decrease code readability and avoids static type checking,

which may favor adding errors into the program. On the other hand, this abstract aspect is supposed to be reused, and therefore, should not be implemented by the programmer. The programmer has only to implement the concrete aspect, which is system-specific, inheriting the redirection behavior.

Defining a concrete aspect

The next step is to define a concrete aspect that should implement the `ClientSide` abstract methods and define the abstract pointcut to identify facade methods call, as in the following pointcut.

```
pointcut facadeLocalCalls():
  this(HttpServlet) && call(* IRemoteFacade+.*(..)) &&
  !call(static * IRemoteFacade+.*(..));
```

This solution is quite superior to the corresponding pure object-oriented implementation. In fact, without using AspectJ and this approach, we would have productivity and maintenance problems. For instance, a common pattern for separating the distribution code in a pure Java implementation is to use factories and a pair of adapters [8, 9] between the facade and the user interface classes. However, in this way, we need to write much more code and a change in the facade class would require changing two classes besides the facade and the remote interface. Besides that, this alternative pure Java implementation cannot separate the distribution code at all, not satisfying the Health Watcher's adaptability and extensibility requirements.

Feature request

Some problems we hade during system restructuring could be avoided if we could add an exception in a method `throws` clause. Therefore, we submitted a *feature request* to the AspectJ team, which they expect to consider for a following version of AspectJ. We suggested the support of a new constructor that adds an exception to a method `throws` clause. For example, it could be used as in the following declaration, where the wildcard `*` is used to match any return type and any method name, and the wildcard `..` matches any parameter list:

```
declare throws: (* IRemoteFacade.*(..)) throws RemoteException;
```

This declaration would add the RMI specific exception, `RemoteException`, to the `throws` clause of all methods of the `IRemoteFacade` interface, assuming that this interface simply contains the signature of the public methods of the facade; it would not extend `Remote` and its methods would not throw `RemoteException`, this should be implemented by the aspect. In this way the client classes could have `IRemoteFacade` fields, since the RMI details would be introduced to the interface by the distribution aspects. The general solution shown at the beginning of this section could then be used; we should only replace `HWFacade` for `IRemoteFacade` in the `facadeCalls` pointcut definition.

The proposed feature would be useful to solve similar problems mentioned elsewhere [21], reinforcing the need for its support. It would allow static exception checking, as opposed to

the use of AspectJ's so-called softened exceptions, which are unchecked and therefore can be thrown anywhere, without further declarations. However, this feature must be used with care. It has to be used together with aspects that handle the newly added exceptions, otherwise a well-typed Java program, when woven with the aspect code, might yield a non well-typed program that does not handle some thrown exceptions. In fact, this feature does not have good compositionality properties.

Synchronizing states

When implementing the client-side aspect we had also to deal with the synchronization of object states. This was necessary because RMI supports only a copy parameter passing mechanism for non-remote arguments. So, when a facade method returns an object to the client, it actually returns a copy of the server-side object. Therefore, modifications to the client copy are not reflected in the server-side object.

The client-side aspect should take care of this distribution concern, and make sure that the modifications to the client copies are reflected on the server. This could be done by intercepting the user interface (client) methods and synchronizing the states of the server-side copies changed by those methods. The synchronization could be performed through calls to update methods declared by the system facade.

Later we concluded that this concern and its associated behavior are necessary for implementing persistence as well. Therefore, we actually implemented it only once, and the details are presented in Section "Data State Synchronization control". This shows that the distribution and persistence concerns are not completely independent. It also shows that careful design activities are also important for aspect-oriented programming. Only in this way we can detect in advance intersections, dependences and conflicts among different aspects. Consequently, we can avoid serious development problems and better plan the reuse and parallel development of different aspects.

Distributuion aspects class diagram

Figure 7 presents a class diagram of the distribution aspects and the Health Watcher classes the aspects affect or use, as described in this section. The `<<Aspect>>` stereotype and the fill color blue identify the aspects.

PERSISTENCE ASPECTS

This section presents the steps that we followed in order to restructure the persistence code of the Health Watcher system and obtain the corresponding persistence aspects. The first step in this direction was to remove the persistence code from the pure Java version of the system. In a system that complies with the architecture presented in Section "The Health Watcher System", the persistence code is mostly concentrated in the data collection and persistence mechanism classes, but also appears in the facade and in the business collection classes.

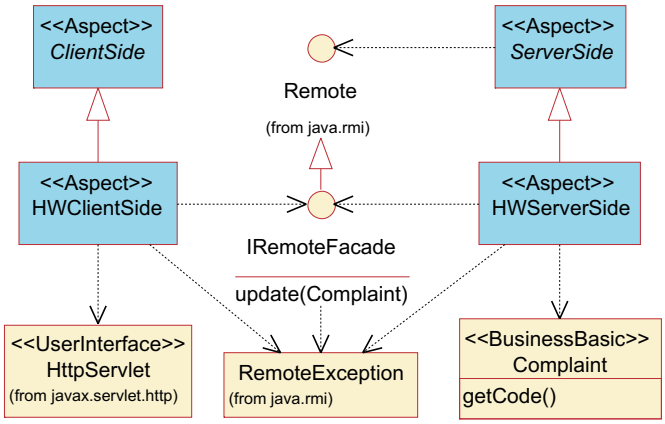

Figure 7. Distribution aspects class diagram

The persistence code was removed from the pure Java system and a similar functionality was implemented as aspects. Figure 8 illustrates that and also shows that we have aspects for making the system work with nonpersistent data structures. As discussed in Section "The Health Watcher System", this is useful for making testing easier and allowing early functional requirements validation, usually before the persistence code is written. When the persistence aspects are woven with the system code, we generate a persistent version of the system. The persistence source code includes the `IPersistenceMechanism` interface and implementations for this interface. The `IBusinessData` interfaces (see Section "The Health Watcher System") is responsible to make transparent to the business collection classes what data management medium they are using, and was not factored out from the core source code. The persistence aspects affect the facade and business collection classes.

The persistence code includes aspects and auxiliary classes and interfaces to address the following major concerns: connection and transaction control, partial (shallow) object loading and object caching for improving performance, and synchronization of object states with the corresponding database entities, for ensuring consistency. We now discuss most of them and briefly explain an auxiliary aspect for supporting data collection customization (one can choose between nonpersistent and persistent).

Persistence mechanism control

The persistence mechanism control aspects are responsible for implementing basic persistence functionality for all operations accessing the data storage mechanism. They create an instance of a persistence mechanism class (an implementation of `IPersistenceMechanism` provided by the persistence source code) and deal with database initialization, connection handling, and resources releasing, services provided through the created instance.

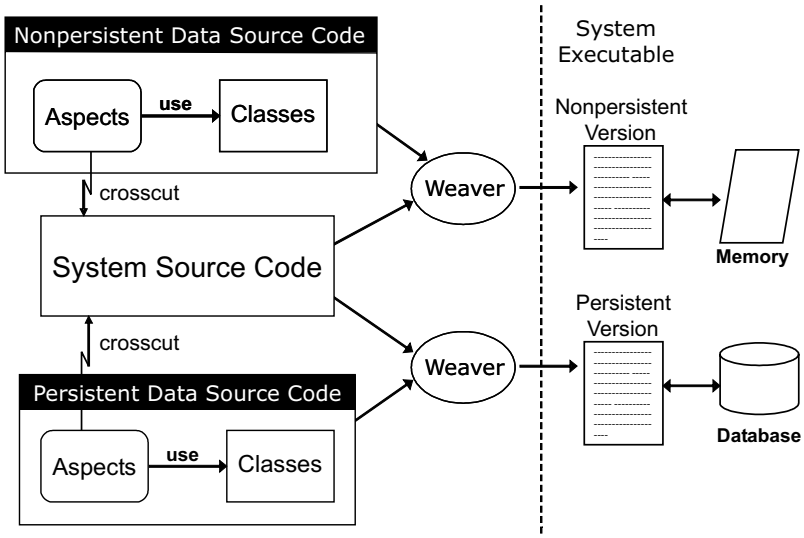

Figure 8. Data management code weaving.

For reuse purposes, this concern was implemented using an aspect hierarchy composed of an abstract aspect and a concrete aspect. The second is specific to the Health Watcher system, whereas the first can be used for implementing other systems that comply with the same architecture of the Health Watcher.

Implementing a reusable aspect

The abstract persistence mechanism control aspect is reusable. It defines two advice that depend on abstract pointcuts, which are made concrete by different concrete aspects, depending on the systems in which it is reused. This aspect (`AbstractPersistenceControl`) defines an abstract pointcut (`initSystem`) to identify the execution of the system initialization process; this is where an instance of a persistence mechanism class should be created and initialized.

```
abstract aspect PersistenceControl {
  abstract pointcut initSystem();
  abstract IPersistenceMechanism pmInit();
```

The aspect also declares an abstract method that should be used to initialize the persistence mechanism instance. Both the method and the pointcut are defined abstractly because their concrete definitions depend on specific classes of the system being implemented.

Two advices were declared to initialize and release resources; their implementations use the abstract pointcut previously defined:

```
before(): initSystem() {
  getPm().connect();
}
after() throwing: initSystem() {
  getPm().disconnect();
}
```

The `before` advice states that, before system initialization, a persistence mechanism instance is created and connected to the database system. If any problem happens during initialization, the `after throwing` advice is executed; the resources allocated by the persistence mechanism are then released. The persistence mechanism `disconnect` method is only called after an abnormal execution, since when the system is shutdown, the persistence mechanisms connections are released. To achieve a more explicit behavior, the aspect might override the `finalize` method to assure resources releasing before the object be garbage collected.

The `getPm` method creates, if necessary, and returns a valid `IPersistenceMechanism` instance.

```
synchronized IPersistenceMechanism getPm() {
  if (pm == null) {
    pm = pmInit();
  }
  return pm;
}
```

Those advices call methods that might raise exceptions, but it would not be interesting to handle them in the advice code, which will usually be executed before or after some facade code, during system initialization time. Therefore those exceptions were declared as *soft*, not checked, by the `declare soft` static crosscutting AspectJ construct. In Section "Exception Handling Aspects" we show how *soft* exceptions are handled.

Note that this aspect uses a single instance of the persistence mechanism for the whole application, but it is simple to adapt this aspect to work with a pool of persistence mechanisms, instead of just one, when required. For example, this would be necessary in a distributed database environment.

Implementing a concrete aspect

The abstract pointcut and method declared in the previous aspect were concretely defined for the Health Watcher system in the following aspect.

```
aspect HWPersistenceControl extends PersistenceControl {
  pointcut initSystem(): call(HWFacade.new(..));
  IPersistenceMechanism pmInit() {
    return PersistenceMechanismRDMS.getInstance();
  }
```

The pointcut definition states that the initialization point of the Health Watcher system is the creation of the facade (`HWFacade`) instance. This aspect also implements the

persistence mechanism initialization method, `pmInit`. This method obtains an instance of the concrete implementation of the persistence mechanism for relational databases, `PersistenceMechanismRBMS`, and then connects it to the database system, using the `connect` method.

As in the previous abstract aspect, we have to indicate that the persistence mechanism exception is soft when raised during the execution of the `getInstance` method:

```
pointcut obtainPmInstance(): call(* PersistenceMechanismRDMS.getInstance(..));
declare soft: PersistenceMechanismException: obtainPmInstance();
```

The `obtainPmInstance` is just an auxiliary pointcut.

The abstract aspect depends only on the persistence mechanism interface `IPersistenceMechanism`, benefiting software evolution, whereas the concrete aspect depends on a concrete persistence mechanism. Only the concrete aspect needs to be modified to support a different data storage mechanism such as object-oriented databases or another implementation for relational databases. The system can then be easily customized by simply replacing the concrete aspects and going through the weaving process.

By using factories, a similar kind of customization could also be achieved in the pure Java implementation of the system. This would require more code to be written. On the other hand, it would allow customization without recompiling the system code, at least for a pre-existing set of customization alternatives. This is not currently supported by AspectJ, but is expected to be. Moreover, with the pure Java version it would be expensive to separate the code for ordering the creation and use of the persistence mechanism. This would have to be tangled to the facade code. Since the tangled code would depend only on the `IPersistenceMechanism` interface, the main direct disadvantage in this case would be with respect to the legibility of the facade, instead of its reusability or extensibility. Those would be indirectly affected only.

Persistence control aspects class diagram

Figure 9 presents a class diagram of the persistence control aspects and the Health Watcher classes and interfaces the aspects affect or use, as described.

Transaction control

When dealing with data stored in a persistence mechanism it is essential to work with transactions in order to guarantee the ACID properties [22]: atomicity of operations, data consistency, isolation when performing operations, and data durability even if the system fails. In the Health Watcher system, the transaction control code was mostly invoked from the facade class. Therefore, we removed this code and implemented the transaction control concern using two aspects to improve reusability, similarly to what was done in the implementation of the previous concern.

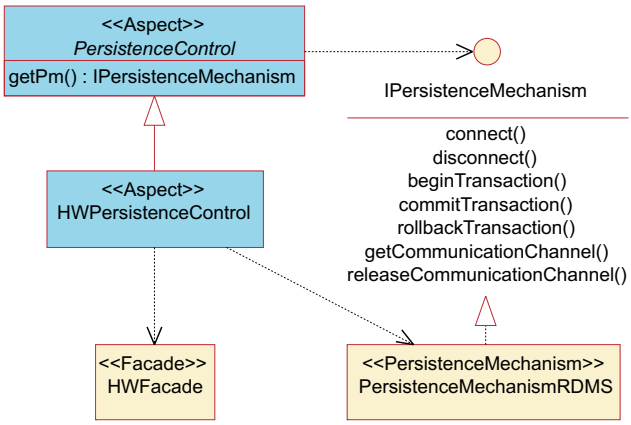

Figure 9. Persistence control aspects class diagram

Implementing a reusable aspect

The simplest version of the abstract transaction control aspect defines an abstract pointcut that should identify the transactional methods of the system; that is, the methods whose execution should be bound by a transaction:

```
abstract aspect TransactionControl {
  abstract pointcut transactionalMethods();
  abstract IPersistenceMechanism getPm();
```

It also declares an abstract method to obtain a valid persistence mechanism instance; it is necessary for invoking the transaction services supported by the persistence mechanism.

The abstract transaction control aspect also implements three advice to begin, commit, and rollback transactions. The first one is a `before` advice that starts a transaction just before the execution of any transactional method:

```
before(): transactionalMethods() {
  getPm().beginTransaction();
}
```

As in the previous aspect definition, we should declare that the exceptions raised by the methods called inside the advice are soft; we omit the code here.

We also have an `after returning` advice that commits the transaction when the method executions returns successfully:

```
after() returning: transactionalMethods() {
  getPm().commitTransaction();
}
```

At last, an `after throwing` advice rolls the transaction back to the original state, maintaining the database in a consistent state, if any problem happens during the execution of any transactional method:

```
after() throwing: transactionalMethods() {
  getPm().rollbackTransaction();
}
```

Notice that any exception that is thrown and not handled by a transactional method aborts the transaction. We had the same behavior in the pure Java version of the Health Watcher system. This decision was perfectly adequate for both versions of the system and we believe that it would be adequate for other systems too. Nevertheless, this shows that the programmer that writes the persistence aspects should be aware of the behavior of the affected code. Likewise, the programmer who wishes to reuse our transaction aspects should be aware of the effect of throwing, and not handling, an exception. In fact, there might be a strong dependency between the aspect code and the Java code [21]. In the transactions case, we do not think that this brings major problems in practice. In general, more powerful AspectJ tools would be necessary to provide multiple views, and associated operations, for the strongly related AspectJ and Java units of code. Current tools, such as AJDT for Eclipse [23] and others [24], only show the dependency between the Java code and the aspects that affect it.

Implementing a concrete aspect

The concrete transaction control aspect (`HWTransactionControl`) inherits from the previous abstract aspect and provides concrete definitions for the abstract pointcut and method.

When defining the concrete pointcut, we do not directly list the signatures of all transactional methods. This would affect aspect legibility and make the aspect code too much directly dependent on modifications on the method signatures. Therefore, we defined an interface containing the signatures of the transactional methods. This interface, `ITransactionalMethods`, is used by the pointcut to identify the transactional methods of the system. The pointcut matches the execution of all methods defined by the interface:

```
aspect HWTransactionControl extends TransactionControl {
  declare parents: HWFacade implements ITransactionalMethods;
  pointcut transactionalMethods(): execution(* ITransactionalMethods.*(..));
```

The aspect also uses the `declare parents` construct to make the facade class, which contains all transactional methods of the Health Watcher system, implementing the `ITransactionalMethods` interface. This is necessary for associating the methods that will be executed with the signatures in the interface.

The definition of the concrete method refers to the concrete persistence control aspect, which provides access to an instance of the persistence mechanism:

```
IPersistenceMechanism getPm() {
  HWPersistenceControl pc = HWPersistenceControl.aspectOf();
  return pc.getPm();
}
```

This method yields a valid instance of the persistence mechanism. This is done by obtaining the instance that is available in the `HWPersistenceControl` aspect, through its `getPm` method. We use the `aspectOf` method to obtain an instance of the aspect. This makes the concrete transaction aspect dependent on the concrete persistence control aspect, but the abstract aspects are independent of each other and can be reused and support different system customization alternatives. In fact, despite being syntactically independent, there is a semantic dependency between these abstract aspects, since persistent software usually also needs transaction control, and, probably, vice versa. The independency we claim relies in the fact that one does not provide nor uses services from the other.

With this approach, the aspect is not directly dependent on the transactional methods signatures, but the auxiliary `ITransactionalMethods` interface is completely dependent on them. In fact, the interface should contain a subset of the signatures of the methods defined by the facade class. Once again this suggests that either AspectJ should have more powerful metaprogramming constructs [7, 20] or code analysis and generation tools [7, 17] would be helpful for better supporting this development step. Those tools would semi-automatically extract information from the facade every time the facade code changes, minimizing maintenance problems.

The Health Watcher system with the transaction aspects is significantly better than the pure Java system. In the original system code, the transactional methods explicitly call methods for transaction control. They also have code for handling the associated exceptions. For each method, there are at least 6 lines of tangled code to call the transaction lifecycle methods and handle the exceptions. Factoring all these repeated lines of code in a single unit avoids tedious work and increases productivity. It also makes the code much easier to evolve, especially if modifications in the transaction control policies are required. In this way, the developers can be more focused on the more interesting aspects of transaction implementation and on the main functionality implementation.

Implementing alternative policies

The aspects illustrated in this section offer a uniform transaction control policy, which was useful for most situations in the Health Watcher system but might not be adequate for more complex or performance demanding systems. The same performance limitations are reported by a similar, although independently developed, AspectJ implementation of transactions in the context of the OPTIMA framework for controlling concurrency and failures with transactions [21]. However, slight variations of our implementation can offer several alternative policies and solve those limitations. For example, we could have different transaction implementations for read only and update operations (read transaction and write transaction, respectively). We could also have more than one class with transactional methods.

In order to support different transaction implementations, it is useful to define an appropriate interface hierarchy to indicate the different kinds of transactional methods. The hierarchy shown in Figure 10 establishes that all transaction control interfaces should extend `ITransactionalMethods`. Interfaces specifying read only methods should extend `ReadOnlyTransactionalMethods`, and interfaces specifying update methods should extend `UpdateTransactionalMethods`. The class that implements the transactional

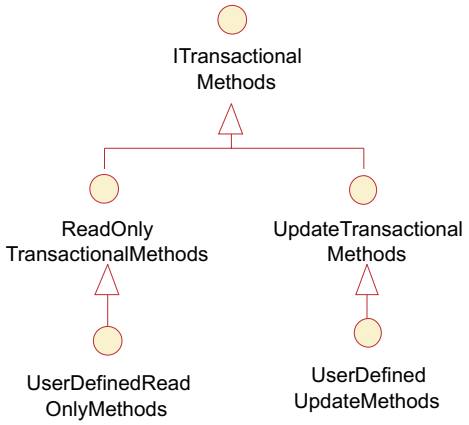

Figure 10. Transactional methods hierarchy.

methods should then implement the specific interfaces, instead of simply implementing `ITransactionalMethods`, as done before.

In addition to a different interface hierarchy, we should have variations of the abstract and concrete aspects. Instead of having a single pointcut, `transactionalMethods`, we should have two pointcuts, one for read operations (`readOnlyTransMethods`) and the other for write operations (`updateTransMethods`). Those pointcuts must match the execution of the methods of the associated interfaces. The abstract aspect should now have two sets of transactions advice, one set for each pointcut. Each set has a `before`, an `after returning`, and an `after throwing` advice, similar to the ones illustrated before. In this way, we can specify different behavior for the different kinds of transactional methods.

Roughly generalizing, the transaction control aspects should contain a pointcut and a set of three transactions advice for each kind of transactional method existing in the system. In an extreme situation, we could maybe imagine each transactional method having a different type of transaction implementation. In this case, the AspectJ version would only have a small advantage over the pure Java version: by removing the tangled code, the facade becomes simpler. On the other hand, considering that we do not have advanced AspectJ tools as discussed in the beginning of the section, there is a disadvantage too: as we separated related code, changes to a code unit might usually affect the other. However, these extreme situations do not seem to be usual. In fact, it seems that our AspectJ implementation of transactions can usually have significant advantages over pure Java implementations. That is certainly the case of systems such as the Health Watcher.

Another straightforward variation of the transaction control aspects supports multiple classes with transactional methods. In this case, one interface should be defined for each one of the classes. Those interfaces should extend the transactional methods interface `ITransactionalMethods`. For instance, suppose that the Health Watcher system contains

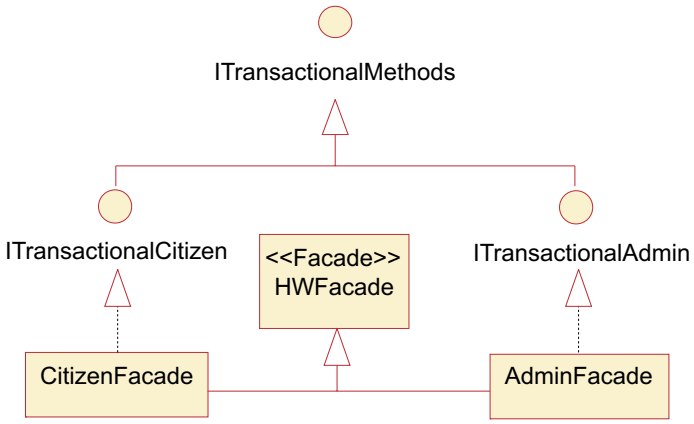

Figure 11. Example of multiple transactional components.

transactional methods in two classes: `CitizenFacade`, defining the main system services, and `AdminFacade`, containing system administration and configuration services. Therefore, we should define two interfaces: `ITransactionalCitizen` and `ITransactionalAdmin`. Figure 11 shows the UML class diagram for this hierarchy.

Besides having the extra interfaces, we should extend the concrete `HWTransactionControl` aspect to reflect this new structure:

```
declare parents: AdminFacade   implements ITransactionalAdmin;
declare parents: CitizenFacade implements ITransactionalCitizen;
```

The concrete `transactionalMethods` pointcut should also be modified to consider the executions of the methods declared in the new transactional interfaces.

Transaction control aspects class diagram

Figure 12 presents a class diagram of the transaction control aspects and the Health Watcher classes and interfaces the aspects affect or use.

Data collection customization

As explained before, the Health Watcher system should also work using nonpersistent data. In order to support this, two aspects were coded in such a way that we can build both application versions: nonpersistent and persistent. Each version is the result of weaving pure Java code with additional AspectJ code, as shown in Figure 8.

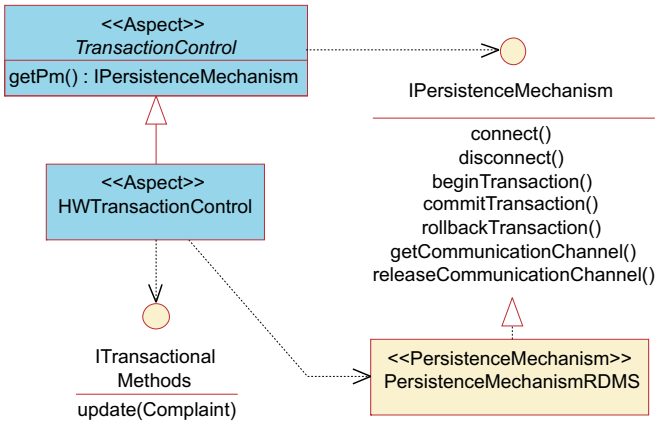

Figure 12. Transaction control aspects class diagram

Implement a system-independent reusable aspect

Again, we define an abstract aspect to increase aspect reuse. The aspect `DataCollectionCustomization` defines a pointcut to identify business collections creation

```
abstract aspect DataCollectionCustomization {
  pointcut recordsCreation():
    call(SystemRecord+.new(..)) && !within(DataCollectionCustomization+);
```

by using an auxiliary class `SystemRecord` that should be made superclass of the business collections classes by the system-specific aspect. The aspect defines an `around` advice to return a business collection using the configured data collection, which is chosen by a concrete aspect.

```
  SystemRecord around(): recordsCreation() {
    Signature signature = thisJoinPoint.getSignature()
    return getSystemRecord(signature.getDeclaringType());
  }
  protected abstract SystemRecord getSystemRecord(Class type);
}
```

The abstract method `getSystemRecord` should return the required business collection object based in the information of what class (business bollection) is being created in the affected join point. Note that this aspects uses information from the execution context through the variable `thisJoinPoint` [25].

This aspect simplifies the programmer work in the sense that the system-specific aspect that will implement the `getSystemRecord` method has only to define methods to retrieve the system-specific data collections for the required medium.

Implement an abstract system-specific aspect

Unlike the other abstract aspects, this one is system-specific in the sense that it must be defined for each system and therefore cannot be reused. The aspect `HWDataCollectionCustomization` identifies the system business collection classes and make them subclasses of `SystemRecord`:

```
abstract aspect HWDataCollectionCustomization extends
    DataCollectionCustomization {
  declare parents: ComplaintRecord || ... extends SystemRecord;
```

The aspect implements the `getSystemRecord` method by creating the corresponding business collection based on its `Class` information.

```
  protected SystemRecord getSystemRecord(Class type) {
    SystemRecord response = null;
    if (type.equals(ComplaintRecord.class)) {
      response = new ComplaintRecord(getComplaintData());
    } else  if (type.equals(...
    }
    return response;
  }
  protected abstract ComplaintData getComplaintData();
  ...
}
```

Note that the `getSystemRecord` method uses an abstract method `getComplaintData` that is responsible to create the data collection to be used by the business collection. We omit the code for the others business collections, but it is similar to the `ComplaintRecord` code. It is important to mention the need to change the `getSystemRecord` method every time a business collection is added or removed from the system. This abstract, but system-specific, aspect is necessary in order to allow other aspects to inherit from it in order to define the medium the software should use. The following concrete aspects definition depicts this.

The use of reflection, when the `Class` type is used by the `getSystemRecord` method, has some negative issues. For example, unlike the previous abstract aspect, this aspect is not part of the aspect framework. Therefore, the aspect has to be implemented by a programmer since it is system-specific. By using reflection, its code is more difficult to understand and reason about. On the other hand, this aspect implements only part of the reflection code, actually a simple part. The other part of the reflection is implemented in the previous abstract aspect, which is part of the framework.

Implement concrete aspects

For the persistent version, we have an aspect responsible for creating persistent data collections to the system, implementing the abstract methods responsible for retrieving the data collections. In fact, we should implement two aspects: `PersistentDataCollection` and

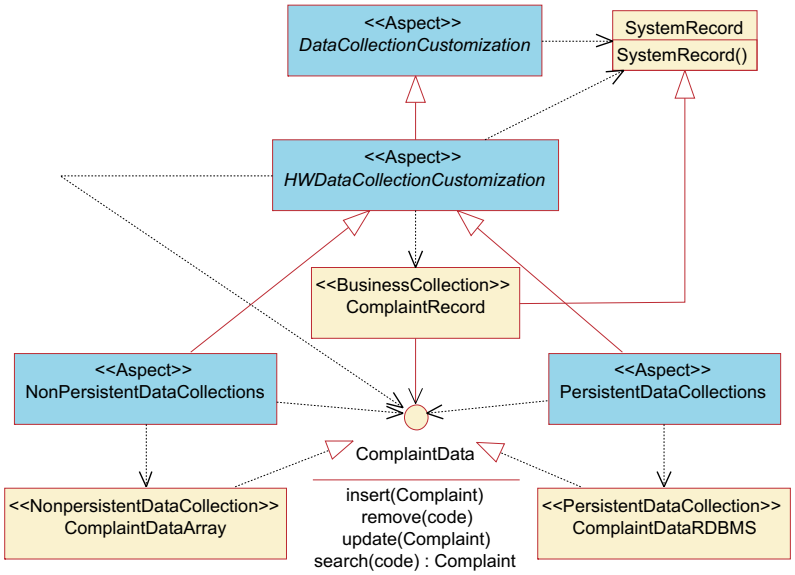

Figure 13. Data collection customization aspects class diagram

`NonpersistentDataCollection`, respectively for retrieving persistent and nonpersistent data collections.

This is possible because both persistent and nonpersistent data collections implement the same interface. Similar aspects can also be defined to associate different kinds of nonpersistent data collections. The aspect that associates the data collections using nonpersitent implementation is quite similar to the persistent aspect.

Therefore, in order to switch between persistent and nonpersistent versions of the system we should only switch between the `PersistentDataCollection` and `NonpersistentDataCollection` aspects when weaving.

As discussed in Section "Persistence mechanism control", this kind of customization can also be supported by the pure Java implementation, with several advantages and some disadvantages.

Data collection customization aspects class diagram

Figure 13 presents a class diagram of the data collection customization aspects and the Health Watcher classes and interfaces the aspects affect or use.

Data access on demand (partial object loading)

Objects might have a complex structure, being composed of several other dependent objects. In those cases, object storage and retrieval in data storage mechanisms need special care to avoid performance degradation. An adequate approach to access this kind of object is to parameterize the data loading level. For each kind of object usage, an adequate loading level should be defined. For example, a service that lists complaints may only need to access the complaints description and code, whereas a service that generates a complex report may need the complaints description, code, associated disease type and related health unit data. This kind of data access on demand is an interesting feature when accessing large persistent object graphs, so it was implemented in the Health Watcher system.

A common solution to associate the object access strategies with the different kinds of object usages is to provide the access methods with an extra parameter, say an integer, to indicate the desired loading level. So, for example, the `search(int)` method for accessing disease types by their integer code should have an extra parameter to indicate how much disease type information should be accessed. There are two problems with this approach. The first is that the extra parameter has nothing to do with the conceptual service being implemented, so we loose in legibility. The second problem is that this approach requires whoever accesses the objects to indicate this parameter value, generating an indirect dependence with specific persistent data collections, where the extra access methods are implemented.

In order to avoid those problems detected in the pure Java version of the Health Watcher, we defined an aspect to deal with data access on demand. This aspect calls access methods with the extra parameter, but those are not visible to the system services. Those services, for example, call the `search(int)` method for accessing disease types. The aspect intercepts those calls and then calls the access methods with an extra argument indicating the required data loading level. In this way, we preserve the implementation of data access calls without needing an extra parameter, or any other kind of workaround in the user interface and business layers.

Identifying kinds of object usages

The data access on demand aspect first declares the pointcuts that identify where a specific kind of object usage appears. In order to illustrate that, we can use an interface (`PartialLoadingServlet`) to identify the servlets that generate web pages listing partial information about several objects of a class.

```
aspect ParametrizedDataLoading {
  private interface PartialLoadingServlet {}
  declare parents: ServletSearchHealthUnit implements PartialLoadingServlet;
```

For example, a servlet could generate a page with partial information about the various health units registered in the system by just showing the code and the name of the health unit, omitting, for example, the specialties of each health unit. In this case, the `search` method of the `HealthUnit` data collection should adopt a particular kind of object usage, namely partial object loading, which avoids retrieving complex attributes of objects, like relationships to other objects. Therefore, we must define a pointcut matching the execution of those methods:

```
pointcut partialLoadingServlets():
  this(PartialLoadingServlets+) && execution(* do*(..));
```

We should list more servlets in the `declare parents` clause to identify all servlets that should generate a page with partial information.

Applying the adequate loading level

After specifying that the subtypes of `PartialLoadingServlets` adopt a specific kind of object usage, we must, for instance, specify that this kind of usage should be applied when searching `HealthUnit` objects in the associated data collection (`HealthUnitDataRDBMS`):

```
pointcut healthUnitSearchCall(HealthUnitDataRDBMS huData, int code) :
  cflow(partialLoadingServlets()) && target(huData) &&
  call(HealthUnit search(int))    && args(code);
```

The target and the argument of the `search` method are exposed by the pointcut because those values are necessary for redirecting the matched method calls. The `cflow` construct is used to match only method calls that are in the execution flow of the join points matched by the `partialLoadingServlets` pointcut. Therefore, we intercept only `search` method calls that originate from the execution of the methods of `PartialLoadingServlet` and its subtypes. Similar pointcuts should be declared for other access methods called in the same context.

In fact, the previous pointcut does not match any execution of those facade methods if the servlets and the facade are executing in different machines, i.e., the software are distributed. A solution for that is discussed in Section "Distribution aspects for partial loading".

Besides the pointcuts, we must have advice that intercept calls to the access methods and apply the appropriate data loading level. For accesses to health units, we have the following:

```
HealthUnit around(HealthUnitDataRDBMS huData, int code) throws ...:
    searchCall(huData, code) {
  return huData.searchByLevel(code, HealthUnit.PARTIAL_ACCESS);
}
```

We basically replace the `search` method call for a `searchByLevel` call, using the same target and argument. The specified partial loading level corresponds to the level adopted by `partialLoadingServlets`. Using this solution, the persistent data collections should provide methods such as `searchByLevel`, with extra parameters to indicate the loading level. Alternatively, they could provide methods with different names. It is important to clarify that in our approach, the `searchByLevel` method is implemented in the data collection. This should not be seen as an invasive chance, since data collections are actually auxiliary aspects classes. For example the Health Watcher data collections are located in packages like `aspects.dataManagement.dataCollections.rdbms`. In fact, our approach contributes to code modularization, by implementing data collection methods directly in the data collections. Data collection and aspects should not be oblivious from each other, since both are in the "aspect side".

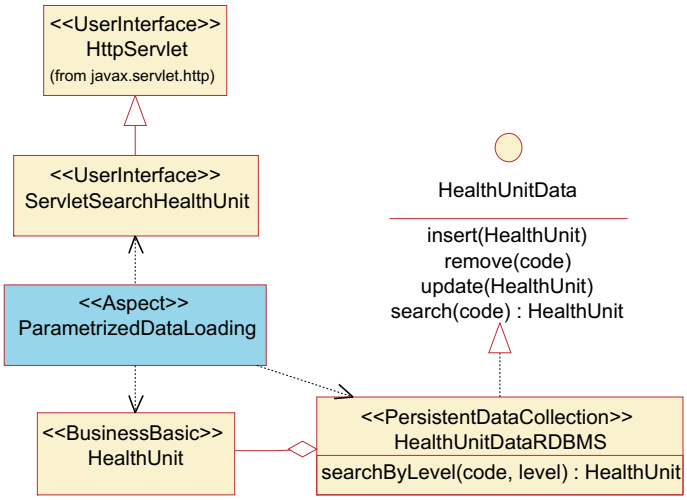

Figure 14. Data access on demand aspects class diagram

This solution modularizes a persistence concern and solves some problems of the pure Java implementation. However, it presents some problems with respect to extensibility and legibility, problems of the original version as well. For example, when modifying the `ServletSearchHealthUnit` code, the programmer must be aware of the advice that intercept that code, otherwise it might try to have access to non-loaded health unit information. It might even be necessary to change the aspect because of changes in the servlet. This shows a strong dependence between the aspects and the Java code, requiring more powerful AspectJ tools as discussed in Section "Transaction control", or more sophisticated pointcut definitions. For the transaction aspects, those tools could be helpful. For the aspect presented in this section, they would be very important.

Our solution to data access on demand also requires more code to be written than in the pure Java version. Fortunately, the extra code follows the same pattern of the pointcuts and advice shown in this section. Code generation tools could easily generate the corresponding code templates. An alternative might be the use of generic aspects, similarly to generic classes in Java.

Data access on demand aspects class diagram

Figure 14 presents a class diagram of the data access on demand aspect and the Health Watcher classes and interfaces the aspects affect or use.

DATA STATE SYNCHRONIZATION CONTROL

The business and presentation layers deal with persistent objects, which contain data that reflect the data stored in the database. Those layers invoke several methods on those objects, changing attribute values in the objects only. Therefore, in order to guarantee object persistence, extra method calls are necessary to synchronize the object data with the database data, reflecting the attribute changes into the database. Similar synchronization calls are also necessary for distribution purposes, as discussed at the end of Section "Client-side distribution aspect". Therefore, this concern is seen as a separate aspect, since it should be used in conjunction in both persistent and distributed versions.

For separating concerns, those layers should not know whether an object is persistent (its state reflects stored data) or not (its state corresponds to nonpersistent data). Therefore, we removed the synchronization calls and implemented a similar functionality in the data state synchronization control aspect. When this aspect is woven with the pure Java code, it introduces the synchronization method calls in the business and presentation code, satisfying both persistence and distribution requirements.

Identifying classes whose objects must be updated

The `UpdateStateControl` aspect defines a tag interface (`SynchronizableObject`) to identify classes whose objects must be updated after being changed.

```
aspect UpdateStateControl percflow(UpdateStateControl.servletService()) {
  private interface SynchronizableObject {
    void synchronizeObject(int updateSource);
  }
```

The aspect uses a `percflow` designator to create an instance of the aspect for each execution flow specified by the pointcut `servletService`:

```
  pointcut servletService() :
    this(HttpServlet) &&
    (execution(void doPost(..)) || execution(void doGet(..)));
```

This avoids undesirable interferences between concurrent executions of Java servlets requests creating an aspect instance per servlet request, instead of sharing a single aspect instance among all requests.

We used the `declare parents` construct to make the class `Complaint` subtype of `SynchronizableObject`.

```
  declare parents: Complaint implements SynchronizableObject;
```

Identifying object updates

The next step is to identify when objects of the previously identified classes are updated in the presentation layer (user interface). Those updates should be identified so that the updated

objects are temporarily stored, in a nonpersistent data structure, and later synchronized with the business layer. We used property-based crosscutting to simplify the specification of the updates in the presentation layer:

```
pointcut remoteUpdate(SynchronizableObject o):
  this(HttpServlet) && target(o) && call(* set*(..));
```

This pointcut matches calls to the `set` methods of persistent objects, the `target` of the calls, but it considers only the calls executed by a servlet, the source (`this`) of the calls. This works well for the Health Watcher system because its user interface is implemented with Java servlets and its classes follow a name convention, actually the Java name convention: methods that change attribute values have names starting with `set`.

The aspect also identifies updates in the business layer. In the Health Watcher architecture, those updates appear in the business collection classes. As we also follow a convention for those classes' names (they all end with `Record`), we can have a general property-based pointcut definition for detecting persistent object updates in the business layer:

```
pointcut localUpdate(SynchronizableObject o):
  this(*Record) && target(o) && call(* set*(..));
```

The name conventions simplify the pointcut definitions, but they are not essential. In fact, more complex pointcuts can be defined when naming conventions are not followed. In general, though, it could be tedious and error prone to list the signatures of the methods that correspond to persistent object updates. Therefore, if no conventions were followed, it would be quite useful to have a code analysis and generation tool that helps the user to identify those methods and generate part of the aspect code.

Capturing updated objects

The aspect identifies the updates and temporarily stores the modified persistent objects in a nonpersistent data structure. This is specified by the following code, which declares an advice and an aspect variable to hold a reference to the data structure:

```
private Set remoteDirtyObjects = new HashSet();
after(SynchronizableObject o) returning: remoteUpdate(o) {
  remoteDirtyObjects.add(o);
}
```

The code for intercepting the updates in the business collection classes is quite similar to this one, so we omit it here.

Synchronizing states

During the execution of a system service, the previous advice captures and stores the updated objects. When the service execution terminates, the aspect can finally introduce the synchronization calls to reflect the updates in the database. This is specified by the following

pointcut and advice, which runs after the execution of the servlet services (`doPost` and `doGet` methods), when there are updated objects that have to be synchronized:

```
pointcut remoteExecution():
  if(UpdateStateControl.aspectOf().hasDirtyObjects()) && servletService();
after() returning: remoteExecution() {
  Iterator it = remoteDirtyObjects.iterator();
  while (it.hasNext()) {
    SynchronizableObject o = (SynchronizableObject) it.next();
    try {
      o.synchronizeObject(UpdateStateControl.REMOTE_UPDATE);
    } finally {
      it.remove();
    }
  }
}
```

The advice basically iterates over the data structure holding the updated objects, synchronizing those objects. We should also define a similar pointcut and advice for synchronizing the objects changed by executing the methods of the business collection classes.

Updating objects

Finally, we should implement the `synchronizeObject` method that is responsible to update objects. This method is introduced into the class using the inter-type declaration mechanism:

```
public void Complaint.synchronizeObject(int updateSource) {
  try {
    if (updateSource == REMOTE_UPDATE) {
      HWFacade facade = HWFacade.getInstance();
      facade.update(this);
    } else {
      ComplaintRecord record = new ComplaintRecord(null);
      record.update(this);
    }
  } catch (Exception ex) {
    throw new SoftException(ex);
  }
}
```

Note once more our exception wrapping approach. The exception handling aspects are responsible to handle them. The record instantiation is actually intercepted by the `DataCollectionCustomization` aspect that initializes the record with the respective data collection to the used storage medium (see Section "Data Collection Customization").

For simplicity, we omitted the declarations for others classes whose objects should be updated if changed. They are, in fact, quite similar to the just illustrated.

Comparing with the pure Java version, this solution is easier to modify since the synchronization concern is completely separated. It is also more concise than the corresponding Java implementation, where the synchronization calls are replicated in several parts of the system. Nevertheless, it also requires some tedious code to be written, so it would be helpful to have a code analysis and generation tool that would help the programmer in implementing this aspect for different systems complying with the same architecture of the Health Watcher system.

Our solution for state synchronization also seems to be less error prone than the Java implementation, where the programmers might usually forget to write some synchronization calls. On the other hand, in the pure Java version the programmer might write the synchronization calls he wants, wherever he wants, benefiting from special optimizations. Some of those optimizations could also be achieved with AspectJ, by implementing different strategies for storing the updated objects and later synchronizing them. In general, though, we expect the AspectJ version to be less efficient than the Java version. In the Health Watcher system, this efficiency loss was insignificant. In more complex systems, dealing with several complex objects, we suspect it might not be worth to separate the synchronization concern using the implementation we proposed. Fortunately, the consequences of not separating this concern are not so drastic. In particular, that would not prevent alternative customizations for the system since the synchronization calls are not middleware dependent.

Generalizing the distribution aspects

As previously shown, the various implementations of the `synchronizeObject` method call facade methods, which indicates that the synchronization originates from updates in the user interface classes. In a distributed version of the system, those calls to the facade methods should be remote. In fact, the distribution aspects should intercept those calls. Unfortunately, as presented in Section "Client-side distribution aspect", the client-side distribution aspect is based on the `facadeCalls` pointcut, which intercepts only facade method calls originating from Java servlets (see the `this(HttpServlet)` constraint in the pointcut). The `synchronizeObject` calls originate from persistent objects.

In order to solve this problem, we had to generalize the definition of the `facadeCalls` pointcut in such a way that it includes new joint points corresponding to the execution of the facade methods called by the `synchronizeObject` method.

```
pointcut facadeLocalCalls():
  (this(HttpServlet) || within(UpdateStateControlPerCflow)) &&
  call(* IRemoteFacade+.*(..)) && !call(static * IRemoteFacade+.*(..));
```

This shows the importance of defining general pointcuts that consider the interception of both the pure Java code and the other aspects code. Moreover, this reinforces the fact that the distribution and persistence concerns are not completely independent. They are semantically dependent. Therefore, careful design activities should have been performed before implementation, avoiding rework, although that was minimal in the reported case. A difficult in that direction is the lack of a proper notion of aspect interface, which would be useful for supporting parallel development.

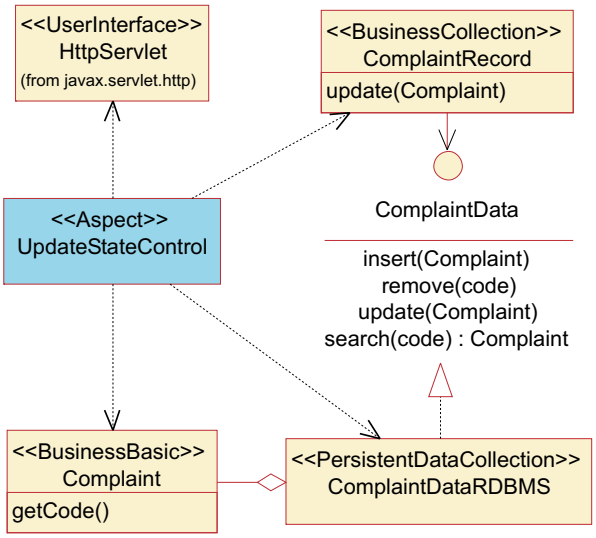

Figure 15. Update state control aspects class diagram

In our previous implementation [1], the persistence aspects depended on the distribution aspects that implement the data synchronization between the server and the clients. However, this concern was factored out, since it crosscuts both distribution and persistence. This allows us to use the distribution aspects together with the state synchronization aspect but without the persistence aspects when they are not necessary, and vice-versa.

Data state synchronization aspects class diagram

Figure 15 presents a class diagram of the data state synchronization control aspect including the Health Watcher classes and interfaces the aspect affects or uses.

EXCEPTION HANDLING ASPECTS

As some of the advice presented so far might raise exceptions that are not handled by the advice themselves, we had to implement auxiliary exception handling aspects. In the Health Watcher system, they basically handle AspectJ's unchecked soft exception, since this is the type of the exceptions raised by the distribution and persistence advice. However, those aspects constitute an exception handling framework that could be used to handle other types of exceptions as well. Although exception handling is a natural crosscutting concern, usually implemented in a scattered form, in our experiment we concentrated on separating distribution and persistence concerns, and simply used the exception handling aspects to handle advice exceptions.

Handling exceptions

We first implemented a general aspect that defines an abstract pointcut for identifying the join points where the (softened) exceptions must be handled:

```
abstract aspect ExceptionHandling {
  abstract pointcut exceptionJoinPoints();
  Object around(): exceptionJoinPoints() {
    Object o = null;
    try {
      o = proceed();
    } catch (SoftException ex) {
      this.exceptionHandling(ex);
    }
  }
  protected abstract void exceptionHandling(SoftException ex);
}
```

The aspect also defines an `around` advice that catches `SoftException` objects in the specified join points. This advice specifies that the exception should be handled by the `exceptionHandling` method, which is also declared as abstract by the aspect.

Handling exceptions with servlets

As the user interface classes of the Health Watcher system are Java servlets, we extended the general exception handling aspect with behavior useful for handling exceptions with servlets. The servlets are basically used to properly notify the user that something went wrong, and maybe suggest some specific actions she should take. In order to do that, the aspect code must have access to `PrintWriter` objects, which are used by servlets to write responses back to the service requester. The following aspect does that by defining a pointcut that identifies the join points where a `PrintWriter` object is obtained through the response object:

```
abstract aspect ServletsExceptionHandling extends ExceptionHandling
    percflow(ServletsExceptionHandling.clientService()) {
  pointcut printWriterCreation():
    target(HttpServletResponse) && call(PrintWriter getWriter());
```

It also declares an `after returning` advice, which actually get and store the `PrintWriter` object returned by the `getWriter` method call:

```
  private PrintWriter printWriter;
  after() returning (PrintWriter out): printWriterCreation() {
    printWriter = out;
  }
```

This aspect uses a `percflow` aspect association to create an instance of the aspect for each entrance to the control flow of the join points defined the `clientService` pointcut (execution

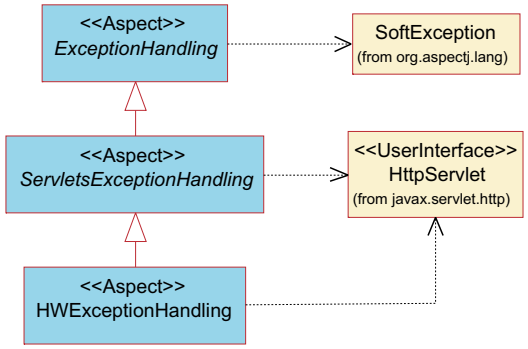

Figure 16. Exception handling aspects class diagram

of the `do*` servlets methods). This is necessary because one `PrintWriter` object is created for each request received by a servlet. So when handling exceptions we should make sure that we use the right `PrintWriter` to notify the user.

This aspect also provides the concrete definition of the `exceptionHandling` method. It basically accesses the exceptions wrapped as soft exceptions and properly notifies the user through the `PrintWriter` object.

In order to be reusable, the previous aspect is abstract and does not provide a concrete pointcut to identify the join points where the exceptions must be caught. Specific aspects should do this. In the Health Watcher system, we defined such a specific aspect (`HWExceptionHandling`) for identifying default exception handling join points: the service methods of the servlets, meaning that the default handling behavior is to notify the user. If other aspects need to define specific exception handling behavior, they must define a specialization of the aspect `ExceptionHandling`, providing the handling behavior and the join points to catch the exceptions.

Exception handling aspects class diagram

Figure 16 presents a class diagram of the exception handling aspects and the Health Watcher classes and interfaces the aspects affect or use.

INTERFERENCES BETWEEN ASPECTS

When performing the experiment we identified interferences between some aspects. Since the distribution aspects change the computing model by distributing the processing in different machines, they interfered with other aspects. The identified interferences were:

- Persistence aspects might call facade methods that should be redirected to the remote facade instance, when generating the distributed version.
- Information about the execution context in the client-side is not available in the server side. For example, the `cflow` designator cannot identify join points in the server-side that was originated from the GUI control flow, because this flow information is not transmitted from the client-side to the server-side.

We believe that the interferences of the distribution aspects are natural, since they modify the programming model when distributing part of the processing. Therefore, the distribution aspect must be aware of these interferences providing solution to them.

The `PersistentDistributedUpdateStateControl` aspect declares a compile-time warning that identifies calls to the facade class. The programmer should investigate these calls and, if necessary, write the proper pointcut and advice to redirect them to the facade's remote instance, similarly to what the `ClientSideDistribution` does. At the moment, this aspect affects the`UpdateStateControl` aspect, which calls facade methods. In Section "Data state synchronization control" the distribution aspect is generalized to affect the `UpdateStateControl` aspect.

Distribution aspects for partial loading

The `ParametrizedDataLoading` aspect uses information about the servlets execution flow in order to determine if a `search` method call should retrieve an object with complete or partial information. However, the current implementation of the AspectJ `cflow` designator cannot track the execution flow if the flow executes in distinct machines. Therefore, by distributing the servlets execution to another machine, or another JVM (Java virtual machine), the `cflow` designators in the aspects do not track the execution flow from the distributed servlets, not reaching the join points defined by the `ParametrizedDataLoading` aspect.

Figure 17 depicts the problem mentioned above. Consider the `PartialLoading` aspect that uses information about the servlet execution flow in order to affect a facade's method. When a specific servlet `X` calls a facade method (a) the aspect should affect its execution (b) changing its behavior. However, if the servlet and facade objects are distributed in different machines, the `cflow` designator used by the aspect does not match the servlet execution, since it does not occur in the same machine the aspect is running.

Our solution adds a new method in the facade class in order to identify calls from servlet `X` to method `m` and redirects the original servlet call to the new method (c). Now, the aspect should use the `cflow` information from the added method (d) in order to affect `m`'s execution (e).

This shows that the current implementation of the AspectJ `cflow` designator should be modified in order to support remote execution at least using RMI, since AspectJ is an extension of Java and RMI is the Java solution for distribution. There are proposals for implementing such kind of construction elsewhere [26]. As AspectJ does not provide such support, we develop a solution. We define new aspects that must be woven to the system if every time the `ParametrizedDataLoading` aspect and the distribution aspects are woven.

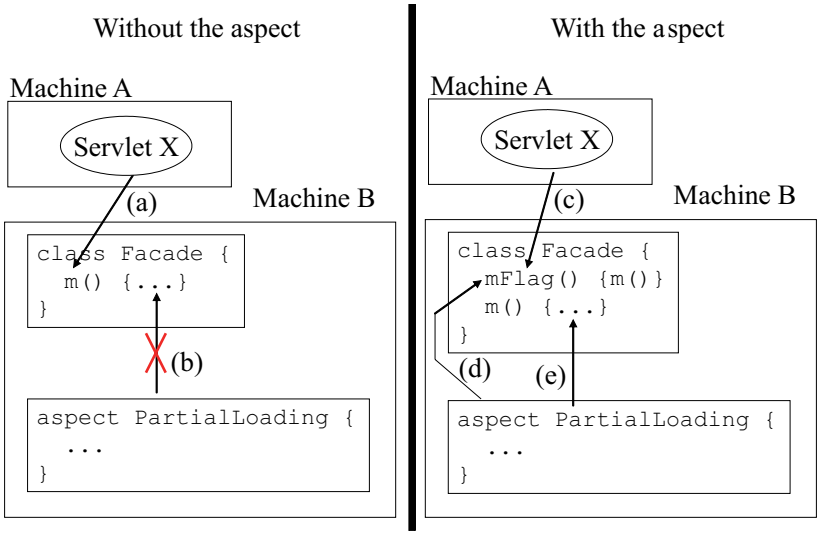

Figure 17. Interference problem and solution.

Our solution has actually two additional aspects that affect the server and the client side. The aspects are responsible to add some explicit information in the server-side, in order to identify that an execution was made from a specific servlet and should be handled differently, as depicted by Figure 17.

The `FacadeDistributedParametrizedDataLoading` aspect adds new methods to the facade class and to the remote interface to replace the servlets `cflow` information.

```
aspect FacadeDistributedParametrizedDataLoading {
  abstract IteratorHW IRemoteFacade.flagGetHealthUnitList()
      throws ObjectNotFoundException, RemoteException;
  IteratorHW HWFacade.flagGetHealthUnitList()
      throws ObjectNotFoundException, RemoteException {
    return this.getHealthUnitList();
  ...
}
```

Note that the added method has the same name of one of the facade methods but with a prefix (`flag`). Whenever this method is called, the data collection `search` method should execute a partial loading. In fact, this aspect should have more methods for dealing with additional use of the `cflow` designator. The next declaration defines a pointcut to identify calls to the added method. The `cflow` designator is used by another pointcut to identify calls to the data collection `search` method exposing the data collection and the argument of the method call.

```
    pointcut healthUnitFetchDataFacadeIntroducedMethod():
      this(HWFacade) &&
      execution(IteratorHW flagGetHealthUnitList());
    pointcut flatLevelOfAccess(HealthUnitDataRDBMS huData, int code):
     target(huData) && cflow(healthUnitFetchDataFacadeIntroducedMethod()) &&
      args(code)     && call(HealthUnit search(int));
```

At last the aspect defines an `around` advice that redirects calls from the data collection `search` method to its `searchByLevel` method, just like the `PartialLoadingServlets` advice. The method redirection is performed based on the `cflow` of the flagged method since there is no explicit information available about the servlets execution flow.

```
    HealthUnit around(HealthUnitDataRDBMS huData, int code)
     throws ObjectNotFoundException : flatLevelOfAccess(huData, code) {
      return huData.searchByLevel(code, HealthUnit.SHALLOW_ACCESS);
    }
  }
```

Similar code should be added into this aspect in order to identify new execution flows originated from servlets.

Now in order to flag partial loading we have to redirect the calls from the servlets affected by the `PartialLoadingServlets` to the added flag methods. This is our solution to identify what method in the server-side was called by servlets that request partial loaded objects.

The `ClientDistributedParametrizedDataLoading` aspect identifies calls to the servlets methods that can manipulate partial loaded objects.

```
  aspect ClientDistributedParametrizedDataLoading {
    pointcut getHealthUnitListCall():
      this(ServletSearchHealthUnit) &&  target(IRemoteFacade+) &&
      call(IteratorHW getHealthUnitList());
```

Note that the method identified by this pointcut is the one that has a corresponding flagged one in the facade. The next step is to define an `around` advice to redirect the servlet call from the original method definition to the flag method added by the previous aspect in order to allow the server side partial loading aspect to identify this call as a partial loading candidate.

```
    IteratorHW around() throws ObjectNotFoundException:
        getHealthUnitListCall() {
      try {
        IRemoteFacade healthWatcher = (IRemoteFacade)
          HWServerSide.aspectOf().getRemoteFacade();
        return healthWatcher.flagGetHealthUnitList();
      } catch(RemoteException ex) {
        throw new SoftException(ex);
      }
    }
  }
```

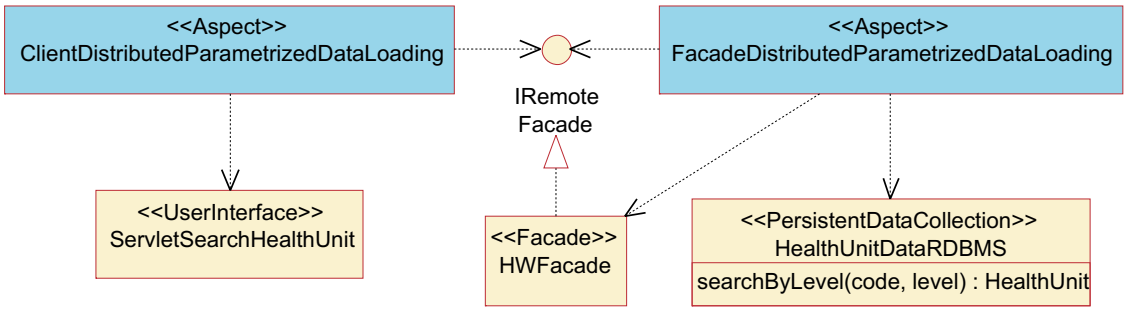

Figure 18. Interferences between aspects

The advice retrieves the remote instance from the `HWServerSide` aspect through the `getRemoteFacade` method. Once more we wrap the concern-specific exception into a `SoftException` to be handled by specific exception handling aspects.

Theses interferences show that an aspect that implements a crosscutting concern should be aware if others crosscutting concerns can interfere with each other.

Figure 18 shows a class diagram that presents two aspects that should be woven to the system when generating a persistence and distribution version of it. The diagram also presents the Health Watcher classes and the affected aspects.

Restructuring experience summary

Our experience on restructuring an object-oriented software to an aspect-oriented one, helped to validate the use of AspectJ for implementing several persistence and distribution concerns in the kind of application considered here. Moreover, we notice that the implementation of those concerns brings significant advantages in comparison with the corresponding pure Java implementation.

However, we have identified a few drawbacks in the AspectJ language and suggested some minor modifications that could significantly improve implementations similar to the one discussed here. In addition, another drawback relies on the fact that the definition of a pointcut identifies (by using methods signatures, class names, etc.) specific points of a given system, the aspects become specific for that system, or for systems adopting the same naming conventions, decreasing reuse possibilities.

Table I summarizes the implemented aspects given an overview of the aspects, and of what concerns the aspect deals with. The table also shows which aspects might be reused in other developments. Auxiliary types used by the aspects of our experiment are presented in Table II, also including the concern related to each type, which type can be reused, and if the type is a class or an interface.

Table I. Aspects summary.

Aspect name	Related concern	Is reusable?
`ServerSide`	distribution	yes
`HWServerSide`	distribution	no
`ClientSide`	distribution	yes
`HWClientSide`	distribution	no
`PersistenceControl`	persistent data management	yes
`HWPersistenceControl`	persistent data management	no
`TransactionControl`	persistent data management	yes
`HWTransactionControl`	persistent data management	no
`DataCollectionCustomization`	data management	yes
`HWDataCollectionCustomization`	data management	no
`PersistentDataCollection`	persistent data management	no
`NonPersistentDataCollection`	nonpersistent data management	no
`ParametrizedDataLoading`	persistent data management	no
`UpdateStateControl`	distribution and persistence	no
`ExceptionHandling`	exception handling	yes
`ServletsExceptionHandling`	exception handling	yes
`HWExceptionHandling`	exception handling	no
`FacadeDistributedParametrizedDataLoading`	distribution and persistence	no
`ClientDistributedParametrizedDataLoading`	distribution and persistence	no

Table II. Auxiliary types summary.

Auxiliary type name	Related concern	Is reusable?	Observation
`IPersistenceMechanism`	persistent data management	yes	interface
`PersistenceMechanismRDMS`	persistent data management	yes	class
`ITransactionalMethods`	persistent data management	no	interface
`SystemRecord`	data management	no	abstract class
Persistent data collections	persistent data management	no	classes
Nonpersistent data collections	nonpersistent data management	no	classes
IRemoteFacade	distribution	no	interface

RELATED WORK

This section presents works related to aspect-oriented programming, in particular, these works are related to the implementation of persistence or distribution concerns.

Persistence as an Aspect

Another work [16] that discusses how to implement persistence with aspects defines reusable aspects developed into a framework. Similarly to our work, this one has similar conclusions such as persistence can be modularized using aspect-oriented programming, some aspects can be reused, and systems cannot be developed unaware of the need for data storage. This means that programmers can only be partially oblivious to the persistence nature of the data. In fact, our work considers data management as an aspect, and considers persistence as one kind of data management. We use such approach to support the progressive approach [7, 27, 28], where persistence and distribution are not initially considered in the implementation activities, but are gradually introduced, preserving the system's functional requirements. This progressive approach helps to decrease the impact caused by requirements changes during development, since a great part of the changes might occur before the final (persistent and distributed) version of the system.

The work shows that data storage and update — the insertion of an object when it is created and its update when it is changed — can be modularized and the system can be unaware of these features (obliviousness). On the other hand, the system should be aware of data retrieval and deletion, since systems have to explicitly obtain or delete persistent objects from an external source, which forbids programmers to be completely oblivious of persistence. Therefore, in their solution the system has to use a specific interface (`PersistenceData`) to retrieve an object and a specific class (`PersistentRoot`) to delete it. This solution is quite similar to ours, however, we use a business-data interface that provides methods to insert, update, retrieve, and delete the objects. By using an interface, data management services, except transactions, can be implemented as Java collections, files, relational databases, or object-oriented databases. This adaptability is one of our goals, and is not supported by the related work that proposes solutions specific to relational databases and briefly discusses on how the framework could be adapted to suit other database technology, like object-oriented databases.

An interesting feature provided by this work is a SQL translation aspect that translates an object-oriented model to a relational database (object-to-relational mapping) schema. It uses reflection (Java and AspectJ APIs) and generates SQL statements to access the database, whereas our approach hard-codes the SQL statements in the implementation of the data management aspects. In fact, our approach also uses reflection, however, with a different goal. We use reflection in the distribution aspects to generalize the redirection of local facade methods calls to the remote instance, avoiding the definition of an advice for each facade method. Our use of reflection can be avoided in future implementations of AspectJ, as explained in the Section Client-side distribution aspect.

Concurrency and Transactions

A related work is another AspectJ implementation of transactions, which was independently developed in the context of the OPTIMA framework for controlling concurrency and failures with transactions [21]. This implementation does not consider distribution and persistence

concerns as we do here, but deals mostly with transactions for implementing concurrency concerns. Nevertheless, there are similarities with our approach, so we discuss it in detail here.

The authors of the OPTIMA approach first analyze the adequacy of AspectJ for completely abstracting transaction concerns in such a way that transactional behavior can be introduced in an automatic and transparent way to existing non-transactional systems. They conclude that AspectJ is not suitable for this purpose. We have not tried to analyze that in our experiment since we believe that the main aim of AspectJ, and aspect-oriented programming in general, is to modularize crosscutting concerns, not to make them completely transparent. For some situations, this transparency could be achieved by proper tools that would generate AspectJ code, but not by the language itself.

The kind of transparency sought by the authors should not be confused with obliviousness, which is supported by AspectJ and allows a system programmer to not worry about inserting hooks in the code so that it is later affected by the aspects. This does not mean that the system programmer should not be aware of the aspects that intercept the system code. Likewise, the aspect programmer should be aware of the code that his aspect intercepts. In this sense, there might be strong dependencies between AspectJ modules, reducing some of the benefits of modularity. In spite of that, there are still important benefits that can be achieved. Moreover, we believe that this problem could be minimized by more powerful AspectJ tools providing multiple views, and associated operations, of the system modules. Appropriate notions of aspect interfaces should also be developed.

AspectJ's ability to separate transactional interfaces (begin, abort, commit), defining aspects to invoke the transactional methods whenever necessary, has also been analyzed by the same authors. Their implementation is similar to what we present, at the beginning of Section "Transaction control", but they do not explore the variations that we present at the end of the same section. Those variations can actually avoid the performance problems they mentioned. They also faced the same problem we had with the impossibility of adding an exception to a method `throws` clause. However, our transaction control approach avoids this problem, which actually appears here when dealing with the distribution concerns (see Section "Server-side distribution aspect").

When separating the transactional interfaces, they also complain about the strong dependencies mentioned before, suggesting that AspectJ might not be useful for this task either. In the transactions case, we argue that the dependencies do not bring major problems in practice. This is the case because changes in the transaction aspects are minimal and usually do not affect the pure Java code, whereas changes in the Java code have only a very small impact on the aspects, assuming that it has been established that any exception that is thrown and not handled by a transactional method aborts the transaction. In fact, powerful AspectJ tools for dealing with dependencies would be needed much more for the data access on demand aspects (see Section "Data access on demand") than for the transaction aspects. It seems that our AspectJ implementation of transactions can usually have significant advantages over pure Java implementations (see Section "Transaction control"). That is certainly the case for systems such as the Health Watcher.

Finally, the OPTIMA experience tries to separate transaction mechanisms, supporting different customizations for transaction and concurrency control. They conclude that AspectJ is useful for that. Although we have not implemented much transaction customization, we had

the same positive experience using aspects to customize data management and distribution services.

EJB

Another related work is The Enterprise JavaBeans (EJB) architecture [14, 15]. EJB supports the development of distributed systems providing services such as transactions, database connectivity, and multi-user safety, which is also supported by our method, however, EJB implements those aspects in a transparent way.

The EJB transparency makes easy to write systems in the sense that developers does not have to understand low-level transaction and state management details, multi-threading, connection pooling, or other complex APIs. On the other hand, developers cannot write their owns algorithms looking for performance improving, which happens in our approach, where the developers have to write their own code for implement transactions, database connectivity, distribution, and multi-user safety, dealing with different APIs.

When using EJB to implement persistence and distribution, usually in a non-progressive way, another problem is the deployment time, which might be very high. To fix errors — including functional and persistence errors — a lot of time might be wasted by compiling the code and them deploying the system into the application server.

When performing the experiment, we did not aim in providing implementation transparency, but in improving software modularity. This modularity allows the use of a progressive implementation approach, previously mentioned, where functional errors are early fixed and will not be mixed with persistence and distribution errors.

Despite these differences between EJB and our approach, one possible implementation of our aspects can use the EJB architecture. Therefore, we would have aspects to implement persistence and distribution using the EJB architecture. This would lead to better modularity, by using AOP, and it would also achieve implementation transparency, provided by EJB. Our approach would also allow using EJB in a progressive approach. Actually, another way to achieve transparency with AspectJ, keeping the others benefits of AOP, is using code generation tools that would generate aspects, similar to EJB that generates classes.

Others related works

JBoss [29] is an Java aspect-oriented framework that provides a set of aspects to implement several concerns, such as transactions and distribution. Another related work in the same line of JBoss is JAC [30], a framework for building aspect-oriented distributed applications in Java. Both works have a great differential to ours by allowing dynamic weaving, which means aspects can be weaving and unweaving at runtime. Our approach is tailored to AspectJ language, whose current's version does not allow dynamic weaving. Despite our approach being tailored to the RMI and JDBC APIs, our idea is to provide separation of concerns in general, modularizing persistence, exception handling, transactions, distribution, and so on. Therefore, those related works might be seen as complementary works, and other studies might be performed to investigate the use of JBoss and JAC with our approach, similar to what should be done with EJB. In addition, with our approach we also have a fine-grained

control over the aspects implementation, avoiding the overhead that might exist in general solutions that are hard to optimize.

The implementation of distribution and persistence concerns in pure object-oriented systems was explored elsewhere, leading to specific design patterns [9, 10]. Those patterns support the progressive implementation of distribution and persistence code in an object-oriented system. Despite having similar goals, this approach does not achieve the same level of separation of concerns; for instance, the distribution and persistence exception handling are tangled with user interface and business code. There is also scattered code over several units, such as in the serialization mechanism implementation, and the identification of what objects should be made persistent, by using class inheritance. In fact, this was our motivation to study AOP and AspectJ in order to improve the modularity of those concerns.

The need for higher adaptability and configurability of middleware is discussed elsewhere [31, 32]. This work discusses the problem of middleware architectures that vary from general features in order to support several domains, to optimizations supporting a particular domain with specialized runtime requirements. The work describes a case study where AOP is used to improve modularization of the CORBA [33] middleware, by factoring out aspects that were identified in CORBA. The identified aspects were implemented in AspectJ to increase the middleware configurability, since the aspects can be chosen at compile-time. This work is another example where AOP and AspectJ are effectively used to modularized crosscutting concerns. However, this work differs from ours because it modularizes concerns of a specific middleware (CORBA), making it customizable, whereas our approach modularizes concerns of a system. Their approach is complementary to ours. We could use their improved version of CORBA to implement another of the distribution aspects. In fact, our approach allows changing the middleware implementation where their approach does not aim in doing that.

An experience to evaluate the suitability of AspectJ for modularizing crosscutting concerns in a middleware product line is related in another work [34]. The motivation was to use AOP to target multiple runtime environments with a single code base. Examples of addressed concerns are tracing and logging, event reporting, error handling, and performance monitoring. The work also discusses the impact of AOP on architectural quality. They derived conclusions about AspectJ similar to our experiment, one of the issues is pointcut fragility, which is the pointcut dependence on the system code. This demands refactoring [35] tools to consider this dependence in order to avoid refactorings breaking the aspect code. It is also considered the need for improvements of the AspectJ compiler, mainly error messages, which could give more support for the programmers. The main conclusion is that AspectJ can be used to modularize many important crosscutting problems, however, they did not report any interference problems as we did, probably because of the different nature of our crosscutting concerns (data management and communication) in contrast to their concerns (tracing and logging, event reporting, error handling, and performance monitoring).

Regarding distribution and aspects, another work [36] proposes a tool for supporting aspect-oriented distributed programming. They have the same goal of implementing distribution without changing the core system code. However, this work uses a specific language to state what objects are located in a host, modifying Java bytecode using Java reflection. In contrast, our approach uses a general-purpose language and does not worry in specify where objects

are located, but uses an API to implement remote methods call from the user interface to the system facade [8].

The need for Quality of Service (QoS) aspects in distributed programs are discussed elsewhere [37]. This work addresses QoS issues, such as, transmission errors, dynamic bandwidth fluctuation, overload situations, partial failures, etc. They define an IDL extension to allow QoS constructions and exemplify its use with CORBA. We agree that some of these issues should be considered when implementing distribution aspects. However, the approach in our experiment was to restructure an OO system to an AOP version and to investigate some issues of such restructuring. For example, how one aspect affects and is affected by others, what are the challengers in AOP, what kind of modification should be made to the AspectJ language to allow a better separation and composition of concerns, when a progressive approach is better than a non-progressive one[§], and so on. As a future work, we should care about QoS aspects to improve our framework allowing QoS management.

Techniques such as implicit context [38] and CaesarJ [39] might be considered as alternatives for solving the remote facade issue, where local calls in the monolithic version of the software should be redirected to a remote instance to provide distribution behavior.

CONCLUSION

We discussed our experience on restructuring a simple, but real and non-trivial, web-based information system with AspectJ. In the new version of the system, the implementation of the distribution and persistence concerns are completely separated from each other and from the business and user interface concerns. Among other benefits, this might make easier changing the distribution middleware or the persistence mechanism without affecting the implementation of the other concerns. In fact, experimental studies or case studies should be carried out to evaluate the effective effort difference in maintaining an object-oriented versus an aspect-oriented version of the software.

The main contribution of our experience is to validate the use of AspectJ for implementing several persistence and distribution concerns in the kind of application considered here. Moreover, we notice that the implementation of those concerns brings significant advantages in comparison with the corresponding pure Java implementation. The only exception is the data access on demand concern; its implementation also has some disadvantages that could only be minimized with more powerful AspectJ tools supporting aspect interfaces and multiple views of the system modules, which would help programmers deal with strong dependencies between the aspects and the pure Java code. In fact, the need for this kind of tool was reported elsewhere [40]. In our experiment, we considered only basic remote communication concerns, not implementing distribution issues such as caching, fault tolerance, and automatic object deployment for load balancing. However, we believe that those issues could be implemented essentially using the presented approach, revealing no further conclusions about the use of

[§]The analysis about the benefits and drawbacks of using a progressive approach will be reported in a PhD thesis and in papers to be written.

AspectJ. Again, experimental studies or case studies should be performed to demonstrate the effectiveness of our approach when applied to other software.

In spite of our successful experience with AspectJ, we have identified a few drawbacks in the language and suggested some minor modifications that could significantly improve implementations similar to the one discussed here. Furthermore, we noticed that AspectJ's powerful constructs must be used with caution, since they might have undesirable and unintended side effects. Moreover, as the definition of a pointcut identifies (by using methods signatures, class names, etc.) specific points of a given system, the aspects become specific for that system, or for systems adopting the same naming conventions, decreasing reuse possibilities. This suggests that we should either support aspect parameterization [41, 42] or have the support of code generation tools [7, 17] when developing with AspectJ. The need for those tools has actually been noticed on several occasions during our experience. AspectJ's development environment is also quite immature and needs considerable improvements in compilation time and bytecode size. It is also true that they have been continuously improved. Other discussions on AspectJ benefits and drawbacks from other point of views, like language design and implementation, can be found elsewhere [43, 44].

The distribution and persistence concerns considered here can be implemented separately. However, we noticed that the exception handling and state synchronization aspects are actually necessary for both distribution and persistence aspects. Moreover, the distribution and persistence aspects can be used separately, but if they are used together then some distribution advice must intercept the execution of some persistence advice. In addition, we showed that the distribution aspects affect the persistence aspects by breaking how some of then work. Therefore, additional aspects were implemented to solve those problems when using persistence in a distributed environment. This shows that the persistence and distribution aspects are not completely independent. Therefore, careful design activities are also important for aspect-oriented programming. This is the only way we can detect in advance intersections, dependencies and conflicts among different aspects. Consequently, we can avoid serious development problems and better plan the reuse and parallel development of different aspects. This need for design activities does not seem to have been considered in [21], leading to some of the problems discussed there. It has been noticed before that distribution issues should not be handled only at implementation or deployment time [45].

Some of the aspects implemented in our experiment are abstract and constitute a simple aspect framework. They can be extended for implementing persistence and distribution in other applications that comply with the architecture of the health complaint system, a layer architecture used for developing web-based information systems. Although specific, this architecture has been used for developing many Java systems: a system for managing client information and mobile telephone services configuration; a system for performing online exams, helping students to evaluate their knowledge before the real exams; a complex point of sale system, and many others.

The other aspects are application specific and therefore have different implementations for different applications. Nevertheless, we suggest that different implementations might follow a common aspect pattern, having aspects with the same structure. Elsewhere [46], we document such an aspect pattern to implement distribution aspects in an object-oriented application. The

pattern structures can be encoded in code generation tools [17] and automatically generated for different applications, increasing productivity.

Based on the framework and the patterns, we can derive architecture specific guidelines that provide practical advice for both restructuring and implementing certain kinds of persistent and distributed applications with AspectJ. However, much more experience with those guidelines is needed before they could be used by a tool for partially automating the refactoring of pure Java systems similar to the one considered here. This tool could do a lot of work mainly because the program structure and the guidelines are tailored to a specific architecture.

ACKNOWLEDGEMENTS

We would like to thanks CAPES and CNPq, brazilian research agencies, which partially support this work, the members of the Software Productivity Group (SPG), for their help during the discussions on this research, and the anonymous reviewers for they careful and important comments for improving the quality of this work.

REFERENCES

1. Sérgio Soares, Eduardo Laureano, and Paulo Borba. Implementing Distribution and Persistence Aspects with AspectJ. In *Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications, OOPSLA'02*, pages 174–190. ACM Press, November 2002. Also appeared in ACM SIGPLAN Notices 37(11).
2. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. Getting Started with AspectJ. *Communications of the ACM*, 44(10):59–65, October 2001.
3. Tzilla Elrad, Robert Filman, and Atef Bader. Aspect–Oriented Programming. *Communications of the ACM*, 44(10):29–32, October 2001.
4. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc, and John Irwin. Aspect-Oriented Programming. In *European Conference on Object-Oriented Programming, ECOOP'97*, LNCS 1241, pages 220–242, Finland, June 1997. Springer-Verlag.
5. James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, second edition, 2000.
6. Sun Microsystems. Java Remote Method Invocation (RMI). At http://java.sun.com/products/jdk/1.2/docs/guide/rmi, 2001.
7. Sérgio Soares. *An Aspect-Oriented Implementation Method*. PhD thesis, Informatics Center, Federal University of Pernambuco — CIn-UFPE — Brazil, October 2004. Available at http://www.cin.ufpe.br/-spg/GenteAreaThesis.
8. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
9. Vander Alves and Paulo Borba. Distributed Adapters Pattern: A Design Pattern for Object-Oriented Distributed Applications. In *First Latin American Conference on Pattern Languages of Programming — SugarLoafPLoP*, Rio de Janeiro, Brazil, October 2001. Published in UERJ Magazine: Special Issue on Software Patterns.
10. Tiago Massoni, Vander Alves, Sérgio Soares, and Paulo Borba. PDC: Persistent Data Collections pattern. In *First Latin American Conference on Pattern Languages of Programming — SugarLoafPLoP.*, pages 311–326, Rio de Janeiro, Brazil, October 2001. Published in University of São Paulo Magazine — ICMC, 2002.
11. Ian S. Graham. *The HTML Sourcebook*. Wiley Computer Publishing, second edition, 1996.
12. David Flanagan. *JavaScript The Definitive Guide*. O'Reilly & Associates, Inc., second edition, 1997.
13. Jason Hunter and Willian Crawford. *Java Servlet Programming*. O'Reilly & Associates, Inc., first edition, 1998.

14. Richard Monson-Haefel. *Enterprise JavaBeans*. Oreilly, second edition, 2000.
15. Sun Microsystems. The Enterprise JavaBeans Specification Version 2.1, August 2002. At http://java.-sun.com/products/ejb.
16. Awais Rashid and Ruzanna Chitchyan. Persistence as an aspect. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD'03)*, pages 120–129. ACM Press, March 2003.
17. Marcelo d'Amorim, Clóvis Nogueira, Gustavo Santos, Adeline Souza, and Paulo Borba. Integrating Code Generation and Refactoring. In *Workshop on Generative Programming, ECOOP'02*, Málaga, Spain, June 2002.
18. Grady Booch, Ivar Jacobson, and James Rumbaugh. *Unified Modeling Language — User's Guide*. Addison-Wesley, 1999.
19. Gail C. Murphy, Robert J. Walker, Elisa L.A. Baniassad, Martin P. Robillard, Albert Lai, and Milk A. Kersten. Does aspect–oriented programming work? *Communications of the ACM*, 44(10):75–77, October 2001.
20. Fernando Castor, Kellen Oliveira, Adeline Souza, Gustavo Santos, and Paulo Borba. JaTS: A Java transformation system. In *XV Brazilian Symposium on Software Engineering*, pages 374–379, Rio de Janeiro, Brazil, October 2001.
21. Jörg Kienzle and Rachid Guerraoui. AOP: Does it Make Sense? The Case of Concurrency and Failures. In *European Conference on Object-Oriented programming, ECOOP'02*, LNCS 2374, pages 37–61, Málaga, Spain, June 2002. Springer–Verlag.
22. Ramez Elmasri and Shamkant Navathe. *Fundamentals of Database Systems*. Addison–Wesley, second edition, 1994.
23. Object Technology International Inc. Eclipse Platform Technical Overview. White Paper. At http://-www.eclipse.org/, 2001.
24. AspectJ development tools subproject. At http://www.eclipse.org/ajdt, 2005.
25. AspectJ Team. The AspectJ Programming Guide. At http://eclipse.org/aspectj, 2003.
26. Michiaki Tatsubori Muga Nishizawa, Shigeru Chiba. Remote pointcut: a language construct for distributed aop. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development, AOSD 2004*, pages 7–15, March 2004.
27. Sérgio Soares and Paulo Borba. PIP: Progressive Implementation Pattern. In Michael Gnatz, Frank Marschall, Gerhard Popp, Andreas Rausch, Maura Rodenberg-Ruiz, and Wolfgang Schwerin, editors, *Proceedings of the 1st Workshop on Software Development Patterns (SDPP'02)*, Technical Report TUM-I0213, Munich University of Technology, Munich 12/2003, November 2002.
28. Paulo Borba, Saulo Araújo, Hednilson Bezerra, Marconi Lima, and Sérgio Soares. Progressive Implementation of Distributed Java Applications. In *Engineering Distributed Objects Workshop, ACM International Conference on Software Engineering*, pages 40–47, Los Angeles, EUA, 17th–18th May 1999.
29. Jboss aspect oriented programming webpage. Avalible at http://www.jboss.org/products/aop, 2005.
30. Renaud Pawlak, Lionel Seinturier, Laurence Duchien, Gérard Florin, Fabrice Legond-Aubry, and Laurent Martelli. JAC: an aspect-based distributed dynamic framework. *Software — Practice and Experience*, 34:1119–1148, 2004.
31. Charles Zhang and Hans-Arno Jacobsen. Resolving feature convolution in middleware systems. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 188–205, New York, NY, USA, 2004. ACM Press.
32. Charles Zhang and Hans-Arno. Jacobsen. Quantifying aspects in middleware platforms. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD'03)*, pages 130–139. ACM Press, March 2003.
33. Robert Orfali and Dan Harkey. *Client/Server Programming with Java and CORBA*. Wiley, 1998.
34. Ron Bodkin, Adrian Colyer, and Jim Hugunin. Applying aop for middleware platform independence. In *Practitioner Report of the 2nd International Conference on Aspect-Oriented Software Development (AOSD'03)*, March 2003.
35. Martin Fowler et al. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
36. Michiaki Tatsubori. Separation of Distribution Concerns in Distributed Java Programming. In *OOPSLA'01, Doctoral Symposium*, Tampa FL, 2001.
37. C. Becker and K. Geihs. Quality of service - aspects of distributed programs. In *International Workshop on Aspect Oriented Programming (ICSE 1998)*, February 1998.
38. Robert Walker and Gail Murphy. Implicit context: easing software evolution and reuse. In *SIGSOFT '00/FSE-8: Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software*

engineering, pages 69–78. ACM Press, 2000.
39. Mira Mezini and Klaus Ostermann. Variability management with feature-oriented programming and aspects. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 127–136. ACM Press, 2004.
40. Robert Filman and Daniel P. Friedman. Aspect–Oriented Programming is Quantification and Obliviousness. In *Workshop on Advanced Separation of Concerns, OOPSLA'00*, 2000.
41. Neil Loughran and Awais Rashid. Framed aspects: Supporting variability and configurability for aop. In *Proceeding of the 8th International Conference on Software Reuse: Methods, Techniques and Tools, ICSR 2004*, pages 127–140. IEEE CS Press, July 2004.
42. Stefan Hanenberg and Rainer Unland. Parametric introductions. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD'03)*, pages 80–89. ACM Press, March 2003.
43. Elisa L. A. Baniassad, Gail C. Murphy, Christa Schwanninger, and Michael Kircher. Managing crosscutting concerns during software evolution tasks: an inquisitive study. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 120–126. ACM Press, 2002.
44. Yvonne Coady and Gregor Kiczales. Back to the future: a retroactive study of aspect evolution in operating system code. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 50–59. ACM Press, 2003.
45. Jim Waldo, Samuel C. Kendall, Ann Wollrath, and Geoff Wyant. A Note on Distributed Computing. Technical Report TR-94-29, Sun Microsystems Laboratories, Inc., November 1994.
46. Sérgio Soares and Paulo Borba. PaDA: A Pattern for Distribution Aspects. In *Second Latin American Conference on Pattern Languages of Programming — SugarLoafPLoP 2002.*, pages 87–99, Itaipava, Rio de Janeiro, Brazil, August 2002. Published in University of São Paulo Magazine — ICMC.