Implementing Distribution and Persistence Aspects with AspectJ

Sérgio Soares*
scbs@cin.ufpe.br

Eduardo Laureano
eagcl@cin.ufpe.br

Paulo Borba
phmb@cin.ufpe.br

Informatics Center
Federal University of Pernambuco
Recife, Pernambuco, Brazil

ABSTRACT

This paper reports our experience using AspectJ, a general-purpose aspect-oriented extension to Java, to implement distribution and persistence aspects 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 application considered, and other similar applications. We have also identified a few drawbacks in the language and suggest some minor modifications that could significantly improve similar implementations. Despite the drawbacks, 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 pattern. The framework and the pattern 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.

Categories and Subject Descriptors

D.1 [**Software**]: Programming Techniques—Aspect-Oriented Programming; D.3.2 [**Programming Languages**]: Language Classifications—AspectJ

General Terms

Languages, Standardization, Experimentation

*Also affiliated to Catholic University of Pernambuco, Informatics and Statistics Department, Recife, Pernambuco, Brazil.

OOPSLA'02, November 4-8, 2002, Seattle, Washington, USA.

Keywords

Aspect-oriented programming, separation of concerns, AspectJ, distributed programming, object persistence

1. INTRODUCTION

This paper reports our experience using AspectJ [14], a general purpose aspect-oriented [6, 15] extension to Java [11], 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.

The distribution aspects implement basic remote access to system services using Java RMI (Remote Method Invocation) [18]. The persistence aspects implement basic persistence functionality using relational databases, and support the following main 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. During implementation of the aspects, we found a need for defining 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 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. 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. 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.

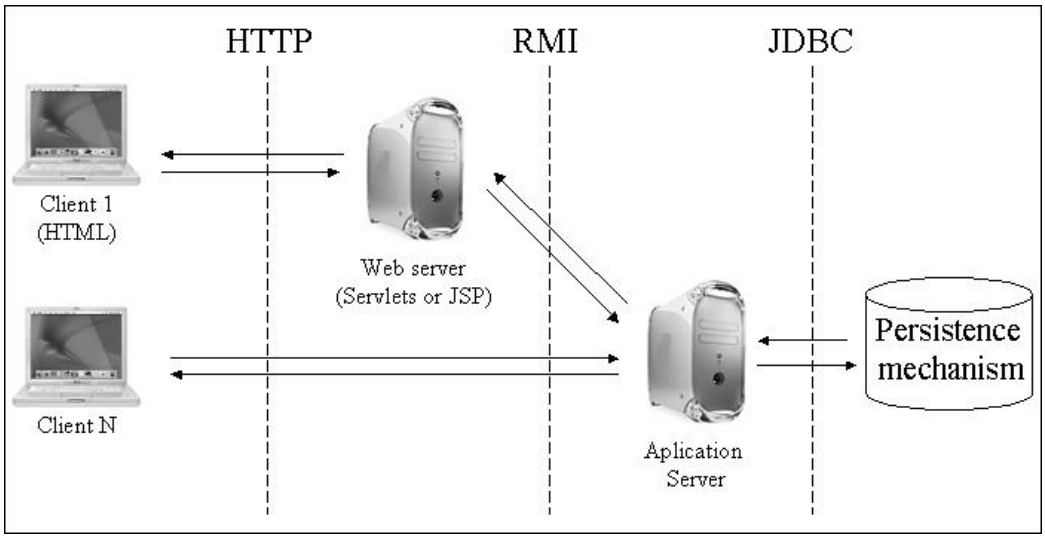

Figure 1: System configurations.

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.

This paper is structured as follows. Section 2 discusses the layer architecture and the Java implementation of the system used in the experiment. Section 3 overviews the AspectJ language. Distribution aspects are discussed in Section 4, persistence aspects in Section 5, and exception handling aspects in Section 6. Finally, Section 7 presents related work, and Section 8 contains our conclusions.

2. 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 [10, 1, 17] 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 cannot, in general, 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. The code for providing data access on demand cannot be untangled too.

The layer architecture of the Health Watcher system does not prevent spread code too. This is the case of the code specifying the classes that 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 on the facade class [10], 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 through several system configurations. Figure 1 shows two possible configurations, where a relational database is used as the persistence mechanism. In the one used in our experiment, the system is used through an HTML [12] and Javascript [8] 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 [23] (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.

Figure 2 presents part of the Health Watcher UML [2] class diagram. For simplification, it only shows the classes involved in the complaint processing services, the others essentially follow the same pattern; we also omit the classes from the communication layer, which allows 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 (`PersistenceMechanism`) should handle database connections and transaction management. Persistent data collections (`PersistentComplaintData`) are

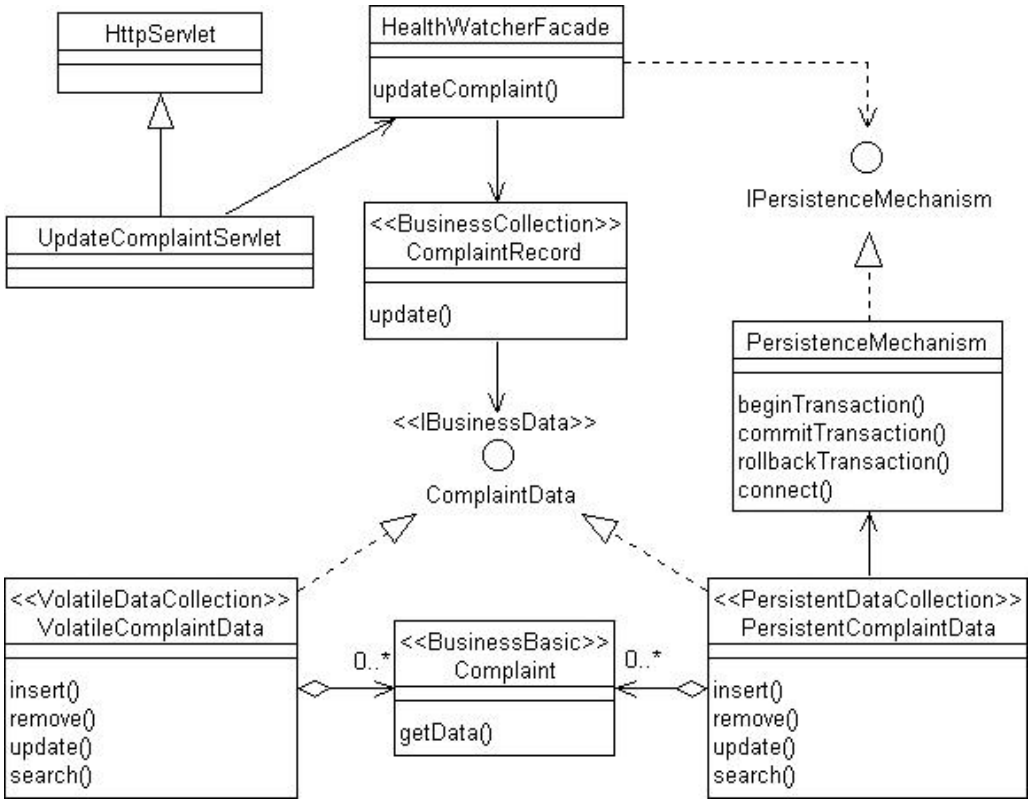

Figure 2: Partial Health Watcher class diagram.

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 (`VolatileComplaintData`).

3. ASPECTJ OVERVIEW

AspectJ [14] 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 tangled code and code spread 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.

The main construct of the AspectJ [14] language is called *aspect*. Each aspect defines a functionality that crosscuts others, called 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 be used to affect the static structure of Java programs, by using AspectJ's static crosscutting mechanism. This mechanism allows one to introduce new methods and fields 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* (instead of) 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.

Usually, an aspect declares *pointcuts* that select sets of join points and context values at those join points. The declarations of *advice* by an aspect specify the code that should be executed when a pointcut is reached during execution. The advice declaration indicates if the code should execute before, after, or around the pointcut.

Figure 3 shows an aspect definition, from the AspectJ Programming Guide [25], where the aspect `FaultHandler` adds one field into the `Server` class (line `3`) and defines two methods (lines `5` to `7` and `9` to `11`). The aspect also defines one pointcut (line `13`) and two advice (lines `15` to `17` and `19` to `22`). The introduced field indicates if the execution of any `public` method (identified by the pointcut definition) has thrown a `FaultException`, as specified by the `after` advice. The `before` advice checks the field before calling any `public` method, in this way avoiding methods call to disabled servers.

```
 1: aspect FaultHandler {
 2:
 3:   private boolean Server.disabled = false;
 4:
 5:   private void reportFault() {
 6:     System.out.println("Failure! Please fix it.");
 7:   }
 8:
 9:   public static void fixServer(Server s) {
10:     s.disabled = false;
11:   }
12:
13:   pointcut services(Server s): target(s) && call(public * *(..));
14:
15:   before(Server s): services(s) {
16:     if (s.disabled) throw new DisabledException();
17:   }
18:
19:   after(Server s) throwing (FaultException e): services(s) {
20:     s.disabled = true;
21:     reportFault();
22:   }
23: }
```

Figure 3: AspectJ example.

4. DISTRIBUTION ASPECTS

In order to minimize the deficiencies of the pure Java implementation of the system considered in our experiment (see Section 2), AspectJ was chosen to restructure the system and implement the distribution and persistence concerns. By doing this, we aimed to achieve better separation of concerns and avoid some tangled and spread code that cannot be avoided by simply using the layer architecture. Therefore, we hoped to obtain a more extensible system, supporting, without invasive changes, the several different configurations required by the Health Watcher stakeholders.

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 kinds of applications, we believe they can be used as specific guidelines for implementing distribution and persistence aspects in systems that comply to the architecture presented in Section 2. They can, likewise, be used as guidelines for restructuring such 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 to the architecture presented in Section 2, 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 structured as in Figure 4. 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.

4.1 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 parameter and return types, since this is required by RMI.

Making the facade remote

RMI remote objects must implement a so-called remote interface, which is used in order to access the remote services provided by those objects. Hence, as we want to make the facade remotely available, the server-side aspect has to modify the facade class (`HWFacade`) to implement a corresponding remote interface (`IHWFacade`). This is done by using the `declare parents` construct of AspectJ's static crosscutting mechanism:

```
aspect ServerSideHWDistribution {
  declare parents: HWFacade implements IHWFacade;
```

The `IHWFacade` interface 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, it is specific and auxiliary to the distribution aspects, so it was grouped with the other auxiliary types. This interface simply extends RMI's `Remote` interface and contains the signatures of the facade public methods, but including `RemoteException` in their `throws` clauses. This exception is used by RMI in order to indicate

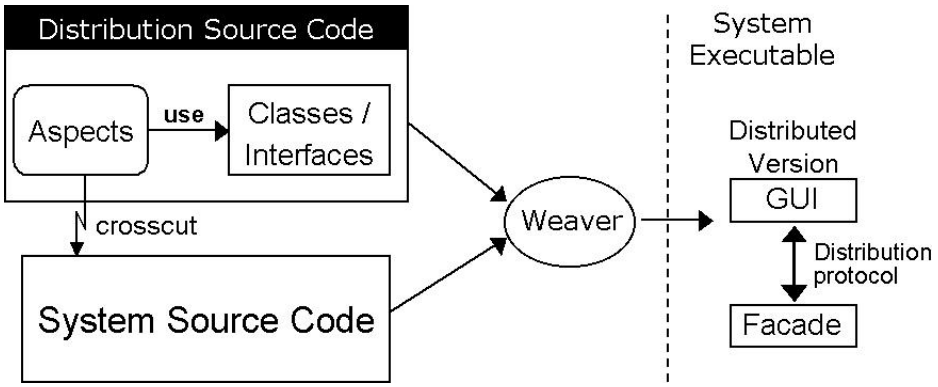

Figure 4: Distribution code weaving.

several kinds of configuration problems and remote communication failures.

Additionally, we should use the `declare parents` construct to make the facade class extend the RMI `UnicastRemoteObject` class, which defines the behavior of remote objects and makes their references remotely available. This approach, although recommended by RMI, would require the server-side aspect to specify that `RemoteException` should be added to the `throws` clause of the facade's constructor. This would be necessary because the subclass 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. The `exportObject` method is called by the facade `main` method, which essentially starts up the remote Health Watcher server. The aspect adds the `main` method to the facade class using static crosscutting:

```
public static void HWFacade.main(String[] args) {
  try {
    HWFacade f = HWFacade.getInstance();
    UnicastRemoteObject.exportObject(f);
    java.rmi.Naming.rebind("/HW",f);
  } catch (Exception rmiEx) { ... }
}
```

The qualified name `HWFacade.main` indicates that the `main` method should be added to the `HWFacade` class. This method exports the facade object and binds its remote reference to the `"/HW"` name, making it available to accept remote calls. This solution avoids the tangled code of the pure Java version without imposing any disadvantages.

Serializing types

The server-side aspect also declares that the methods of the facade have serializable parameter and return types. The only exceptions are for parameter and return values that correspond to remote objects themselves, which should not be serializable.

In order to be serializable, a class has to implement the Java `Serializable` interface, which indicates that 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 || complaint.DiseaseType ||
  complaint.Symptom implements
  java.io.Serializable;
```

This might indeed be repetitive and tedious, suggesting that either AspectJ should have more powerful metaprogramming constructs or code analysis and generation tools 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 spread way.

4.2 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 `IHWFacade`. 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 `IHWFacade` fields, but those classes would then depend on RMI code, not satisfying the Health Watcher requirements and going against our goals in restructuring the system.

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 to 4), as long as they originate from the user interface classes (line 5), which in our case are Java servlets:

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

In this code, the pointcut parameter `f` 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 `f` 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 6) 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:

```
6: Object around(HWFacade f) throws /*...*/:
7:     facadeCalls(f) {
8:   return proceed(remoteHW);
9: }
```

This advice affects the facade calls, exposing the reference to the target of each call (lines 6 and 7). It uses a reference to the remote instance (`remoteHW`, declared and initialized by the aspect) to proceed with the execution flow (line 8), 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 `f` 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 `IHWFacade`.

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 `registerComplaint` method is the following:

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

It redirects the `registerComplaint` 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 `IHWFacade`, 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 advice 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.

In spite of the problems with the implemented solution, it is superior to the corresponding pure object-oriented implementation. In fact, without using AspectJ, we could have worse 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 [10, 1] between the facade and the user interface classes. However, in this way, we need to write more code (around 20%) and a change to the facade would require changing two classes besides the facade and the remote interface. An alternative pure Java implementation could not separate the distribution code at all, but this would not satisfy the Health Watcher's adaptability and extensibility requirements.

As the necessary advice follow the same templates, we could use code generation [3] and refactoring [9, 4] techniques to reduce the problems with our solution, and also with the pure object-oriented solution, but it would be better to avoid those problems in the first place.

In order to eliminate the productivity and maintenance problems, we submitted a *feature request* to the AspectJ team, and they expect to consider that for the next version of AspectJ. We suggested the support of a new static crosscutting 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: (* IHWFacade.*(..))
  throws RemoteException;
```

This declaration would add the RMI specific exception, `RemoteException`, to the `throws` clause of all methods of the `IHWFacade` 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 `IHWFacade` fields, since the RMI details would be introduced to the interface by the distribution aspects. The general solution shown at the beginning of the section could then be used; we should only replace `HWFacade` for `IHWFacade` in that code.

The proposed feature would be useful to solve similar problems mentioned elsewhere [16], 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 has not 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 synchro-

nizing 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.

Surprisingly, later we concluded that this concern and its associated behavior are necessary for implementing persistence as well. So we actually implemented it only once, and the details are presented in Section 5. 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.

In our current implementation, the distribution aspects depend on the persistent aspects, which implement the data synchronization between the server and the clients. However, this concern could be easily factored out, being considered as a crosscutting concern to distribution and persistence. This would allow us to use the distribution aspects together with the state synchronization aspect, but without the persistence aspects when they are not necessary.

5. 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 to the architecture presented in Section 2, 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.

The persistence code was removed from the pure Java system and a similar functionality was implemented as aspects. Figure 5 illustrates that and also shows that we have aspects for making the system work with nonpersistent data structures. As discussed in Section 2, 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 and the `IBusinessData` interfaces (see Section 2). 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 also briefly explain an auxiliary aspect for supporting data collection customization (one can choose between nonpersistent and persistent).

5.1 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.

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 to the same architecture of the Health Watcher.

Implementing a reusable aspect

The abstract persistence mechanism control aspect is reusable. It defines 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 AbstractPersistenceControl {
  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 advice 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 `getPm` method declared by this aspect, but omitted here, creates, if necessary, and returns a valid `IPersistenceMechanism` instance. 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.

Those advice 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. So those exceptions were declared as *soft*, not checked, by the `declare soft` static crosscutting AspectJ construct. This allowed us to handle them in auxiliary exception handling aspects, defined in Section 6. Those aspects intercept the user interface code for properly handling the soft exceptions.

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.

Implementing a concrete aspect

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

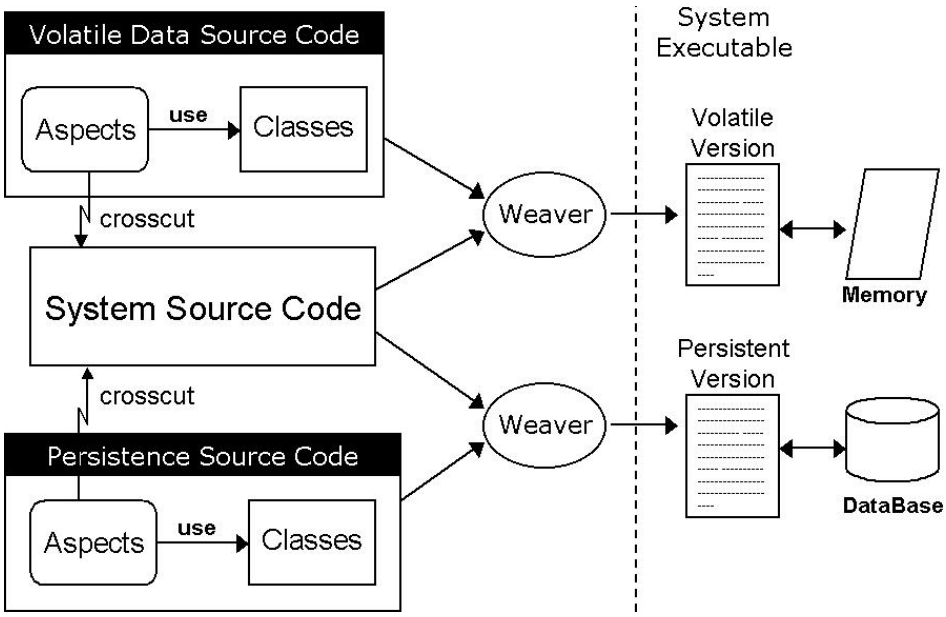

Figure 5: Persistence code weaving.

```
aspect PersistenceControlHW
    extends AbstractPersistenceControl {
  pointcut initSystem(): call(HWFacade.new(..));
  IPersistenceMechanism pmInit() {
    IPersistenceMechanism pm;
    pm = PersistenceMechanismRDBMS.getInstance();
    return pm;
  }
```

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, `PersistenceMechanismRDBMS`, 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(*
    PersistenceMechanismRDBMS.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.

5.2 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 [5]: 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.

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 AbstractTransactionControl {
  abstract pointcut transactionalMethods();
  abstract IPersistenceMechanism getPm();
```

It also declares an abstract method that is used to obtain a valid persistence mechanism instance; it is necessary for in-

voking the transaction services supported by the persistence mechanism.

The abstract transaction control aspect also implements three advice to start, successfully terminate, and abort 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 executes if any problem happens during the execution of any transactional method:

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

The `rollbackTransaction` method should rolls the transaction back to the original state, maintaining the database in a consistent state.

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 [16]. 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 only show the dependency between the Java code and the aspects that affect it.

Implementing a concrete aspect

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

When defining the concrete pointcut, we did not want to directly list the signatures of all transactional methods. This would impact 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 TransactionControlHW
    extends AbstractTransactionControl {
  declare parents: HWFacade
    implements ITransactionalMethods;
  pointcut transactionalMethods():
    execution(* ITransactionalMethods.*(..));
```

The aspect also uses the `declare parents` static crosscutting construct to make the facade class, which contains all transactional methods of the Health Watcher system, implement the `ITransactionalMethods` interface. This is necessary for associating the methods that are going to 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() {
  PersistenceControlHW a;
  a = PersistenceControlHW.aspectOf();
  return a.getPm();
}
```

This method yields a valid instance of the persistence mechanism. This is done by obtaining the instance that is available in the `PersistenceControlHW` 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.

With this approach, the aspect is not directly dependent on the transactional methods signatures, but the auxiliary `ITransactionalMethods` interface is totally dependent on them. In fact, the interface should contain a subset of the signatures of the methods defined by the facade class. This suggests that the interface could be easily generated by semi-automatically extracting information from the facade. This could be done 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

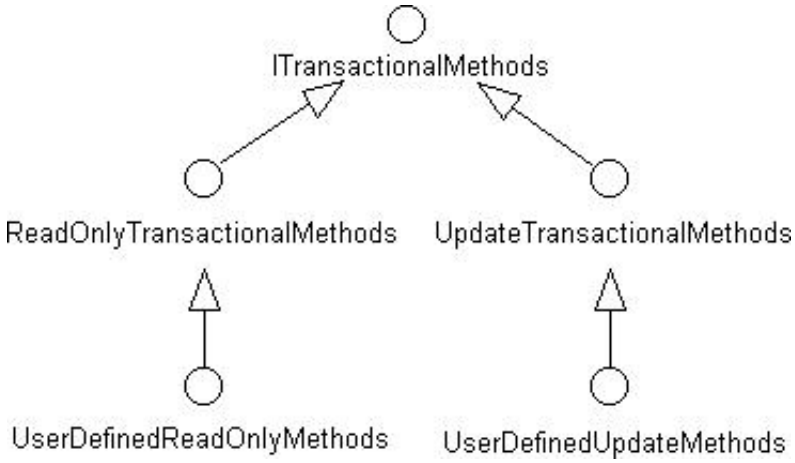

Figure 6: Transactional methods hierarchy.

controlling concurrency and failures with transactions [16]. 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 6 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 methods should then implement the specific interfaces, instead of simply implementing `ITransactionalMethods`, as done before.

In addition to a different interface hierachy, 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 an disadvantage too: as we separated related code, changes to a code unit might usually impact 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 transactional methods in two classes: `CitizenFacade`, defining the main system services, and `AdminFacade`, containing system administration and configuration services. So we would define two interfaces: `ITransactionalCitizen` and `ITransactionalAdmin`. Figure 7 shows the UML class diagram for this hierarchy.

Besides having the extra interfaces, we should extend the concrete `TransactionControlHW` 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.

5.3 Data state synchronization control

The business and presentation layers deal with persistent objects, which contain nonpersistent data that reflect the data stored in the database. Those layers invoke several methods on those objects, changing attribute values in the objects only. So, 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 4.2.

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). There-

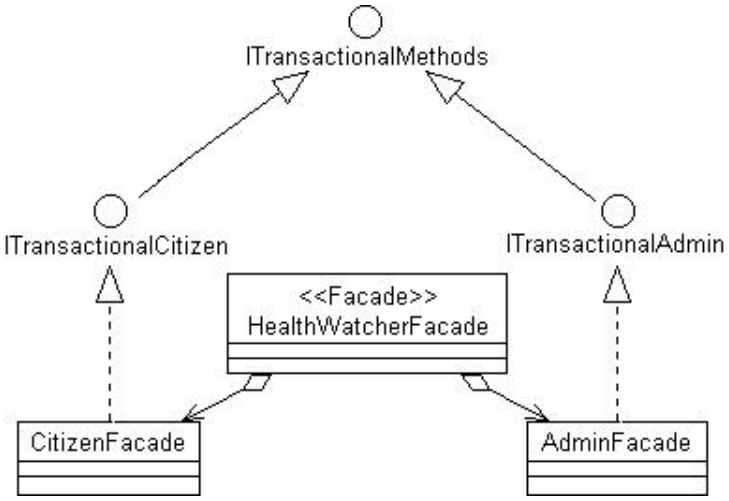

Figure 7: Example of multiple transactional components.

fore, 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 persistent object classes

The first thing declared by the data state synchronization control aspect is an internal interface (`PersistentObject`) used to identify classes whose objects are persistent:

```
aspect UpdateStateControl {
  private interface PersistentObject {
    void synchronizeObject(String s);
  }
  declare parents: Complaint || HealthUnit
    implements PersistentObject;
```

We use the `declare parents` construct to make the persistent classes implement the interface. In this example, classes representing health complaints, `Complaint`, and health units, `HealthUnit`, are declared to have persistent objects. For simplicity, we omitted the declarations for the other persistent classes, and the implementations of the `synchronizeObject` method for each persistent class. Those implementations are introduced into the classes using static crosscutting.

Identifying object updates

After identifying the persistent classes, we can identify when their objects are updated in the presentation layer. Those updates should be identified so that the updated objects are temporarily stored, in a nonpersistent data structure, and later synchronized with the database data. We used property-based crosscutting to simplify the specification of the updates in the presentation layer:

```
pointcut remoteUpdate(PersistentObject po):
  this(HttpServlet) && target(po) &&
  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 servlets and its persistent classes follow a name convention: the 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(PersistentObject po):
  this(*Record) && target(po) &&
  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(PersistentObject po) returning:
    remoteUpdate(po) {
  remoteDirtyObjects.add(po);
}
```

In fact, this is a simplification that works only for nonconcurrent systems. Instead of having just one set of updated objects, we should have one set for each system thread. In this way, the objects updated by one system client would

be stored in a specific structure for that client. This is essential for avoiding concurrency problems when storing and accessing the updated objects. 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 need to be synchronized:

```
pointcut remoteExecution():
  if (hasDirtyObjects()) &&
  this(HttpServlet) &&
  ( execution(void doPost(..))
    ||
    execution(void doGet(..)) );
after() returning: remoteExecution() {
  Iterator it = remoteDirtyObjects.iterator();
  while (it.hasNext()) {
    PersistentObject po =
        (PersistentObject) it.next();
    try {
      po.synchronizeObject("Remote");
    } 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.

Comparing with the pure Java version, this solution is easier to modify since the synchronization concern is completely separated. It is also conciser 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 to 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

Although not shown here, the various implementations of the `synchronizeObject` method call facade methods when receive the `"Remote"` argument, 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 4.2, the client-side distribution aspect is based on the `facadeCalls` pointcut, which intercepts only facade method calls originating from 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. 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. 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.

5.4 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 5.

For the nonpersistent version, we have one aspect responsible for attaching nonpersistent data collections to the system:

```
aspect VolatileDataCollections {
  after() returning (CitizenFacade facade):
      call(CitizenFacade.new(..)) {
    facade.setComplaintRecord(new
      ComplaintRecord(new
        ComplaintRepositoryArray())); ...
  }
}
```

This attachment is possible because both nonpersistent and persistent data collections implement the same interface. Similar aspects can also be defined to attach different kinds of persistent data collections. We omit the aspect that attaches the data collections using relational databases with Java Database Connectivity.

As discussed in Section 5.1, this kind of customization can also be supported by the pure Java implementation, with several advantages and some disadvantages.

5.5 Data access on demand

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 is adopted. In order to illustrate that, suppose that any subclass of `ListServlet` is used for generating a web page listing partial information about several objects of the same class. For example, such a subclass could generate a page with partial information about the various disease types registered in the system. In this case, the methods of `ListServlet` (and its subclasses) clearly adopt a particular kind of object usage, namely partial object loading. Therefore, we must define a pointcut matching the execution of those methods:

```
aspect ParameterizedDataLoading {
  pointcut listServlet():
    this(ListServlet+) &&
    execution(* *(..));
```

More pointcuts should be defined for later indicating the points where other kinds of object usages are necessary. We omit the details here, but they would be similar to `listServlet`.

Applying the adequate loading level

After specifying that the methods of `ListServlet` adopt a specific kind of object usage, we must, for instance, specify that this kind of usage should be applied when searching disease type objects in the associated data collection (`DiseaseRepositoryRDBMS`):

```
  pointcut diseaseTypeAccess(
    DiseaseRepositoryRDBMS rep,
    int code):
    call(DiseaseType search(int)) &&
    target(rep) && args(code)     &&
    cflow(listServlet());
```

The target and the argument of the `search(int)` 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 `listServlet` pointcut. Therefore, we intercept only `search(int)` method calls that originate from the execution of the methods of `ListServlet` and its subclasses. Similar pointcuts should be declared for other access methods called in the same context.

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

```
  Object around(DiseaseRepositoryRDBMS rep,
      int code) throws
      RepositoryException,
      ObjectNotFoundException:
      diseaseTypeAccess(rep, code) {
    return rep.searchByLevel(code,
      DiseaseType.SHALLOW_ACCESS);
}
```

We basically replace the `search` method call for a `searchByLevel` call, using the same target and argument. The specified shallow loading level corresponds to the level adopted by `listServlet`. 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.

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 `ListServlet` code, the programmer must be aware of the advice that intercept that code, otherwise it might try to have access to non loaded disease type information. It might even be necessary to change the aspect as a result of the `ListServlet` change. This shows a strong dependence between the aspects and the Java code, requiring more powerful AspectJ tools as discussed in Section 5.2. 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 code templates.

6. 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 with spread code, 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:

```
public abstract aspect ExceptionHandlingAspect {
  public abstract pointcut exceptionJoinPoints();
  after() throwing (Throwable ex):
      exceptionJoinPoints() {
    this.exceptionHandling(ex);
  }
  protected abstract void
    exceptionHandling(Throwable ex);
}
```

The aspect also defines an `after throwing` advice that executes when an exception is thrown 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 (line 3) that identifies the join points where a `PrintWriter` object is obtained (line 5) through the response object (line 4):

```
1: abstract aspect ServletsExceptionHandlingAspect
2:     extends ExceptionHandlingAspect {
3:   pointcut printWriterCreation():
4:     target(HttpServletResponse) &&
5:     call(PrintWriter getWriter());
```

It also declares an `after returning` advice (lines 7 to 11), which actually get and store the `PrintWriter` object returned by the `getWriter` method call:

```
 6: Hashtable printWriters = new Hashtable();
 7: after() returning (PrintWriter out):
 8:   printWriterCreation() {
 9:   Thread id = Thread.currentThread();
10:   printWriters.put(id, out);
11: }
```

We store the object in a `Hashtable` (line 6) indexed by the threads that execute the intercepted method calls. 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, obtains the correct `PrintWriter` object, and properly notifies the user. We should also define an advice for removing an object inserted into the hashtable as soon as the associated servlet request terminates. For simplicity, we omit the details here.

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. This should be done by specific aspects. In the Health Watcher system, we defined such a specific aspect 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 `ExceptionHandlingAspect`, providing the handling behavior and the join points to catch the exceptions.

7. RELATED WORK

Another AspectJ implementation of transactions was independently developed in the context of the OPTIMA framework for controlling concurrency and failures with transactions [16]. 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 applications. 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 an application 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 application programmer should not be aware of the aspects that intercept the application 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 (be-

gin, 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 5.2, 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 4.1).

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 5.5) than for the transaction aspects. It seems that our AspectJ implementation of transactions can usually have significant advantages over pure Java implementations (see 5.2). 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 collections and distribution services.

The implementation of distribution and persistence concerns in pure object-oriented applications was explored elsewhere, leading to specific design patterns [1, 17]. Those patterns support the progressive implementation of distribution and persistence code in an object-oriented application. Despite having similar goals, this approach does not achieve full separation of concerns; for instance, the distribution and persistence exception handling are tangled with user interface and business code. There is also code spread over several units, such as in the serialization mechanism implementation, and the identification of what objects should be made persistent. Several parts of the paper explain why the AspectJ version of the system is superior to a corresponding pure object-oriented implementation that follows most of those patterns.

Regarding distribution and aspects, another work [24] 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, and modifies bytecodes using Java reflection. In contrast, our approach uses a general-purpose language.

Some approaches relate persistence with aspects and databases. However, they investigate the persistence of the aspects [20], and also how to handle crosscutting features in the database implementations, generating aspect-oriented database management systems (AODBMS) [21]. Those approaches differ from ours because we use aspects to implement object persistence whereas they implement methods to persist aspects. Therefore, they are, in fact, orthogonal to what we propose here.

8. 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 allows us, for instance, to easily change the distribution middleware or the persistence mechanism without affecting the implementation of the other concerns.

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 [7]. 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 AspectJ.

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 or have the support of code generation tools 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.

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. 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 [16], 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 [26].

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 [22], 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 [4] 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.

9. ACKNOWLEDGEMENTS

We would like to thank the anonymous referees for making several suggestions that significantly improved our paper. Special thanks go to Doug Lea who was extremely helpful by reading the paper twice, discussing several important points, and pointing out related work. We also thank Gregor Kickzales for being available to give us some feedback on how to improve the paper.

This work was supported by CAPES and CNPq, brazilian research agencies. CAPES supported the first two authors and the third was supported in part by CNPq, grant 521994/96-9.

10. REFERENCES

[1] V. Alves and P. 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. UERJ Magazine: Special Issue on Software Patterns.

[2] G. Booch, I. Jacobson, and J. Rumbaugh. *Unified Modeling Language — User's Guide*. Addison-Wesley, 1999.

[3] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison–Wesley, 2000.

[4] M. d'Amorim, C. Nogueira, G. Santos, A. Souza, and P. Borba. Integrating Code Generation and Refactoring. In *Workshop on Generative Programming, ECOOP'02*, Málaga, Spain, June 2002. Springer Verlag.

[5] R. Elmasri and S. Navathe. *Fundamentals of Database Systems*. Addison–Wesley, second edition, 1994.

[6] T. Elrad, R. E. Filman, and A. Bader. Aspect–Oriented Programming. *Communications of the ACM*, 44(10):29–32, October 2001.

[7] R. E. Filman and D. P. Friedman. Aspect–Oriented Programming is Quantification and Obliviousness. In *Workshop on Advanced Separation of Concerns, OOPSLA'00*, 2000.

[8] D. Flanagan. *JavaScript The Definitive Guide*. O'Reilly & Associates, Inc., second edition, 1997.

[9] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[11] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, second edition, 2000.

[12] I. S. Graham. *The HTML Sourcebook*. Wiley Computer Publishing, second edition, 1996.

[13] J. Hunter and W. Crawford. *Java Servlet Programming*. O'Reilly & Associates, Inc., first edition, 1998.

[14] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. Getting Started with AspectJ. *Communications of the ACM*, 44(10):59–65, October 2001.

[15] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect–Oriented Programming. In *European Conference on Object–Oriented Programming, ECOOP'97*, LNCS 1241, pages 220–242, Finland, June 1997. Springer–Verlag.

[16] J. Kienzle and R. 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.

[17] T. Massoni, V. Alves, S. Soares, and P. Borba. PDC: Persistent Data Collections pattern. In *First Latin American Conference on Pattern Languages of Programming — SugarLoafPLoP*, Rio de Janeiro, Brazil, October 2001. UERJ Magazine: Special Issue on Software Patterns.

[18] S. Microsystems. Java Remote Method Invocation (RMI). At http://java.sun.com/products/jdk/1.2/docs/guide/rmi, 2001.

[19] G. C. Murphy, R. J. Walker, E. L. Baniassad, M. P.

Robillard, A. Lai, and M. A. Kersten. Does aspect–oriented programming work? *Communications of the ACM*, 44(10):75–77, October 2001.

[20] A. Rashid. On to Aspect Persistence. In *2nd International Symposium on Generative and Component–based Software Engineering*, LNCS 2177, pages 453–463. Springer–Verlag, October 2000.

[21] A. Rashid and E. Pulvermueller. From Object–Oriented to Aspect–Oriented Databases. In *11th International Conference on Database and Expert Systems Applications — DEXA 2000*, LNCS 1873, pages 125–134. Springer–Verlag, September 2000.

[22] S. Soares and P. Borba. PaDA: A Pattern for Distribution Aspects. In *Second Latin American Conference on Pattern Languages of Programming — SugarLoafPLoP*, Itaipava, Rio de Janeiro, Brazil, August 2002.

[23] Sun Microsystems. The Enterprise JavaBeans Specification, October 2000. At http://java.sun.com/products/ejb/docs.html.

[24] M. Tatsubori. Separation of Distribution Concerns in Distributed Java Programming. In *OOPSLA'01, Doctoral Symposium*, Tampa FL, 2001.

[25] A. Team. The AspectJ Programming Guide. At http://aspectj.org, 2002.

[26] J. Waldo, S. C. Kendall, A. Wollrath, and G. Wyant. A Note on Distributed Computing. Technical Report TR-94-29, Sun Microsystems Laboratories, Inc., November 1994.