Universidade Federal de Pernambuco
Centro de Informática

Pós-graduação em Ciência da Computação

An Aspect-Oriented Implementation Method

por

Sérgio Castelo Branco Soares

Tese de doutorado

Recife, 2004

UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA

Sérgio Castelo Branco Soares

An Aspect-Oriented Implementation Method

Este trabalho foi apresentado à Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito para a aprovação da tese de doutorado.

ORIENTADOR:
Prof. Paulo Henrique Monteiro Borba

Soares, Sérgio Castelo Branco

An Aspect-oriented Implementation Method / Sérgio Castelo Branco Soares. – Recife : O Autor, 2004.

xii, 166 f. : il., fig., tab.

Tese (doutorado) – Universidade Federal de Pernambuco. CIn. Ciência da Computação, 2004.

Inclui bibliografia e apêndice.

1. Engenharia de software. 2. Linguagens de programação (Orientada a aspectos) – Aplicação. 3. Desenvolvimento de software – Processos e métodos. 4. Experimentos (Engenharia de software) – Avaliação. 5. Ferramentas de suporte (Engenharia de software) - Implementação. I. Título.

004.415.3	**CDU(2.ed.)**	**UFPE**
005.117	**CDD (21.ed.)**	**BC2004-419**

Acknowledgment

First, I thanks to God for giving me strength to keep going since undergraduate. To my mother and father that despite being 1000 km far from me had always supported me and my family in all the ways they could and I needed. To my daughter, Heloisa, my wife, Fernanda, my brothers André and Ricardo, and my sister-in-law, Vanessa.

To my advisor, for being so competent and supporting me since undergraduate. I know it was hard.

To my friends, alphabetically cited to avoid jealousy, Cuca, Denise, Eduardo, Everaldo, Isabella, Marcelo, Paula, Paulinho, and Rosaly, which also played an important role.

To my PhD colleagues Tiago and Vander for reading and commenting previous versions of some chapters. To the undergrads, Ives, Madson, and Renan, that helped implementing the engine of the tool support and to the SPG research group for the discussions over those years, specially Leonardo, Rohit, Tiago, and Vander.

To the PUC-Rio's Separation of Concerns and Multi-Agent Systems group of the TecComm Group/Software Engineering Laboratory (LES), specially to Alessandro Garcia and Uirá Kulesza, for comparing the aspect-oriented and the object-oriented version of the Health Watcher software deriving interesting results.

To professor Guilherme Travassos, that helped me a lot when designing the experimental study to characterize the progressive approach, and the students that performed the study.

To the professors of the committee, Augusto Sampaio, Silvio Meira, Jaelson Casto, Guilherme Travassos and Paulo Masiero, for making important suggestions on how to improve this thesis.

To the Informatics Center, its professors and the technical staff, for supporting my research, and to CAPES for funding it.

Abstract

This thesis defines an Aspect-Oriented Implementation Method that defines data management, communication, and concurrency control concerns (requirements) as aspects. An aspect is a new abstraction mechanism, added by this new paradigm extending the object-oriented paradigm, aiming to increase software modularity, and therefore, software maintainability. The modularity reached by using aspects allows programmers to add or change software functionality with non-invasive changes, which keeps the base code clean and easy to understand and evolve. Furthermore, this avoids code of a specific concern to be tangled with other concerns and spread through several modules. We also define how the implementation method can be composed with the RUP development process, in order to tailor management, requirements, analysis, and design activities to support the method. Moreover, the method presents an alternative implementation approach that tries to anticipate requirement changes by yielding a functional prototype earlier than in a regular approach. This allows customers and developers to test the software before additional effort to implement non-functional requirements. A study was performed to characterize how useful this alternative implementation approach is, providing a support to decision-making when using the alternative or a regular approach. In addition, the method provides tool support to generate types of the base software and aspects to implement data management, communication, and concurrency control concerns. In fact, this tool guides the method application and the use of an aspect-framework provided by the method, which allows reusing base aspects for implementing those concerns. The method is tailored to a specific software architecture that despite being specific can be used to implement several kinds of software.

Resumo

Esta tese define um método de implementação orientado a aspectos que guia a implementação de requisitos (concerns) de comunicação (distribuição), gerenciamento de dados e de controle de concorrência como aspectos. Um aspecto é um novo mecanismo de abstração adicionado pelo paradigma orientado a aspectos estendendo o paradigma orientado a objetos. O objetivo desta nova abstração é aumentar a modularidade do software e, portanto, sua manutenibilidade. A modularidade alcançada pelo uso de aspectos permite que programadores adicionem ou modifiquem a funcionalidade do software com mudanças não-invasivas, as quais mantém o código base livre de detalhes sobre estas mudanças e, portanto, mais fácil de entender e modificar. Além disso, este tipo de mudança evita que códigos de diferentes requisitos (concerns) fiquem misturados com o código base e entre si e que fiquem espalhados por vários módulos do software. Tambm definimos como o método de implementação pode ser composto com o processo de desenvolvimento RUP, de modo a ajustar atividades de gerenciamento, levantamento de requisitos, análise e de projeto para que possam suportar a aplicação do método num contexto do desenvolvimento de software. Além disso, o método apresenta uma abordagem de implementação alternativa que tenta antecipar mudanças de requisitos através da implementação de protótipos funcionais mais precocemente do que numa abordagem regular. Desta forma, clientes e desenvolvedores podem testar o software antes de aplicar esforço adicional para implementar requisitos de distribuição, persistência e de controle de concorrência. Um estudo foi executado de modo a caracterizar quão útil é esta abordagem alternativa, provendo um suporte para a tomada de decisões sobre quando utilizar a abordagem alternativa ou a regular. Em adição, o método provê suporte automatizado para a geração de tipos do software base e de aspectos para implementar requisitos de gerenciamento de dados, comunicação e de controle de concorrência. De fato, esta ferramenta guia a aplicação do método e o uso de um framework de aspectos gerado pelo método, o qual permite um reuso de parte dos aspectos gerados neste trabalho. O método de implementação foi definido com base numa arquitetura de software específica que apesar de específica pode ser utilizada pra implementar vários tipos de softwares.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter introduces and motivates the need for implementation methods and improvements in object-oriented techniques. It also outlines the contributions of this thesis and how it is organized.

The following sections introduce this thesis work by discussing the research motivation, the chosen technologies, and the benefits of its contributions.

1.1 Motivation

Usually, researchers and software engineers do not give much attention to implementation methods [4, 39] because implementation mistakes have less impact in project schedule and development costs than mistakes during requirements and design. However, the effort given to requirements and design can be wasted if there is not a commitment with implementation activities.

This commitment is necessary in order to increase productivity, reliability, reuse, and extensibility levels. For example, the maintenance activities usually have the highest costs [16, 28, 77], which are inversely proportional to reuse and extensibility. This motivates the continuous search to increase productivity and quality factors, which can be achieved by using an appropriate implementation method, besides, of course, using appropriate analysis and design methods.

1.1.1 Object-oriented software development

Object-oriented programming languages provide effective means that help to increase productivity and quality. However, the object-oriented paradigm has several limitations, sometimes leading to code responsible for different requirements (concerns) mixed with each other, called tangled code, and code responsible for a single requirement (concern) spread over several places/modules, called spread code.

These limitations decrease modularity, and therefore, system maintainability. Examples are business code tangled with presentation code or data access code, and distribution, concurrency control, and exception handling code spread over several classes. Those problems can be reduced by adopting complementary techniques such as design patterns [26, 12].

In order to achieve better results [67, 66], in some cases one has to use new programming techniques [21, 7, 67, 94] that adapt or extend the object-oriented paradigm. These techniques aim to increase software modularity in practical situations where object-oriented programming and design patterns do not offer an adequate support.

1.1.2 Aspect-oriented software development

This thesis uses one of the new development techniques previously mentioned, aspect-oriented programming (AOP) [21, 42], to separate data management, communication, and concurrency control concerns for implementing systems with better modularity. In fact, aspect-oriented programming (AOP) has evolved into Aspect-Oriented Software Development (AOSD), and AOSD is nowadays considered a synonym for other techniques that deal with crosscutting concerns in general.

Actually, this thesis proposes an aspect-oriented implementation method using AspectJ [41], an aspect-oriented programming language extended from Java. The next chapter elaborates more on AspectJ and AOP.

The aspect-oriented implementation method definition proposes two different ways to implement crosscutting concerns. One alternative is to implement the different concerns at the same time the functional requirements are being implemented. Another idea is to follow a progressive approach, where persistence, distribution, and concurrency control are not initially considered in the implementation activities, but are gradually introduced, preserving the system's functional requirements. It was grateful to notice the natural match between the progressive implementation approach and the aspect-oriented paradigm.

This progressive approach helps to decrease the impact caused by requirement changes during development, since a great part of the changes might occur before the final (persistent, distributed, and concurrency safe) version of the system.

1.2 Methodology

The main goal of this thesis is to define an implementation method using aspect-oriented programming, helping to achieve better software with higher productivity levels. The implementation method defines patterns to structure the system architecture in order to comply with the method, and frameworks to support implementing the concerns in a separate way using aspect-oriented programming.

The method also defines activities to implement functional requirements, data management, communication, and concurrency control concerns of a system. It suggests how these activities are related and how they interact with *Use Case Driven Development* [35], a well known and used development technique, and which is adopted by the Rational Unified Process (RUP) [39]. The implementation method definition affects management, requirements, analysis, design, and test activities and, therefore, modifications to these activities are defined to comply with the method, showing that we recognize the importance of these activities in the software life cycle.

Besides the definition of the implementation method, this thesis has other contributions. We performed a study to characterize the benefits and drawbacks of using the progressive approach, which supports the decision of when using or not the progressive approach. When performing this study and other experiences on implementing aspects [91], we realized that the concerns implementations might affect each other and, therefore, the implementation activities should consider this side effect.

Tool support is also provided to generate some of the method's aspects and types, which guarantees productivity increasing.

1.3 Summary of contributions

In summary, this thesis' contributions are:

- The implementation method, including:
    - guidelines to implement distribution, data management, concurrency control, and exception handling concerns (Sections 3.4, 3.5, 3.7, and 3.8);
    - how those activities can be combined with *Use Case Driven Development* (Section 4.2;

  - how those activities affect RUP's workflows and activities (Section 4.3);
  - an alternative implementation approach, progressive implementation, and its impact on RUP (Section 3.10);

- An implementation with AspectJ that identified:

  - patterns tailored to aspect-oriented programming that define the software architecture used to implement the concerns (Section 3.2);
  - an aspect framework to implement data management, communication, and concurrency control concerns in AspectJ (Sections 3.4.4, 3.5.5, and 3.8.1);
  - dependencies and impacts among the concerns implementations (Section 3.9);
  - problems with the AspectJ language and proposed modifications to it (Sections 3.11 and 3.4.2);

- A study to characterize the benefits and drawbacks of the progressive approach (Chapter 5);

- A tool that increases productivity during the implementation activities, also guiding the aspects implementation (Chapter 6).

1.4 Thesis organization

This thesis contains seven chapters, including this.

Chapter 2 presents a new paradigm, aspect-oriented programming, and a language that implements it as an extension to Java, AspectJ.

The main contribution is described in Chapter 3, where the guidelines to implement data management, communication, and concurrency control concerns are presented, and in Chapter 4, where the guidelines of the aspect-oriented implementation method defined in the previous chapter are related to a software development process.

Chapter 5 relates the study made with the implementation method described in Chapter 3 in order to characterize benefits and drawbacks of the progressive approach.

A tool implemented to support the method by generating part of the aspects and other types is presented in Chapter 6.

Finally, Chapter 7 presents related work and Chapter 8 the conclusions and future work.

Chapter 2

An Overview of AspectJ

This chapter presents a new paradigm and language proposed to address problems that object-oriented programming does not solve properly. The language is used to define the aspect-oriented implementation method. Examples show how powerful this language is and prepares readers to understand the concepts. We also discuss benefits and drawbacks of using design patterns to implement some features of the AspectJ language.

2.1 Aspect-oriented programming

The need for quality software motivated the use of object-oriented programming [55, 9] towards higher reuse and maintainability levels, increasing development productivity and support for requirement changes. However, the object-oriented paradigm presents a number of limitations [66, 67], such as tangled and spread code across different concerns. A concern in software engineering terms means a particular goal, functionality, or requirement. Examples of tangled and spread code across different concerns are distribution code tangled with business and user interface code, and concurrency control code spread through several software units. In this example, distribution, business, user interface, and concurrency control are concerns of a software. Some of these drawbacks can be minimized by design patterns [12, 26].

On the other hand, extensions of the object-oriented paradigm (OO), such as aspect-oriented programming [21, 42], subject-oriented programming [67], and adaptive programming [34], try to address these OO limitations. These techniques allow software to reach a higher modularity in practical situations where OO and design patterns do not offer an adequate support.

Several researchers, including us, believe that aspect-oriented programming (AOP), and more generally aspect-oriented software development (AOSD), is very promising [63, 21, 45]. AOP tries to solve inefficiencies in capturing some of the important design decisions that a system must implement. This difficulty leads the implementation of these design decisions to be spread through the functional code, resulting in tangled code with different concerns. This tangled and spread code hinders system development and maintenance. AOP increases modularity by fully separating code that has specific goals (concerns) and affects different parts of the system. These are called crosscutting concerns. Typical examples are persistence, distribution, concurrency control, exception handling, and debugging.

By separating crosscutting concerns, AOP supports implementation that isolate functional from non-functional requirements. Figure 2.1 depicts the aspect-oriented software development steps. In (`a`) the developer must identify the concerns to be implemented from the software requirements. In the next step, a set of components written in an object-oriented programming language, such as Java [27], might implement functional requirements. On the other hand, a set of aspects (crosscutting concerns) related to the properties that affect system behavior might implement non-functional requirements using an aspect-oriented language. Step (`c`) is responsible for composing the concerns in a process called weaving, which may be automatically executed by a weaver of the aspect-oriented language. Using this approach, non-functional requirements can be easily manipulated without affecting business code (functional requirements), since they are not tangled and spread over the system. Therefore, AOP allows the development of programs using such aspects, including isolation, composition and reuse of the aspects code.

2.1.1 An example of a crosscutting concern — distribution

Consider a Health Watcher system as an example. This system allows a citizen to register complaints to the health public system. One of the Health Watcher's requirements is to allow several customers to access the system at the same time. Therefore, a client-

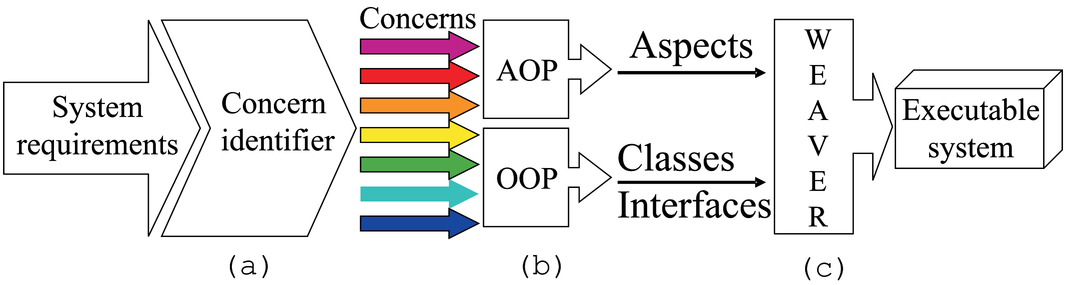

Figure 2.1: Aspect-oriented developement.

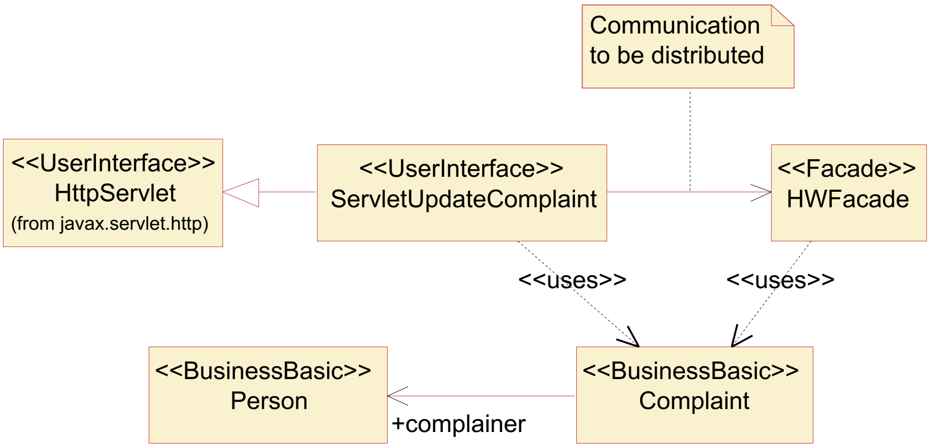

Figure 2.2: Health Watcher's class diagram.

server approach is used to distribute part of the execution. Our aim is to distribute the user interface, allowing simultaneous access. We used RMI [58] to implement this distribution.

Figure 2.2 shows a UML [10] class diagram presenting a few classes of the Health Watcher software. The diagram presents the facade [26] class, the unique access point to the system, and a Java Servlet [32], which is responsible for implementing the user interface. In this particular case, the servlet is responsible for updating complaint information into the system. Besides that, there are business basic classes that model complaints (`Complaint`) and complainers (`Person`). The communication to be distributed is the one between the user interface (servlets) and the system facade. This diagram is actually a simplification of the software architecture used by the Health Watcher software. The whole software architecture is presented in Section 3.2.

Distribution without AOP

To implement distribution without AOP, we must change several classes by adding specific code for using a selected distribution protocol. Figure 2.3 shows pieces of code[1] that have to be invasively changed in order to add distribution specific constructs. The

[1]Those pieces of code are not supposed to be readable. The idea is to visually show how tangled is distribution with business and user interface source code, without scanning the pieces of code.

distribution code is highlighted to demonstrate how tangled distribution is to business and user interface classes. In this case, the distribution protocol is RMI.

```java
public class Complaint implements java.io.Serializable {
    private String description;
    private Person complainer; ...
    public Complaint(String description, Person complainer, ...) {
        ...
    }
    public String getDescription() {
        return this.description;
    }
    public Person getComplainer() {
        return this.complainer;
    }
    public void setDescription(String desc) {
        this.description = desc;
    }
    public void setComplainer(Person complainer) {
        this.complainer = complainer;
    } ...
}
```

```java
public class HealthWatcherFacade implements IFacade {
    public void update(Complaint complaint)
            throws TransactionException, RepositoryException,
            ObjectNotFoundException, ObjectNotValidException {
        ...
    }
    public static void main(String[] args) {
        try {
            HealthWatcherFacade facade = HealthWatcherFacade.get Instance();
            System.out.println("Creating RMI server...");
            UnicastRemoteObject.exportObject(facade);
            java.rmi.Naming.rebind("/HealthWatcher");
            System.out.println("Server created and ready.");
        }
        catch (RemoteException rmiEx) {... }
        catch (MalformedURLException rmiEx) { ...}
        catch(Exception ex) {... }
    }
}
```

```java
public class ServletUpdateComplaintData extends HttpServlet {
    private IFacade facade;
    public void init(ServletConfig config) throws ServletExcepti on {
        try {
            facade = (IFacade) java.rmi.Naming.lookup("//HealthW atcher");
        }
        catch (java.rmi.RemoteException rmiEx) {...}
        catch (java.rmi.NotBoundException rmiEx) {...}
        catch (java.net.MalformedURLException rmiEx) {...}
    }


    public void doPost(HttpServletRequest request, HttpServletRe sponse response)
            throws ServletException, IOException {
        ...
        facade.update(complaint);
        ...
    } ...
}
```

```java
public class Person implements java.io.Serializable {
    private String nome; ...
    public Person(String nome, …) {
        this.nome = nome; …
    }
    public String getNome() {
        return nome;
    } …
}
```

```java
public interface IFacade extends java.rmi.Remote {

    public void updateComplaint complaint)
            throws TransactionException, RepositoryException,
            ObjectNotFoundException, ObjectNotValidException,
            RemoteException;
    . . .
}
```

Figure 2.3: Source code of the distributed software without AOP.

Besides being tangled with business and user interface code, the distribution code is spread over those different modules, which decreases software modularity, and therefore, maintainability and extensibility. If we want to change the distribution protocol, for example, from RMI to CORBA [65], we have to invasively change several classes. Changes might be required at the highlighted points to replace RMI specific code by CORBA. In fact, it could be necessary even adding more code in different places if using CORBA or another protocol.

Distribution with AOP

On the other hand, implementing distribution to the same software with AOP would require a distribution aspect that affects the monolithic software, adding distribution. As described in the following sections, AOP can change the static and the dynamic structure of a software. Figure 2.4 shows the classes source code of the distributed Health Watcher system using AOP.

Business and user interface code

```
public class ServletUpdateComplaintData extends HttpServlet {
    private HealthWatcherFacade facade;
    public void init(ServletConfig config) throws ServletExcepti on {
        try {
            facade = HealthWatcherFacade.getInstance();
        }
        catch (Exception ex) {...}
    }
    public void doPost(HttpServletRequest request, ... ) throws ... {
        ...
    } ...
}
```

```
public class Person {
    private String nome; ...
    public Person(String nome, ...) {  this.matricula = matricula; ...    }
    public String getNome() {  return nome;   } ...
}
```

```
public class HealthWatcherFacade {
    public void update(Complaint complaint)  throws TransactionException,
        RepositoryException, ObjectNotFoundException, ObjectNotValidException
    {
        ...
    }
}
```

```
public class Complaint {
    private String description;
    private Person complainer; ...
    public Complaint(String description,
                  Person complainer, ...) {   ...   }
    public String getDescription() {   return this.description;    }
    public Person getComplainer() {   return this.complainer;     }
    public void setDescription(String desc) {   this.description = desc;  }
    public void setComplainer(Person complainer) {
        this.complainer = complainer;
    } ...
}
```

Distribution aspects and auxiliary types using RMI

```
aspect DistributionAspect {
    declare parents: HealthWatcherFacade implements IFacade;
    declare parents: Complaint || Person implements java.io.Serializ able;
    public static void HealthWatcherFacade.main(String[] args) {
        try {
            HealthWatcherFacade facade =...;
            UnicastRemoteObject.exportObject(facade);
            java.rmi.Naming.rebind("/HealthWatcher");
        } catch (RemoteException rmiEx) {...}
        catch(Exception ex) {...}
    }
    private IFacade remoteFacade;
    pointcut facadeMethodsExecution():
        within(HttpServlet+) && execution(* HealthWatcherFacade. *(..)) &&
        this(HealthWatcherFacade);
    before(): facadeMethodsExecution()   { prepareFacade();}
    private synchronized void prepareFacade() {
        if (healthWatcher == null) {
            try {  remoteFacade = (IFacade java.rmi.Naming.looku p(...);
            } catch (Exception rmiEx) {...}
    }
    void around(Complaint complaint) throws  ..., RepositoryException
            ObjectNotFoundException,ObjectNotValidException:
            facadeRemoteExecutions()  && args(complaint)   &&
            call(void update(Complaint)) {
        try {    remoteFacade.update(complaint);
        } catch (RemoteException rmiEx) {...}
    } ...
}
```

```
public interface IFacade extends java.rmi.Remote {

    public void updateComplaint complaint)
            throws TransactionException, RepositoryException,
            …, RemoteException;
  . . .
}
```

Figure 2.4: Classes source code of the distributed system with AOP.

In this case, the business and user interface classes source code are preserved, since the distribution code is not tangled and spread all over those classes. If we need to change the distribution protocol, the changes are localized into the distribution aspect and its auxiliary types. The source code of the business and user interface classes would not be invasively modified to implement distribution.

In order to generate the distributed software, the distribution aspect must be composed to the business and user interface classes. Most aspect-oriented languages provide a compiler, so-called weaver, to automatically compose the aspects to the base software. Figure 2.5 depicts how the aspects affect the software when woven into it.

The weaver is a compiler that, given the core classes of the software and a set of aspects, generates a version of the software with these aspects. For example, by weaving our distribution aspect and the core classes, we have the distributed version of the classes using RMI. Therefore, in order to distribute the system using another distribution technology, another distribution aspect must be written to use the required technology. The new aspect is then weaved to the core software.

In this way we have defined a distribution aspect that affects the core classes to implement distribution with RMI, as presented in the class diagram of Figure 2.6. Note that the distribution aspect affects the user interface, the facade, and the data transmitted between them, as demanded by the distribution technology.

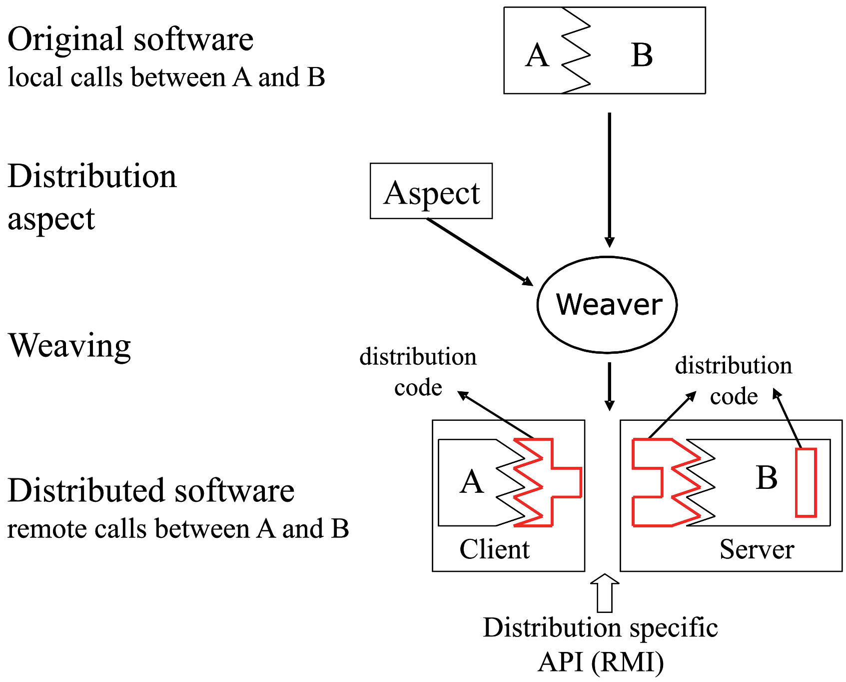

Figure 2.5: Distribution aspect woven into the software.

In the next chapter, we elaborate on distribution aspects presenting more details and more sophisticated aspects that define a distribution aspect framework. The distribution example used in this chapter is an oversimplified example, abstracting several details, just to present the aspect-oriented paradigm and the used language, which is described in the next section.

2.2 AspectJ

In this section we present AspectJ [41], a general-purpose aspect-oriented extension to Java. Programming with AspectJ allows developers to use both classes and aspects to separate concerns. Concerns that are well modeled as classes are separated that way; concerns that crosscut the classes are separated using units called aspects, and those are weaved with the classes to obtain a new AspectJ system. Actually, the resulting code is standard Java bytecode.

2.2.1 The anatomy of an aspect

The main construct of AspectJ is an aspect. Each aspect defines a specific function that might affect several classes of a software, for example, the distribution aspect previously mentioned. An aspect, similar to a class, can define members (fields and methods) and a hierarchy of aspects, through the definition of specialized aspects.

Aspects may affect the static structure of Java programs, by introducing new methods and fields to an existing class, converting checked exceptions into unchecked exceptions, and changing the class hierarchy by, for example, extending an existing class with

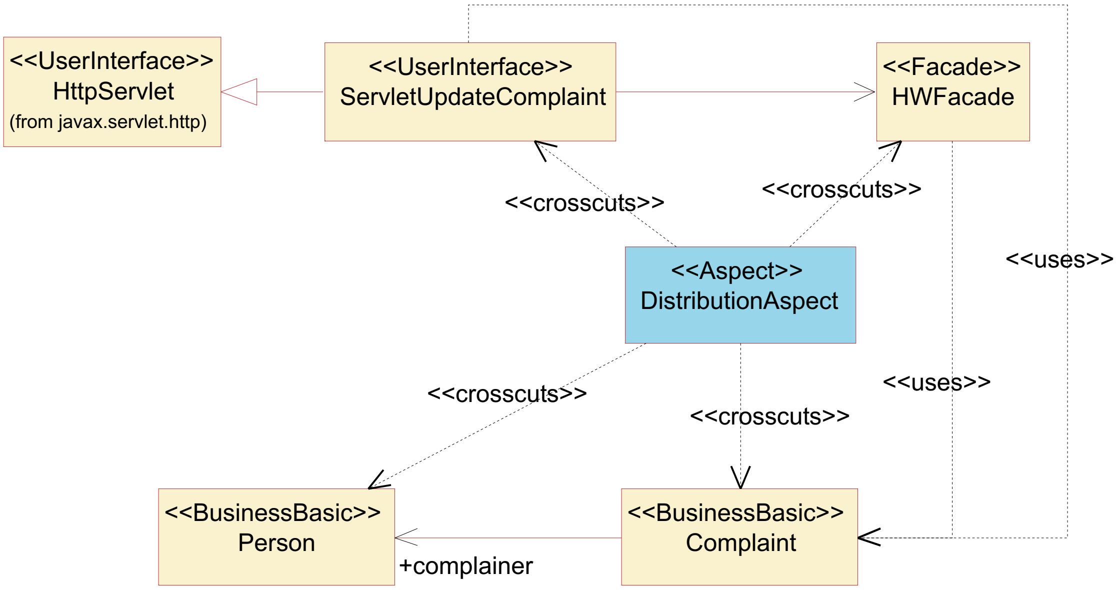

Figure 2.6: Health Watcher's class diagram with the distribution aspect.

another one. This is called static crosscutting.

Besides that, aspects can also affect the dynamic behavior of a software by changing the way a software executes. They can intercept certain points of the execution flow, called *join points*, and add before, after, or around (instead of) behavior to 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 logical operators.

Usually an aspect composes several join points, defining a pointcut. Pointcuts select join points and values at these join points. Besides identifying the points to be affected, the aspect defines advices that execute when the join points defined by the related pointcut are reached, changing the software behavior.

In the following sections, we present the AspectJ constructs and exemplify how they are used to implement the distribution in the Health Watcher system.

2.2.2 The join point model

A join point is a well-defined point in a program execution flow. Figure 2.7 [31] shows an example of execution flow between two objects and identifies some join points.

The first join point is the call of a method of object A, which can successfully execute and return or throw an exception. The next join point is the method execution, which can also successfully execute and return or throw an exception. During this method execution, a method of object B is called. This method call and its execution are also join points. Like object A's join points, those can successfully execute and return or throw an exception.

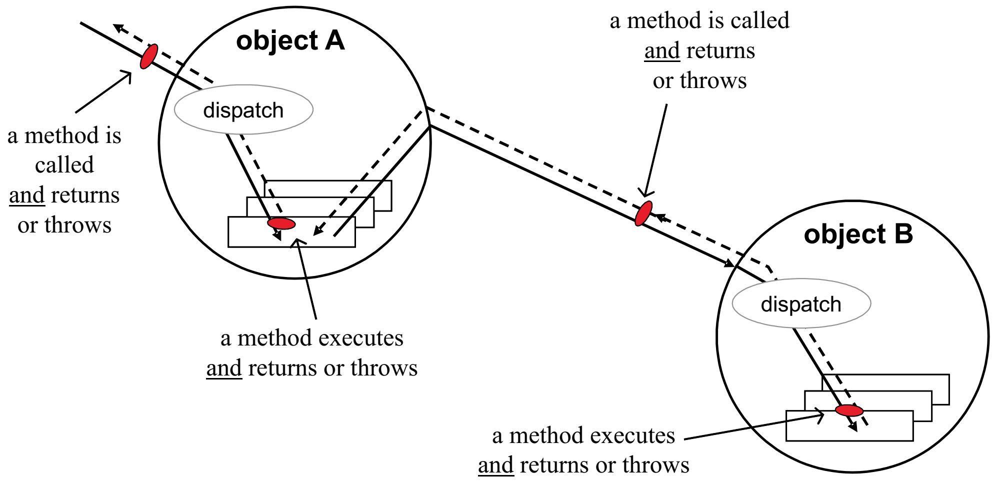

Figure 2.7: Join points of an execution flow [31].

2.2.3 Pointcut

Pointcuts are defined by composing join points through the use of && (and), || (or), and ! (not) operators. By using pointcuts we can expose and access values of method arguments, objects in execution, fields, and exceptions of the join points.

Consider the definition of the distribution aspect for the Health Watcher system. The following code fragment is the definition of a pointcut that identifies calls to all methods of the facade class, with any name and any return type, through the use of the "∗" wildcard, and any list of parameters through the use of the ".." wildcard.

```
pointcut facadeMethodsCall():
  within(HttpServlet+)  && call(* IFacade.*(..));
```

Note that this pointcut restricts the join points to the ones whose executing code are in Java Servlets. The wildcard "+" stands for subtyping. This means that the join points are restricted to Java Servlets or any subtype, which are the Health Watcher servlets. Therefore, the `facadeMethodsCall` pointcut identifies facade method calls made by user interface classes. Those method calls are the ones to be remotely executed.

There are other AspectJ constructs for defining pointcuts identifying the join points, called pointcut designators. Table 2.1 presents selected designators, used by the aspects in the next chapter.

In Table 2.1, *Signature* is a method or constructor signature, *Expression* is a boolean expression, and *TypePattern* can define a set of types using wildcards, such as $*$ and $+$. The former is a well-known wildcard, and can be used alone to represent the set of all types of the system, or after any character, representing any sequence of characters. The latter should be used after a type name to represent the set of all its subtypes.

The complete list of wildcards and pointcut designators can be found in the AspectJ Programming Guide [97].

`call`(*Signature*)	Method or constructor call matched by *Signature*
`execution`(*Signature*)	Method or constructor execution mached by *Signature*
`get`(*Signature*)	Field access matched by *Signature*
`set`(*Signature*)	Field assignment matched by *Signature*
`this`(*TypePattern*)	Matches join points where the currently executing object is an instance of *TypePattern*
`target`(*TypePattern*)	Matches join points where the target object is an instance of *TypePattern*
`args`(*TypePattern*, ...)	Matches argument objects that are instances of *TypePattern*
`within`(*TypePattern*)	Matches executing code defined in the types in *TypePattern*
`withincode`(*Signature*)	Matches executing code defined in the method or constructor with signature *Signature*
`cflow`(*Pointcut*)	Matches executing code in the control flow of the join points specified by *Pointcut*
`if`(*Expression*)	Matches executing code when the *Expression* evaluates to true

Table 2.1: Pointcut designators.

2.2.4 Advice

An advice defines the additional code that should execute at join points. Considering the distribution aspect, the next code fragment defines an advice that uses the previously defined pointcut (`facadeMethodsCall`). This advice is responsible for assuring that the remote instance is available before the user interface executes any call to the facade methods.

```
private IFacade remoteFacade;
before(): facadeMethodsCall() {
  remoteFacade = this.getRemoteInstance();
}
```

where the `getRemoteInstance` method retrieves a reference to the facade remote instance, using RMI API, and stores it in an aspect field (`remoteFacade`).

Next, we must write an advice to redirect the calls to the local facade instance to calls to the remote instance. The following advice redirects calls to update complaint objects from the local to the remote facade instance.

```
void around(Complaint complaint) throws ... :
    facadeMethodsCall()  &&
    call(void update(Complaint)) && args(complaint) {
  try {
    remoteFacade.update(complaint);
  }
  catch (RemoteException rmiEx) { ... }
}
```

This advice uses the `facadeMethodsCall` pointcut and restrict it to calls to the `update` method that receives a complaint as parameter. By using the `args` designator we expose the argument value, binding it to the advice's `complaint` parameter. Note that the `around` advice has total control over the join points execution flow. In this case, the affected join point is not executed; its execution is replaced by the remote call. However, it is possible to resume the join point execution by calling the `proceed` method inside the advice. The kinds of advices supported by AspectJ are presented by Table 2.2.

`before`	Executes immediately before the join point execution
`after returning`	Executes after the success execution of the join point
`after throwing`	Executes after the execution of the join point that thrown an exception
`after`	Executes after the execution of the join point, returning normally or throwing an exception
`around`	Executes when the join point is reached, it has complete control over its execution

Table 2.2: AspectJ advice.

2.2.5 Static crosscutting

As previously mentioned, the AspectJ language allows changing the static structure of a software by adding class members, changing the classes' hierarchy, or replacing checked by unchecked exceptions. In the following sections we present and exemplify these situations.

Inter-type declarations

The mechanism that adds members to a class is called inter-type declarations. AspectJ can add concrete or abstract methods, constructors, and fields to a class.

The examples shown define pointcuts and advices that only affect classes from the user interface. The following code fragment is an example of static crosscutting that adds a `main` method in the facade class. This method exports and names a facade instance using the RMI API, allowing the instance to respond to remote invocations. Other AspectJ inter-type declaration constructs are presented in Table 2.3.

```
public static void HealthWatcherFacade.main(String[] args) {
  try {
    HealthWatcherFacade facade = HealthWatcherFacade.getInstance();
    UnicastRemoteObject.exportObject(facade);
    java.rmi.Naming.rebind("/HealthWatcher");
  }
  catch (RemoteException rmiEx) { ... }
  catch (MalformedURLException rmiEx) { ... }
  catch(Exception ex) { ... }
}
```

Modifiers Type TypePattern.Id(*Formals*) {*Body*}	Defines a method *Id* in the types of TypePattern
`abstract` *Modifiers Type TypePattern.Id*(*Formals*);	Defines an abstract method *Id* in the types of TypePattern
Modifiers Type TypePattern.`new`(*Formals*) {*Body*}	Defines a constructor in the types of TypePattern
Modifiers Type TypePattern.Id[= *Expression*];	Defines a field in the types of TypePattern

Table 2.3: AspectJ inter-type declarations.

Other constructs

Due to a RMI demand we must define a remote interface (`IFacade`) to the class whose objects are remotely accessed, which is the facade class, adding `RemoteException` in its methods `throws` clause. This interface is an auxiliary type of the distribution aspect. The facade class has to implement the defined RMI remote interface. An example of AspectJ constructs that change the original software hierarchy is depicted by the next code fragment.

```
declare parents: HWFacade implements IFacade;
```

Table 2.4 presents other AspectJ constructs that change the static structure of a program. Additional information about static crosscutting can be found in the AspectJ Programming Guide [97].

`declare parents` : *TypePattern* `extends` *TypeList*;	Declares that the types in *TypePattern* extend the types in *TypeList*
`declare parents` : *TypePattern* `implements` *TypeList*;	Declares that the types in *TypePattern* implement the types of *TypeList*
`declare soft` : *TypePattern*: *Pointcut*;	Declares that any exception of *TypePattern* raised at any *join point* identified by *Pointcut* should be wrapped into an unchecked exception

Table 2.4: Other constructs.

2.2.6 Reusable aspects

AspectJ allows the definition of abstract aspects that have to be extended to provide the implementation of the abstract component. Those components can be methods, like in a Java class, and pointcuts, which have to be defined in a concrete aspect that reuses the behavior of the abstract aspect.

Consider the exception handling concern. A possible solution to implement such concern would define an abstract aspect that declares an abstract pointcut responsible for defining points of the program flow that might raise the exceptions to be handled. In addition, the aspect defines an advice to catch the exception raised after the execution of the join points of the abstract pointcut.

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

Note that the exception handling is actually performed by an abstract method that should be defined in a subaspect. This very broad aspect defines the general behavior to handle exceptions.

The next step is to write subaspects to define exception-handling policies for different concerns or software that use several user interfaces. For example, the next aspect defines how to handle exceptions in Java servlets. The aspect only provides the implementation of the `exceptionHandling` method, and therefore, it is still abstract.

```
public abstract aspect ServletsExceptionHandling
    extends ExceptionHandling {
  protected void exceptionHandling(Throwable ex) {
    // code to handle exceptions in Java servlets
  }
}
```

The following aspect reuses the `ServletsExceptionHandling` aspect behavior to handle exceptions related to the Health Watcher software.

```
public aspect HealthWatcherExceptionHandling
    extends ServletsExceptionHandling {
  public pointcut exceptionJoinPoints():
    DistributionAspect.facadeMethodsCall();
}
```

This concrete aspect defines the join points where the Health Watcher exceptions should be handled. In this case, the join points can be derived from the `facadeMethodsCall` pointcut of the distribution aspect.

By using this approach, we would have the hierarchy showed by the class diagram of Figure 2.8. The diagram shows an example where there are two aspects to handle exceptions: for Java servlets and applications that use the swing API. Those aspects can be reused to handle exceptions in different applications (Health Watcher, APP1, APP2, and APP3).

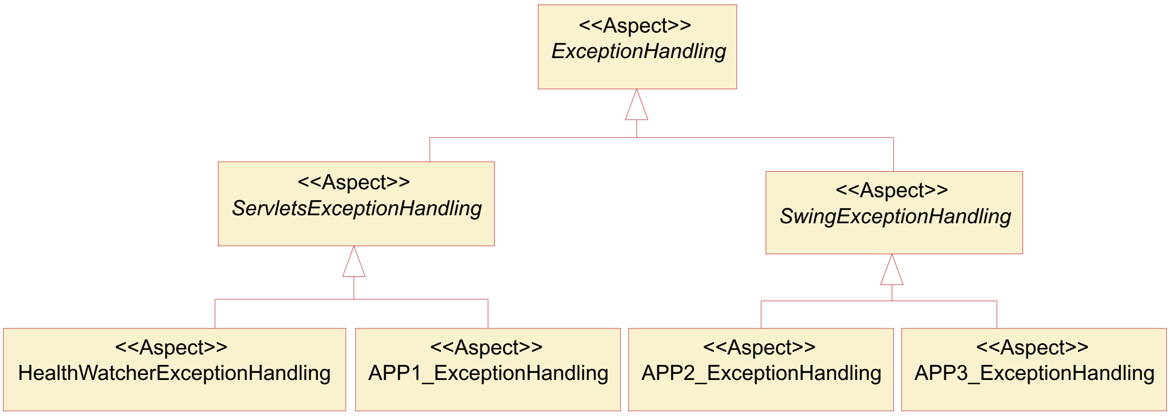

Figure 2.8: Exception handling aspects class diagram.

2.2.7 AspectJ expressiveness

The following aspect, defined in 14 lines of code, debugs a program that uses the JDBC [103] API to execute SQL [20] commands. This is an example of development aspect[2] that shows to the programmer the string to be submitted to the database before the submission. Usually this SQL command is created by several string concatenations, which might be difficult to define it correctly.

```
aspect DatabaseDebugging {
  private interface TypesDebugged { }
  declare parents : DataCollection1 ||
                    DataCollection2 ||
                    ...
                    DataCollectionN implements TypesDebugged;
  pointcut queryExecution(String sql):
    call(* Statement.*(String)) && args(sql) &&
    this(TypesDebugged);
  after(String sql) throwing: queryExecution(sql) {
    System.out.println(sql);
  }
}
```

The aspect declares an interface used to flag the classes to be debugged, which have to implement the defined interface. It also declares a pointcut to identify calls to any methods of the Statement interface that receive a String as a parameter. Those

[2]Development aspects are used only during development, in contrast to production aspects, which should be deployed with the software to the clients environment.

methods are responsible for receiving a string with the SQL command and then executing it on the database. The pointcut also expose the SQL command to be executed. After that, an advice is defined to print the SQL string in the console output if any exception is thrown after executing the SQL command. In order to provide such a behavior an `after throwing` advice is used. A `before` or `after` advice can be used to print every SQL command before or after executing it into the database. This can also be used in order to log these executions, instead of just printing them.

2.3 AOP and design patterns

Part of the aspect-oriented programming functionality can be implemented using design patterns [26], such as the separation of code with different concerns. For example, we can use the Adapter [26] pattern to add a behavior to a class. The class diagram depicted by Figure 2.9 presents the class `Source` that uses the service `m` of the `ITarget` interface.

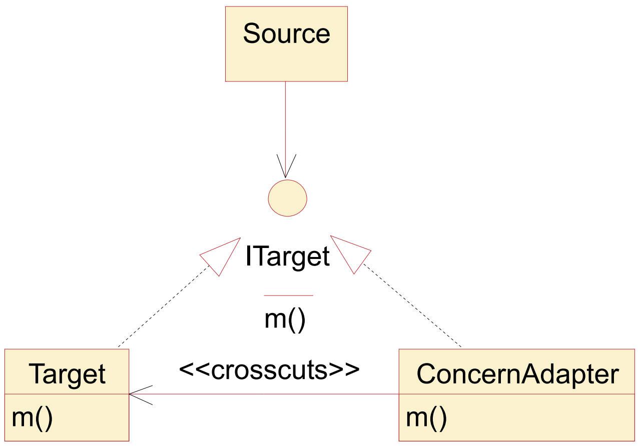

Figure 2.9: Adapters implementing separation of concerns.

The `Target` class has the business logic, the code that implements the software's functional requirements. Now consider the need for implementing an "aspect" to affect executions of the `Target`'s `m` method. An adapter, which has the reference to an object of `Target`, can do this by implementing the `ITarget` interface, adding the new behavior, and delegating, if necessary, the execution to the `Target` object, in order to execute the business logic.

This approach simulates the AspectJ `around` advice where the adapter's `m` method has total control over the execution of the `Target`'s method `m`. The delegation to execute the method of the target object is similar to calling the `proceed` method inside the `around` advice.

However, this design pattern approach leads to code duplication, since for each affected class we need an adapter, even for implementing a single concern (aspect), like debugging, for example. We showed in the previous section that using AspectJ, an aspect could be written to debug several, potentially all, classes of a software. On the other hand, the use of patterns makes it explicit the structural change.

Another drawback of the design pattern approach is the impossibility to capture calls made by the `Target`'s `m` method to another method of `Target`. This would require the `Target` class to use another adapter. In contrast, using AspectJ these internal method calls could be captured. In addition, changing the adapter implementation would require an invasive change, whereas AspectJ allows changes at compile time when the aspects to compose (weave) into the software are chosen.

2.4 Conclusion

Aspect oriented programming (AOP) provides several benefits due to the ability to increase software modularity. The aim of AOP is to separate crosscutting concerns avoiding code tangling and spreading over several units. Therefore, software maintainability and extensibility can be potentially increased, besides increasing software core reuse, since the aspects code are not mixed with the software's core code.

The AspectJ language is an aspect-oriented extension of the Java language, allowing aspect-oriented programming with Java. Some drawbacks are the need for understanding and familiarizing with a new programming paradigm and the low maturity of the development environment (AspectJ 1.0.6). However, the environment has considerably improved, and the support team is ready to answer questions and to provide workarounds to solve problems, through a discussion list. Some points to improve are the compilation (weaving) time and the generated bytecode size.

Another AspectJ drawback is the exception handling policy. If we add a new behavior into a software, such behavior might raise new exceptions. However, AspectJ does not allow adding new exceptions into a method `throws` clause. The AspectJ solution is to use soft exceptions, which wrap the new raised exception into an unchecked exception. This does not obligate the programmer to handle the unchecked exceptions, which might lead the software to respond unexpectedly to those errors. Next chapter elaborates on this.

Moreover, AspectJ provides very powerful constructs that should be used with care. For instance, the use of wildcards might affect several points of a software, including unexpected ones.

The pointcut definition demands the identification of specific points of a program. Since these points are identified using type, method, and parameter names, the aspects are dependent on the software, or on its naming standards. This can minimize the aspects reuse and changes made into the core software might affect the way the aspects work. For example, we can identify all calls to methods that insert data into the system by using the method name `insert`. However, this would always demand this name to define methods that insert data into the system, instead of `register` or `add`. There are Integrated Development Environment (IDE) extensions that support development with AspectJ, for example, with Borland JBuilder and Eclipse. These extensions allow visualizing which parts of the code are affected by aspects and vice-versa. Therefore, we consider essential the use of an IDE when programming with AspectJ.

A great advantage of using AspectJ is software customization, where functionalities are implemented separately in relation to the core software, and automatically added or removed. To add a functionality (aspect) we need only to compose (weave) the core software with the aspects needed, and to remove an aspect we weave the software with the aspects without the one to be removed. We can also have different implementations of a functionality, such as several aspects that implement several distribution protocols, or aspects to implement persistence in different storage media.

There are some alternatives to the AspectJ language, such as HyperJ [68, 94], a language that supports multi-dimensional separation of concerns for Java. HyperJ has constructions different of AspectJ, like hyperslice and hypermodule. A hyperslice is similar to an aspect, in the sense it defines a unit to encapsulate a concern. Another HyperJ construction is a hypermodule, a set of hyperslices that should be composed to a software. Due to its similarity to Java and the wider use of AspectJ, which implies in a larger community support, that was the chosen language.

Chapter 3

Guidelines for aspect-oriented implementation

This chapter presents guidelines derived from restructuring an object-oriented software to an aspect-oriented one. From these guidelines we derived the aspect-oriented implementation method, which uses AspectJ to modularize data management, communication, and concurrency control concerns.

As mentioned in Chapter 1, implementation methods are usually disregarded by software developers, as implementation mistakes may have less impact in project schedule and costs than requirements or design mistakes. If there is not a commitment with the implementation activities, all the effort given to requirements and design can be wasted when performing implementation activities.

No matter how good the programming language or paradigm, an implementation method is important to guide the definition of the implementation activities to be executed and their interrelations, including execution order. By focusing on implementation activities, we do not disdain analysis and design activities.

In fact, another important point covered by the implementation method is how the implementation activities, the specific software architecture, the aspect-oriented software development, and others, affect other important activities of the system life cycle, such as analysis, design, and tests, and how they can be composed to a development process (see Chapter 4).

Our main goal is to define an implementation method using aspect-oriented programming, aiming at software quality with increasing productivity levels. Our method guides the implementation of data management, communication, and concurrency control concerns that comply with a specific software architecture, resulting in a more extensible and modular software.

3.1 Introduction

The first step towards an implementation method was to use AspectJ [41] to implement distribution and persistence aspects [91] in a simple but real and non trivial web-based health complaint system, which was originally implemented in Java. After that, concurrency control aspects were implemented to guarantee safety. These aspects evolved considerably, resulting in improved and more reusable aspects, defining a suitable framework for reuse [90]. In fact, the persistence aspects are treated as part of a more general concern, data management, since the method should also support nonpersistent media, as well as the distribution aspects, which are part of the communication concern.

The distribution aspects implement basic remote access to system services using Java RMI (Remote Method Invocation) [59]. They are mainly concerned with establishing communication between the user interface and the remote facade class. A distributed environment is inherently concurrent. Therefore, it is necessary to define concurrency control aspects in order to guarantee a safe execution of the software in such an environment. In addition, data management aspects are responsible for storing the software's data. The aspects may provide nonpersistent and persistent storage. The persistence aspects implement basic persistence functionality using relational databases, and support the following main concerns: connection and transaction control, partial (shallow) object loading, for improving performance, and synchronization of object states with the corresponding database entities, for ensuring consistency. During implementation of those aspects, it was necessary to define auxiliary exception handling aspects, which are also present in this chapter. We discuss the lessons learned in implementing those aspects and justify our design decisions.

Some of the aspects implemented in our restructuring experience are abstract and constitute a simple aspect framework. They can be extended for implementing data management, distribution, and concurrency control in other applications that comply with the architecture of the health complaint system, a layered architecture used for developing web-based information systems. The other aspects are application specific and therefore may be implemented differently in other applications. Nevertheless, we suggest that different implementations might follow a common aspect pattern, presenting aspects with the same structure. Based on the framework and on 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.

Figure 3.1 depicts the aspect-oriented restructuring presented in this chapter, where an aspect-oriented software is derived from an object-oriented one. The guidelines can also be used for aspect-oriented implementation from the beginning.

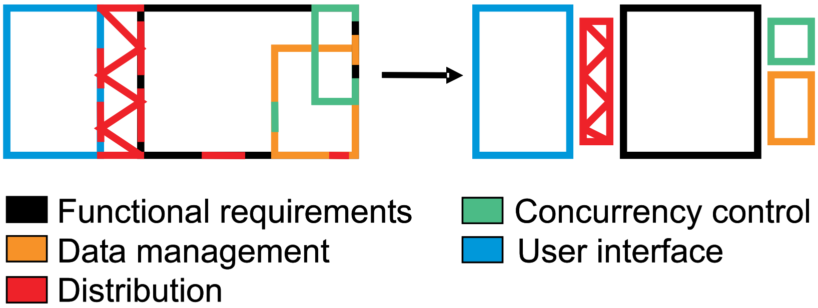

Figure 3.1: Aspect-oriented restructuring.

3.2 The Specific Software Architecture

The implementation method is tailored to a specific architecture. Despite being specific, this software architecture can be used to implement a broad range of systems [89]. Examples of real web-based systems that use this architecture are the following:

- A system to manage a telecommunication company's clients. The system is able to register mobile telephones and change client and telephone services configurations. The system can be used over the Internet.

- A system for performing on-line exams. This system is been used to offer different kinds of exams, as simulations based on previous university entry exams, which help students to evaluate their knowledge before the real exams.

- A complex supermarket system. The system is responsible for controlling the sales in a market. This system is been used in several supermarkets and other kinds of stores and has more than 2 million lines of code.

- A system for registering health system complaints. The system allows citizens to complaint about disease problems and allows retrieving information about the public health system, such as the location or the specialties of a health unit. This system was used as a case study in several works [80, 1, 52, 48], including this thesis, being useful to derive and tailor our method. We also use it to exemplify the software architecture as shown through this section.

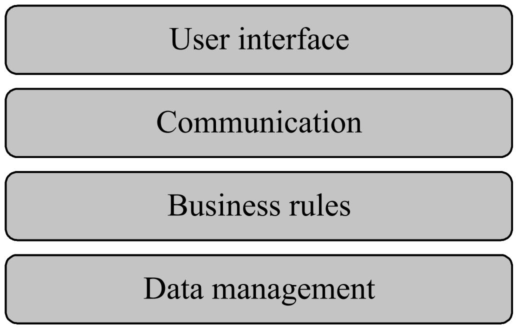

Figure 3.2: Four-layered architecture.

The object-oriented version of the software architecture was primarily conceived to implement a four-layered architecture depicted by Figure 3.2. The architecture aims at separating user interface, communication, business rules, and data management concerns. This structure leads to less tangled code — such as when business code is mixed with distribution code — but does not completely avoid it. For example, the code for starting and terminating transactions, in general, cannot be easily untangled from the business code 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 [2, 3] 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 either.

Furthermore, the layered architecture of the object-oriented Health Watcher system does not prevent spread code. This is the case of the code specifying which classes have to be made serializable for allowing the remote communication of its objects. The exception handling code for data management, distribution, and concurrency control is also scattered throughout the system. The code for handling transactions appears only in the facade [26] class, the unique entry point to the system, but it is essentially replicated to all transactional methods of this class.

Despite not completely separating concerns, the layered architecture gives some support to adaptability. Figure 3.3 shows two possible system configurations, where a relational database is used as the persistence mechanism. In the one used in our restructuring experience, the system is accessed through an HTML [29] and Javascript [23] user interface, which interacts with Java servlets [32] 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 [93] (Enterprise JavaBeans) or another distribution technology. Similarly, we could also have an object-oriented database as the persistence mechanism.

On the other hand, by using aspect-oriented programming and AspectJ, we could improve the architecture deriving a fifth layer, depicted by Figure 3.4, responsible for implementing the concurrency control approach, which was originally tangled with the other layers.

In fact, in the aspect-oriented environment we also have a sixth layer, which is orthogonal to the others, being responsible for exception handling. This sixth layer was

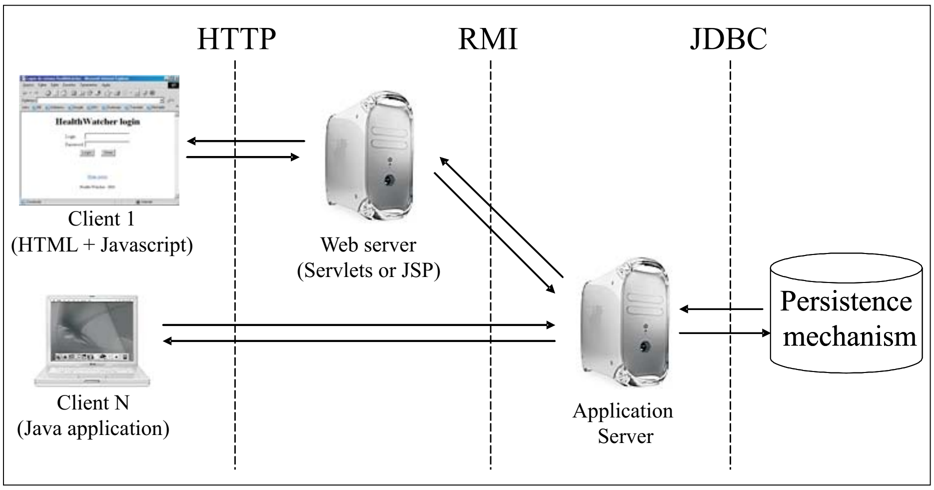

Figure 3.3: System configuration.

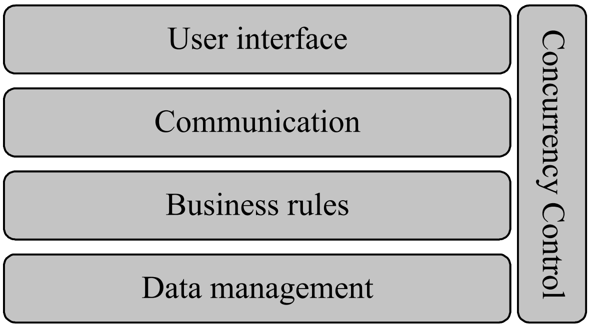

Figure 3.4: Five-layered architecture.

identified during the concerns definition and aspect-oriented analysis. It was natural to separate exceptions specific to data management, distribution, and concurrency control concerns implementation. In the core software there are only exceptions related to business rules (functional requirements). This aspect-oriented software architecture is depicted in Figure 3.5.

Instead of presenting general classes without meaning, Figure 3.6 presents a UML [10] class diagram of a fragment of the Health Watcher system with examples of each kind of class and interface of the object-oriented software identified by UML stereotypes [10]. For simplification, it only shows the classes involved in the complaint processing services, other classes essentially follow the same pattern [53]; we also omit the classes from the communication layer, which allows remote access to system services.

The Health Watcher — the information system used in our restructuring experience — 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. Complaints are registered, updated, and queried through a web client implemented using Java servlets. Accesses to the Health Watcher services are made

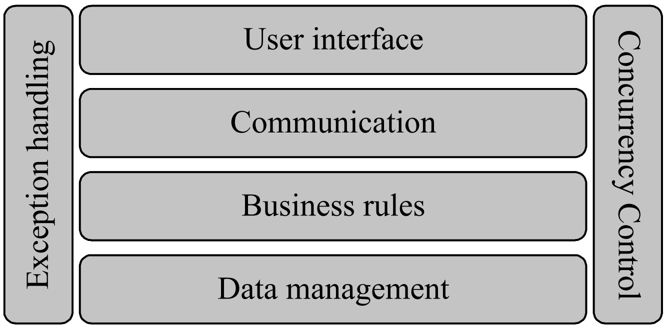

Figure 3.5: Aspect-Oriented Layered Architecture.

through its facade (`HWFacade`), which is composed of business collections. The interface `IPersistenceMechanism` abstracts which persistence mechanism is in use. Classes implementing this interface (`HWPersistenceMechanism`) should handle database connections and transaction management. Persistent data collections (`ComplaintRepository-RDBMS`) 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 (`IComplaintRepository`). These interfaces allow multiple implementations of the data collections, using different data storage and management mechanisms, including nonpersistent data structures (`ComplaintRepositoryArray`).

This architecture was originally defined elsewhere [101], having been added here just some modifications. Usually, a software contains several business collections, data collections, and basic classes.

3.3 Implementation Methods Overview

The implementation method we define is composed of two major parts:

- Implementation Guidelines — guidelines to restructure an object-oriented software to an aspect-oriented one or to implement an aspect-oriented software from the beginning.

- Implementation Activities — activities that manage the guidelines execution in the context of a software development process. In fact, we also provide changes to other activities of the development process to make them complying with the implementation activities and the aspect-oriented software development paradigm.

Figure 3.7 depicts the implementation method impact in some development activities. Some of them were changed to comply with the implementation method, and new activities were added to the implementation activities, as well as the use of specific guidelines to implement the aspect-oriented software.

The following sections present the four concerns that the implementation method supports: distribution, data management, concurrency control, and exception handling. These sections guide the concerns implementation, which are the methods implementation guidelines. Those aspects are presented by discussing the steps we performed towards restructuring the pure Java version of the system for obtaining the AspectJ

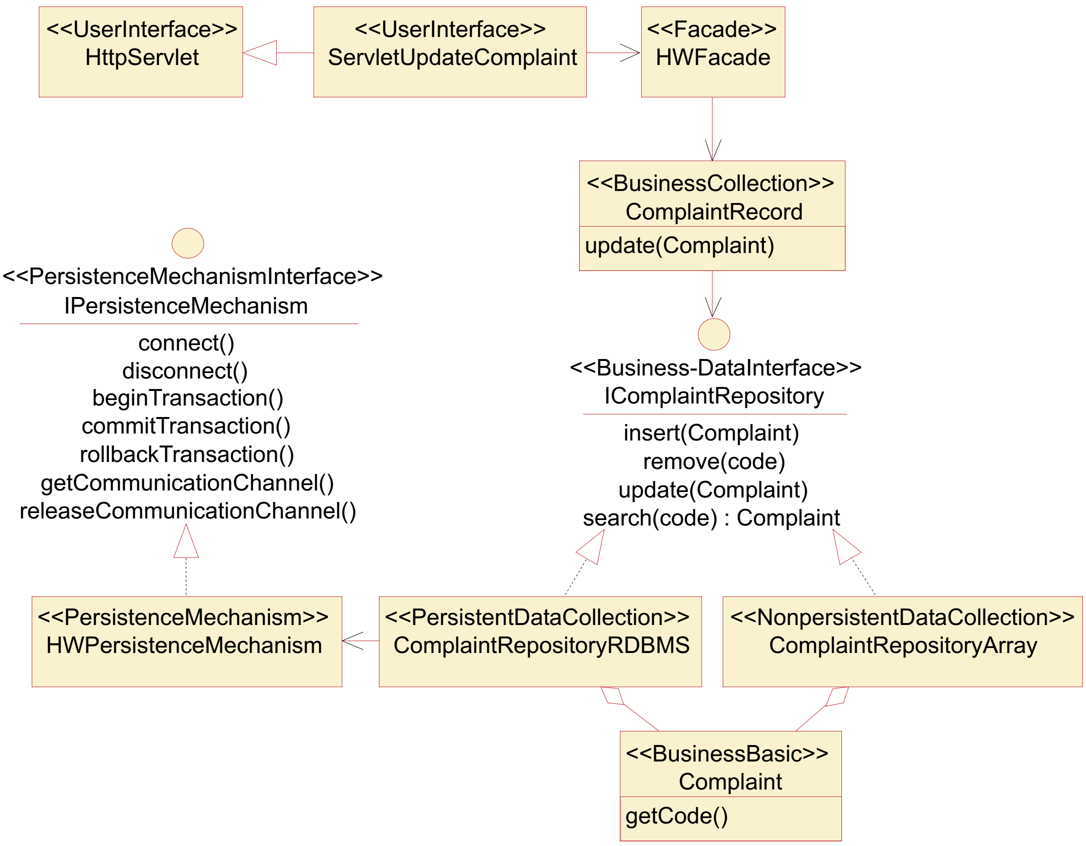

Figure 3.6: Software architecture class diagram.

version. Although those steps are not generally applicable for all kinds of applications, they can be used as specific guidelines for implementing distribution, concurrency control and data management aspects in systems that comply with the Health Watcher's software architecture. They can, therefore, be used as guidelines for restructuring such systems.

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, those abstract aspects comply with the software architecture, like all other 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.

In addition, Section 3.10 describes an alternative implementation approach to implement aspect-oriented software using the guidelines presented in this chapter. The second major methods part is presented in next chapter (Chapter 4), which defines how the guidelines should be executed in the context of a development process, and how a development process is affected by the guidelines, the alternative implementation approach, and the aspect-oriented software development.

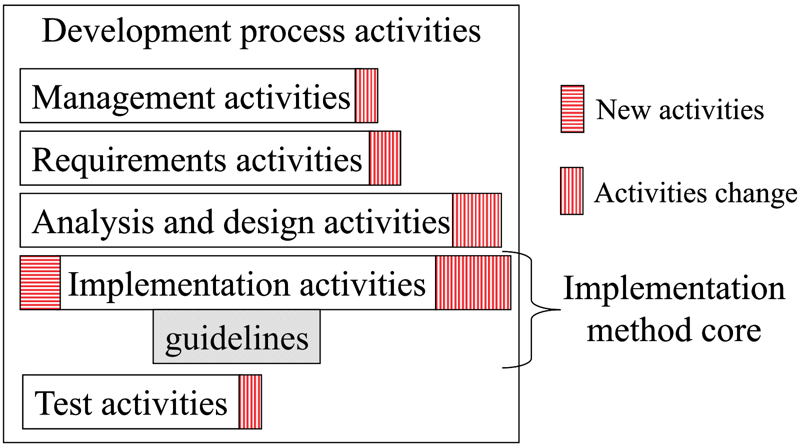

Figure 3.7: Development activities changed by the implementation method.

3.4 Distribution concern

In this section, we focus on the distribution aspects. In the next sections, we present the other aspects.

The first step of the restructuring process for separating the distribution code is to remove the RMI specific code from the pure Java version of the system. Roughly, in a system that complies with the Health Watcher's architecture, 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.

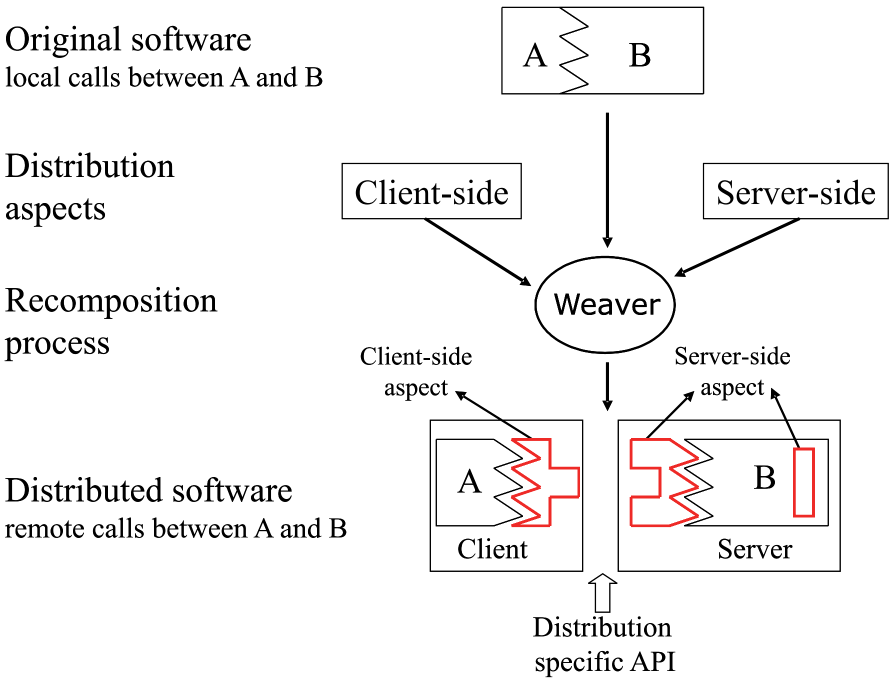

Figure 3.8: Distribution code weaving.

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 composed to the system using a composition process called weaving, resulting in the distribution version, as shown in Figure 3.8. This seems to be a common AspectJ pattern [63], 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.

3.4.1 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 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 super class (`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 facade cannot extend `UnicastRemoteObject`, we can have a similar effect using an RMI alternative. The `exportObject` static method, declared in `UnicastRemoteObject`, is 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 a name that is used to bind to the instance and to get the server machine's name where the remote instance is available.

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

Those abstract methods and pointcut are implemented by a system specific aspect. The abstract aspect also defines an `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 is despised. However, usually, the `main` method of a facade class might not even exist or it may only make 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`). This is done by using AspectJ's `declareparents` construct:

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

The `IRemoteFacade` interface is 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. 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 are defined as

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

and the pointcut as

```
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 (a method with an empty body) 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 serialize 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 `declareparents` 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 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.

3.4.2 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 `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 the `HWFacade` type, another possibility would be to intercept calls to the facade methods, directing them to the remote facade instance. This could be achieved by first defining the following pointcut to identify calls to the non-static `HWFacade` methods, as long as they originate from the user interface classes, which in our case are Java servlets:

```
pointcut facadeCalls(HWFacade local):
  target(local)         && call(* *(..))  &&
  !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 (below) 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. It uses a reference to the remote instance (`remoteHW`, declared and initialized by the aspect) to proceed with the execution flow, 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 have 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 method that updates complaints 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 code, which is tedious. 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

To allow generality and reuse, we define an abstract aspect that uses Java reflection and AspectJ features to redirect facade method 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, and therefore, can be reused in other software. 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 is responsible for handling 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 decreases 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 [26, 2] 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 had during system restructuring could have been avoided if we added 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 [43], 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 provide good compositionality properties.

Synchronizing states

When implementing the client-side aspect we have also to deal with the synchronization of object states. This is necessary because RMI supports only a copy parameter passing mechanism for non-remote arguments. Therefore, 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, by reflecting modifications to the client copies 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 3.5. This shows that the distribution and data management 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.

3.4.3 Distribution aspects class diagram

Figure 3.9 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 a different fill color identify the aspects.

3.4.4 Distribution framework

Figure 3.10 presents the abstract aspects that constitute a distribution framework that can be primarily reused to distribute user interface from facade. In fact, this framework is general enough to distribute execution from any two objects of different classes. To extend the framework, one has to identify the source and target classes, which in the specific software architecture are respectively the user interface and facade classes.

3.4.5 Distribution dynamics

Figure 3.11 depicts the distribution aspects dynamics in a UML sequence diagram. The diagram only presents the dynamics of the abstract aspects.

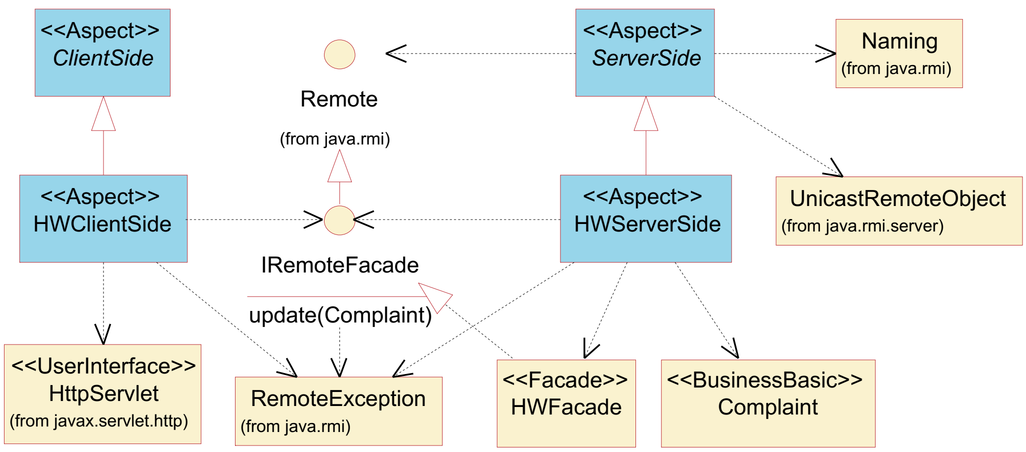

Figure 3.9: Distribution aspects class diagram.

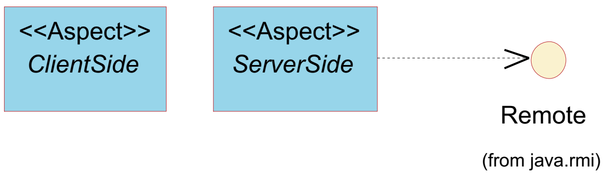

Figure 3.10: Distribution framework.

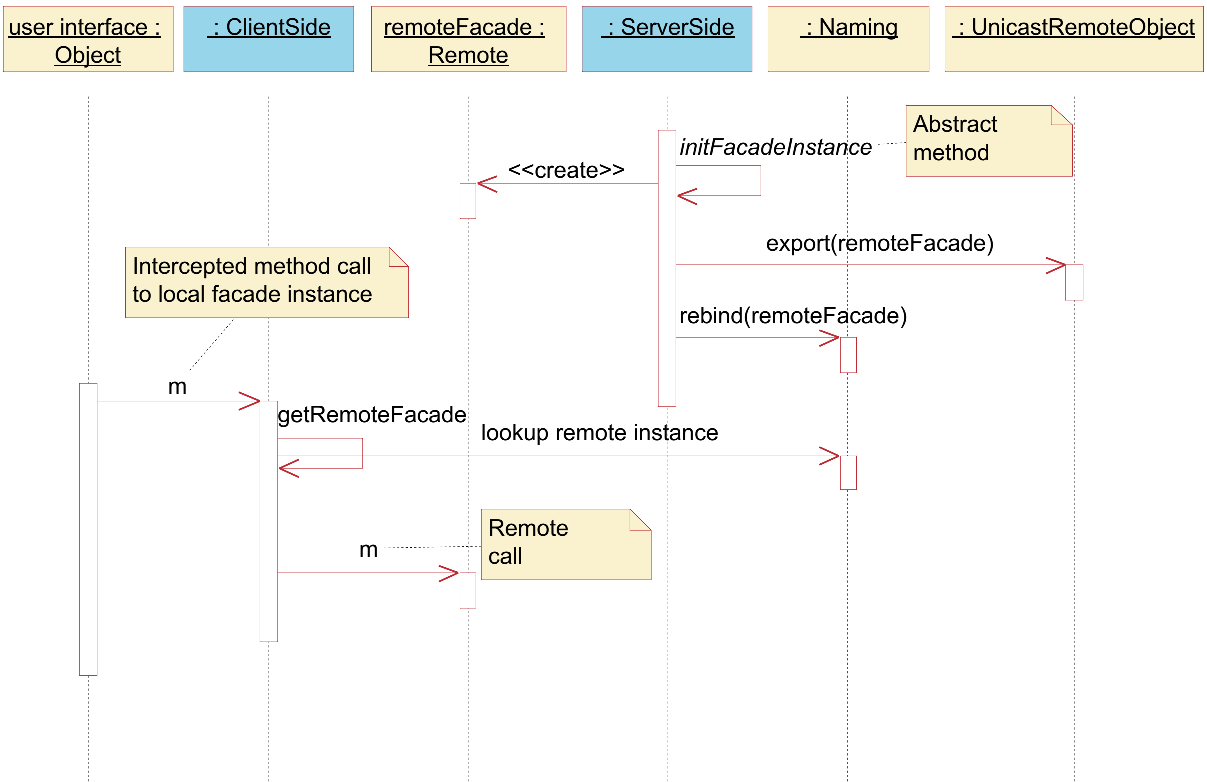

Figure 3.11: Distribution aspects dynamics.

First of all, the server-side aspect retrieves a facade instance through an abstract method, exports it through `UnicastRemoteObject` class, making it available to be remotely accessed, and binds the instance to a name using the `Naming` class. The lookup service uses this name in order to access the remote instance.

After that, the remote instance is ready to be accessed. The client-side aspect can retrieve the remote instance using the lookup service and redirect to it any method call made by user interface objects, which originally would be made to a local facade instance.

3.5 Data management concern

This section presents the steps that we followed in order to restructure the persistence code of the Health Watcher system and obtain the corresponding data management aspects. The first step in this direction is to remove the data management code from the pure Java version of the system. In a system that complies with the Health Watcher's architecture, data management 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 data management code should be removed from the pure Java system and a similar functionality should be implemented as aspects. Figure 3.12 illustrates that and also shows that we have aspects for configuring the system to run with persistent and nonpersistent data management. As discussed in Chapter 4, 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 3.2) is responsible for abstracting what data management medium is being used by the business collection classes, and was not factored out from the core source code. The data management 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, for improving performance, and synchronization of object states with the corresponding database entities, for ensuring consistency.

3.5.1 Persistence mechanism control

This section describes the persistence mechanism control aspects, which 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 code) and deal with database initialization, connection handling, and resources releasing, services provided through the created instance.

For reuse purposes, this concern is implemented using an aspect hierarchy composed of an abstract aspect and a concrete aspect. The second is specific to the Health Watcher

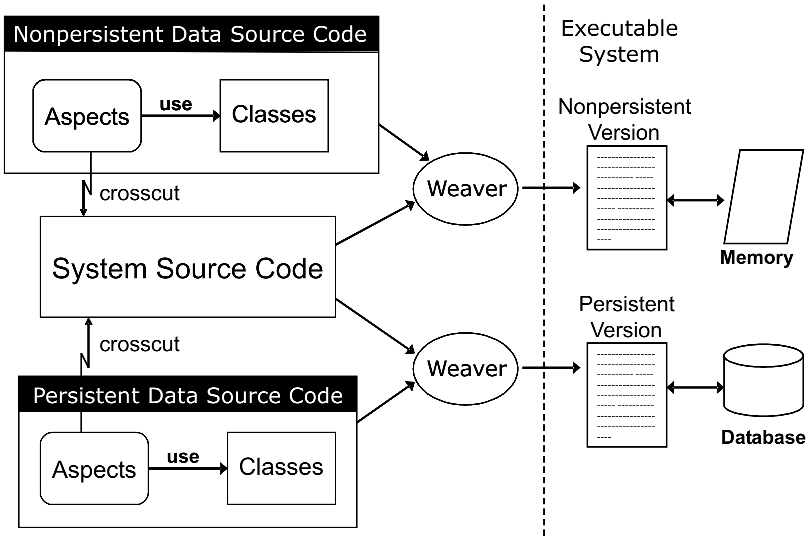

Figure 3.12: Data management code weaving.

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 advices 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 are 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 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 is executed usually before or after some facade code, during system initialization time. Therefore those exceptions are declared as *soft*, not checked, by the declaresoft static crosscutting AspectJ construct. Section 3.8 presents 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 are concretely defined for the Health Watcher system in the following aspect.

```
aspect HWPersistenceControl extends PersistenceControl {
  pointcut initSystem(): call(HWFacade.new(..));
  IPersistenceMechanism pmInit() {
    return HWPersistenceMechanism.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, HWPersistenceMechanism, 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(* HWPersistenceMechanism.getInstance(..));
declare soft: PersistenceMechanismException: obtainPmInstance();
```

The `obtainPmInstance` is just an auxiliary pointcut to be used in the `declare soft` constructor.

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

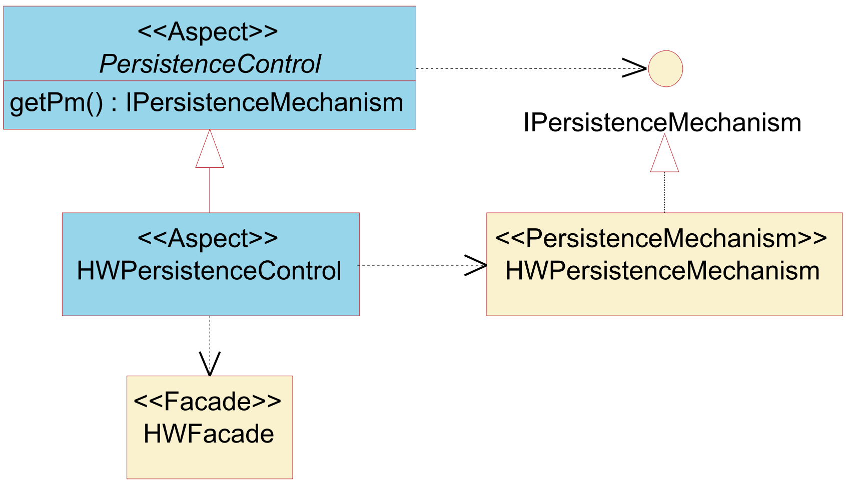

Figure 3.13: Persistence control aspects class diagram.

3.5.2 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 [20]: 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 is done when implementing 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 TransactionControl {
  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 invoking the transaction services supported by the persistence mechanism.

The abstract transaction control aspect also implements three advices 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 `afterreturning` advice that commits the transaction when the method executions returns successfully:

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

At last, an `afterthrowing` 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 have the same behavior in the pure Java version of the Health Watcher system. This decision is perfectly adequate for both versions of the system and, in fact, 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 [43].

In this specific case of transactions, this does not bring 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 (`HWTransactionControl`) 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 affect aspect legibility making the aspect code too much dependent on modifications in 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 `declareparents` 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 are executed with the signature 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. With this solution, the concrete transaction aspect is dependent on the concrete persistence control aspect, however 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 completely 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 more modular 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.

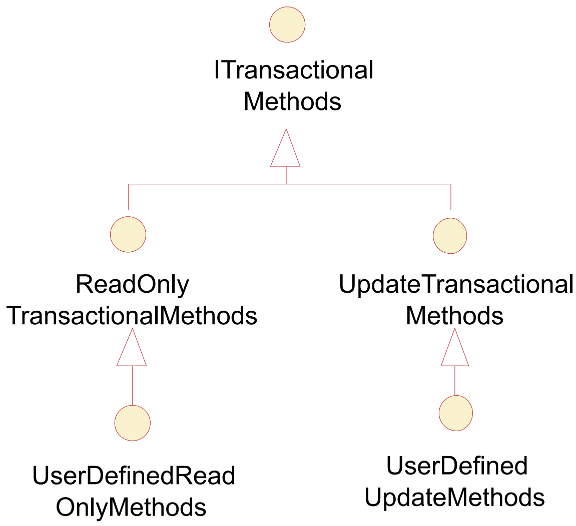

Figure 3.14: Transactional methods hierarchy.

Implementing alternative policies

The aspects illustrated in this section offer a uniform transaction control policy, which is 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 [43]. 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 3.14 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 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 `afterreturning`, and an `afterthrowing` 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 transaction advices 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 impact the others. However, these extreme situations are not usual. In fact, 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.

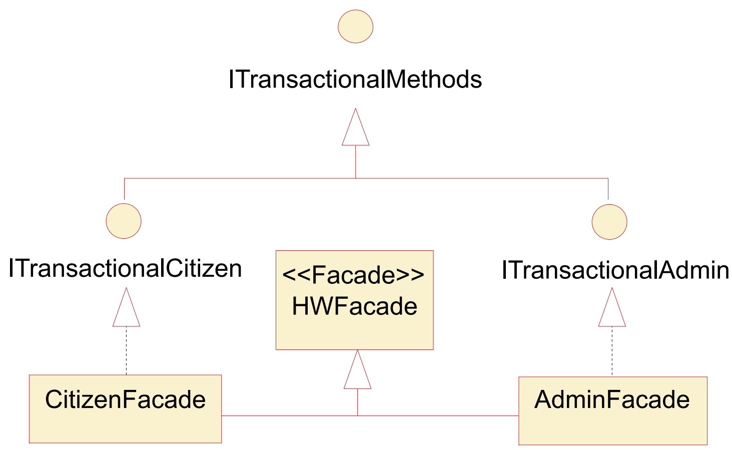

Figure 3.15: Example of multiple transactional components.

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 3.15 shows the UML class diagram for this hierarchy.

Besides having the extra interfaces, we should extend the concrete `HWTransaction–Control` 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 3.16 presents a class diagram of the transaction control aspects and the Health Watcher classes and interfaces the aspects affect or use.

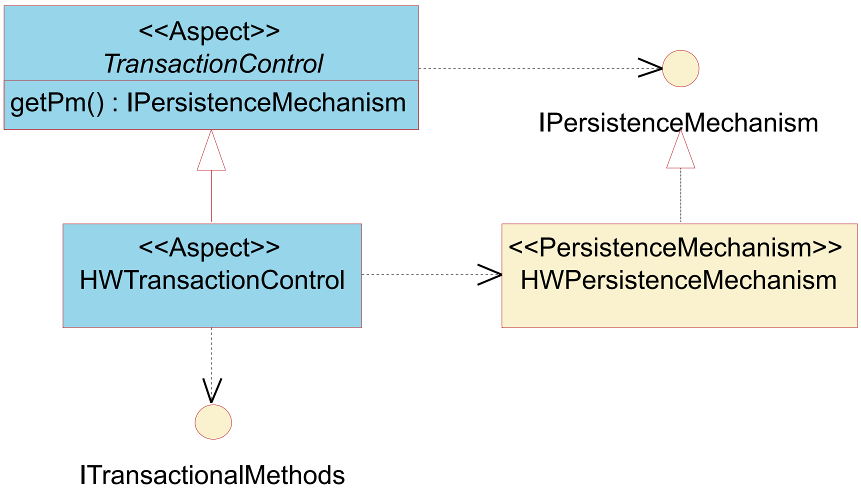

Figure 3.16: Transaction control aspects class diagram.

3.5.3 Data collection customization

As explained before, the Health Watcher system should also work using nonpersistent data. In order to support this, two aspects should be 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 3.12.

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 super class of the business collection 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 on the information of what class (business collection) is being created in the affected join point. Note that this aspect uses information from the execution context through the variable `thisJoinPoint` [97].

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

Implement an abstract system-dependent 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 `HWDataCollection–Customization` identifies the system business collection classes by subclassing `System–Record`:

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

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

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

Note that the `getSystemRecord` method uses an abstract method `getComplaintRepo–sitory` that is responsible for creating the data collection to be used by the business collection. We omit the code for the other business collections, but it is similar to the `ComplaintRecord` code. 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 discussion on concrete aspects exemplifies 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. 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 `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 `NonPersis–tentDataCollection` aspects when weaving.

As discussed in the Persistence mechanism control section, 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 3.17 presents a class diagram of the data collection customization aspects and the Health Watcher classes and interfaces the aspects affect or use.

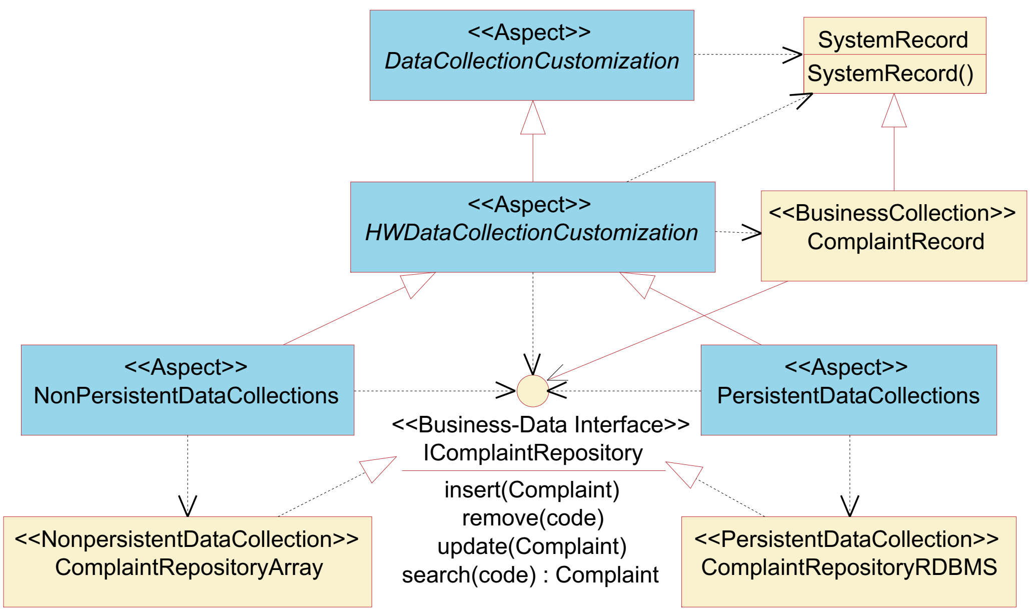

Figure 3.17: Data collection customization aspects class diagram.

3.5.4 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 identification 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 is 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 (`PartiallyLoad`) to identify the servlets that generate web pages listing partial information about several objects of a class. 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. Therefore, we must define a pointcut matching the execution of those methods:

```
aspect ParametrizedDataLoading {
  private interface PartiallyLoad { }
  declare parents: ServletSearchHealthUnit implements PartiallyLoad;
  pointcut partialLoadingServlets():
    this(PartiallyLoad+) && execution(* do*(..));
```

We should add more servlets in this pointcut to identify all servlets that should generate a page with partial information.

Applying the adequate loading level

After specifying that the subtypes of `PartiallyLoad` 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 (`HealthUnitData– RDBMS`):

```
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 `PartiallyLoad` 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 is distributed. A solution for that is discussed in Section 3.9.

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.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 `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.

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 3.5.2, 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 3.18 presents a class diagram of the data access on demand aspect and the Health Watcher classes and interfaces the aspects affect or use.

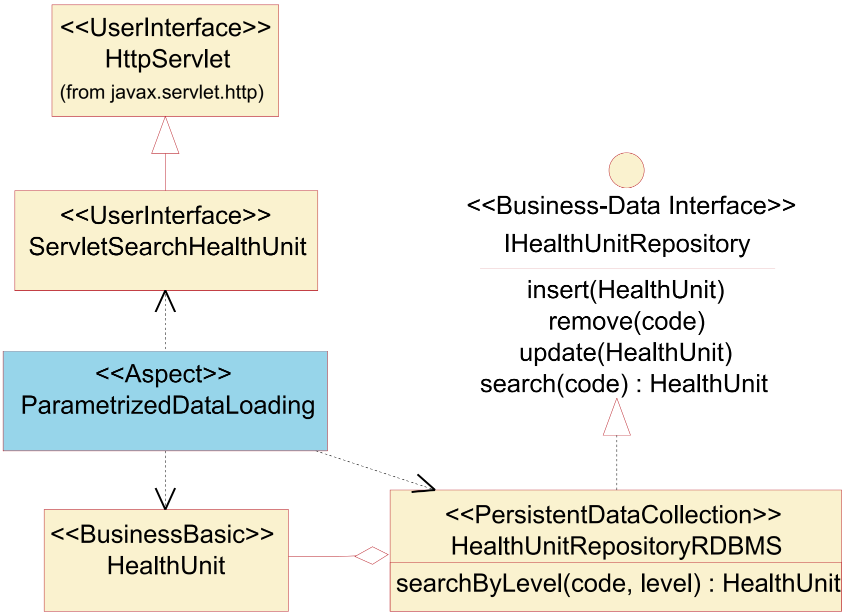

Figure 3.18: Data access on demand aspects class diagram.

3.5.5 Data management framework

Figure 3.19 presents the data management framework. Those abstract aspects can be reused to implement data management in several applications that comply with the Health Watcher's software architecture.

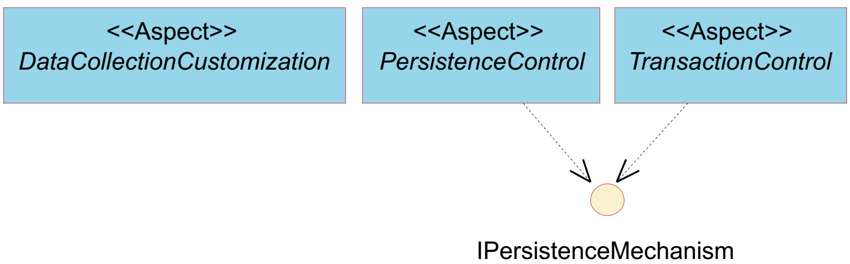

Figure 3.19: Data management framework.

3.5.6 Data management dynamics

Figure 3.20 presents the dynamics of the data management reusable aspects.

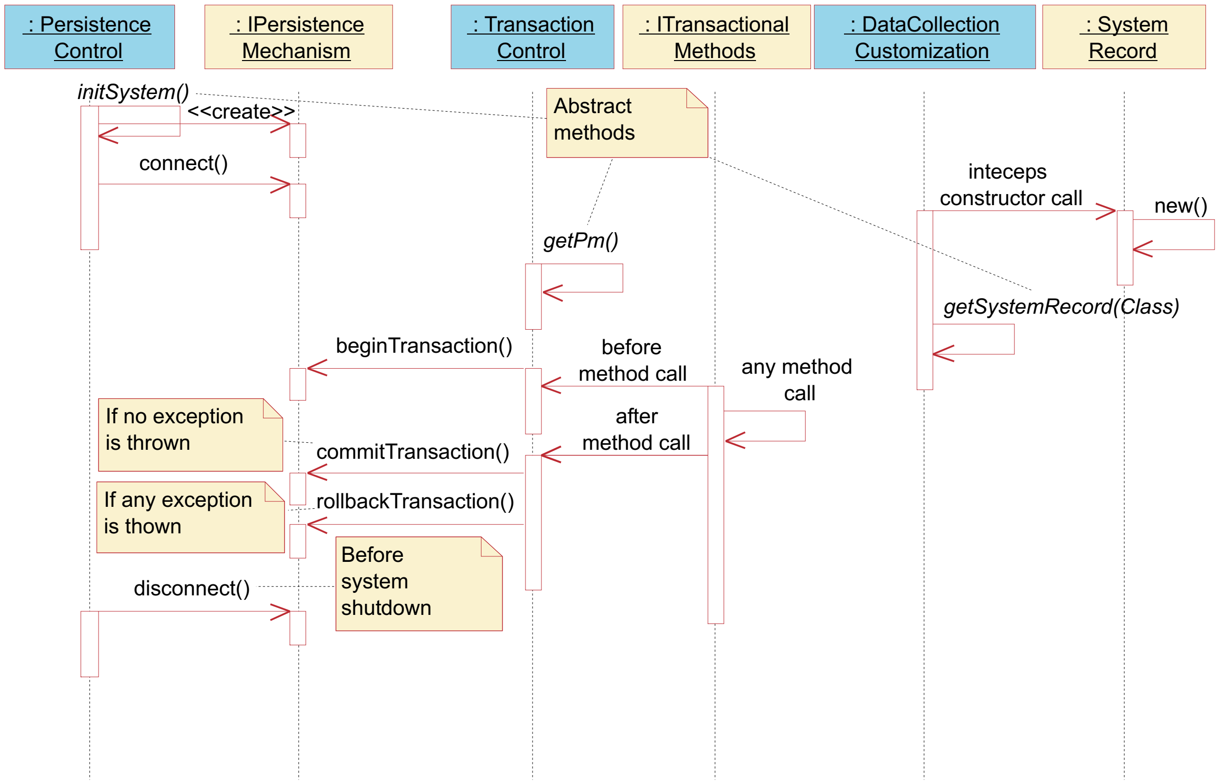

Figure 3.20: Data management dynamics.

The aspect responsible for data collection customization intercepts records creation in order to create them with the data collection specified by the concrete subclasses through the implementation of the `getsystemRecord` abstract method. If persistence aspects are selected, the one responsible for persistence control has to create and connect to the persistence mechanism when the software starts, which is defined through the implementation of abstracts pointcut and method, and disconnect to the persistence mechanism before software shutdown. The transaction control aspect calls `IPersistenceMechanism` methods in order to start, finish, and roll a transaction back, when a selected facade method, from `ITransactionalMethods` interface, is called.

3.6 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 3.4. 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 from the business and user interface classes 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 user interface code, satisfying both persistence and distribution requirements. This aspect is separately defined from data management and distribution to be used when only one of these (distribution or persistence) is woven into software. For example, Chapter 4 proposes gradual tests, where distribution might be tested without persistence and vice-versa, which demands using the update state aspect in both situations.

Identifying classes whose objects must be updated

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

```
aspect UpdateStateControl
    percflow(UpdateStateControl.servletService()) {
  private interface SychonizableObject {
    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 `declareparents` construct to make the class `Complaint` subtype of `SychonizableObject`.

```
  declare parents: Complaint implements SychonizableObject;
```

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(SychonizableObject 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(SychonizableObject 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(SychonizableObject 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 finishes, the aspect can finally introduce the synchronization calls to reflect the updates in the database. This is specified by the following pointcut

```
pointcut remoteExecution():
  if(UpdateStateControl.aspectOf().hasDirtyObjects()) &&
  servletService();
```

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:

```
after() returning: remoteExecution() {
  Iterator it = remoteDirtyObjects.iterator();
  while (it.hasNext()) {
    SychonizableObject o = (SychonizableObject) 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 3.5.3).

For simplicity, we omitted the declarations for other 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 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 is also 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 is insignificant. In more complex systems, dealing with several complex objects, we suspect it might not be worth to separate the state 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 client-side distribution aspect section, 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 have to generalize the definition of the `facadeCalls` pointcut in such a way that it includes new join 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. Therefore, careful design activities should be performed before implementation, avoiding rework, although that is 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.

In our previous implementation [91], 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 3.21 presents a class diagram of the data state synchronization control aspect including the Health Watcher classes and interfaces the aspect affects or uses.

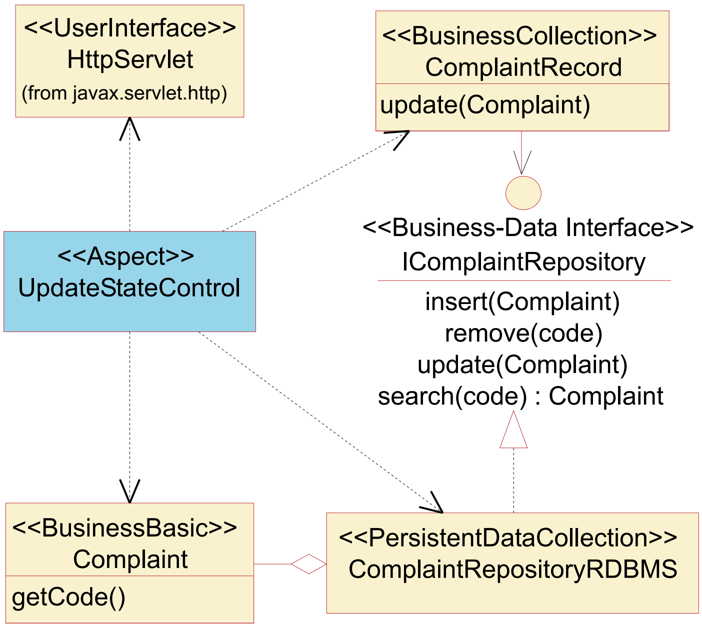

Figure 3.21: Update state control aspects class diagram.

3.7 Concurrency control concern

Concurrent environments have a great complexity inserted by their non-determinism, which can lead the system to an inconsistent state, under abnormal interference. By using aspect-oriented programming, we can separate concurrency control from other concerns, such as business rules, data management and user interface. This separation makes it easier to change concurrency control policies, since concurrency control code is not tangled with other concerns code. Therefore, besides increasing software modularity, this separation also decreases complexity, since there is not concurrency control to reason about when implementing business, data management and user interface.

In this section, we present guidelines for restructuring object-oriented software, by removing concurrency control code that is tangled and spread over software units and implementing aspects to modularize such a control. In fact, by ignoring the steps to remove the concurrency control from object-oriented software, the restructuring guidelines can be used for aspect-oriented software development, like the other concern's guidelines.

These aspect-oriented restructuring/development guidelines are complementary to a concurrency control implementation method [84, 81, 80]. This method is also tailored to the same specific software architecture and provides guidelines for implementing concurrency control in object-oriented software. In fact, the restructuring guidelines provide an aspect-oriented implementation to the concurrency control defined by the implementation method and use the method's analyses to identify what concurrency control mechanism and where to use it in each case.

3.7.1 Identifying concurrency control code

Examples of concurrency control code are:

- `synchronized` Java modify [49, 27] — synchronized methods of an object cannot be concurrently executed;

- `synchronized` Java block [49, 27] — synchronized blocks use an object lock to define a mutual-exclusion region where a thread must acquire the lock before executing the block;

- calls to `wait`, `notify`, and `notifyAll` methods [49, 27] — used to implement semaphore-like behavior;

- use of Concurrency Manager design pattern [82] — optimistic alternative that synchronizes only potential conflicting executions based on the methods semantics, in contrast to a pessimistic approach of `synchronized` Java modify that synchronizes every method execution;

- use of a timestamp technique — avoids undesirable update of objects copies that might lead the system to an inconsistent state.

However, as previously mentioned, a previous work [84, 81, 80] defined a concurrency control implementation method. This method precisely defines which concurrency control is needed and what modules of the software need to be controlled. Therefore, we can also use the method's definition to identify where the concurrency control code is.

After implementing and analyzing several systems with the software architecture we use, we could identify which concurrency controls are most frequently applied in the classes of the software architecture.

Usually there is no concurrency control in user interface and communication layers. In business layer there might be concurrency control in the facade class to guarantee the implementation of transactions if using persistent data management. Therefore, it is natural to leave such implementation to data management aspects, which, in fact, is already accomplished by aspects defined in Section 3.5.2.

However, there might be concurrency control in business collections, where some methods use the Concurrency Manager design pattern [82] to avoid undesirable interferences. Some business collections methods that do not use the concurrency manager might also have some synchronized methods. This happens when the system has few simultaneous users accessing the system and these methods are lightweight [84, 81].

Another place that generally should have concurrency control code is the `update` method of the data collections classes, which are synchronized, if implementing the timestamp technique [84, 81]. This technique solves problems in concurrent updates of object copies. Usually, the approach for implementing database access is to return a new copy of an object every time an object is requested through the data collection. If two different requests (threads) retrieve two copies of the same object and change these copies, the system state might become inconsistent if both threads try to update their copies. The solution also adds a timestamp field to all objects, usually of basic class type, that cannot be concurrently updated; an object can be updated only if there is not a newer copy of it already stored in the database. The data collections classes implementing timestamp also have to manage the new field added in the basic objects; therefore `insert`, `search`, and `update` methods should have additional code, for example SQL code, to handle the field. More details can be found elsewhere [84, 81, 80].

In the basic classes, the concurrency control applied is to implement the timestamp technique, as mentioned above, since the basic objects are not concurrently accessed, due to the persistent data collection implementations in many systems that use relational databases. This is our alternative to intuitive controls that tend to synchronize and to implement transactions in all the facade methods.

3.7.2 Removing concurrency control code

After identifying where concurrency control code is, we must remove it. We should also document where and which kind of concurrency control the software has in order to implement it later with aspects. A summary of where concurrency control might be is as follows:

- User interface classes — no concurrency control;
- Communication classes — no concurrency control;
- Facade class — transactions if in persistent environment. However, transactions are already implemented by persistence aspects;
- Business collections classes — Concurrency Manager [82] or `synchronized` modifier;
- Data collections classes — `synchronized` modifier and additional SQL commands in the methods of the persistent collections to implement timestamp;
- Basic classes — timestamp field and related methods.

The next step defines aspects to implement, in a modular way, concurrency control we removed from these classes.

3.7.3 Implementing concurrency control aspects

As previously mentioned, there are five types of concurrency control: `synchronized` modify, `synchronized` block, use of Concurrency Manager design pattern, use of timestamp technique, and calls to `wait`, `notify`, and `notifyAll` methods.

However `wait`, `notify`, and `notifyAll` methods are not used by the implementation method. Therefore, we implemented aspects for the other four types of concurrency control.

Implementing reusable aspects

In order to allow aspects reuse we defined abstracts aspects that constitute a simple aspect framework. They can be extended for implementing concurrency control in other applications that comply with the layered architecture presented in Section 3.2.

The first abstract aspect is responsible for identifying join points that cannot execute concurrently, and therefore, should be synchronized.

```
abstract aspect Synchronization {
  protected abstract pointcut synchronizationPoints(Object syncObj);
}
```

Besides identifying synchronization join points, the `synchronizationPoints` pointcut also receives the object to be used by the chosen synchronization technique. The following abstract aspect extends the `Synchronization` aspect and defines a synchronization approach similar to the `synchronized` modifier or block.

```
abstract aspect PessimisticSynchronization extends Synchronization {
  Object around(Object syncObj): synchronizationPoints(syncObj) {
    synchronized(syncObj) {
      return proceed(syncObj);
    }
  }
}
```

The `around` advice uses the inherited pointcut to synchronize any join point, which should be methods execution, using the `synchronized` block with the specified object. To implement the `synchronized` block behavior, each join point to be synchronized must specify the object whose lock should be used by the synchronization policy. On the other hand, to implement the `synchronized` modifier behavior each join point to be synchronized should also expose the currently executing object (using `this` designator), whose lock is used by the synchronization policy, which is the semantics of the `synchronized` method modifier [49, 27].

As an alternative to this pessimistic approach, a third abstract aspect implements the use of the Concurrency Manager design pattern [82].

```
abstract aspect OptimisticSynchronization extends Synchronization
    perthis(synchronizationPoints(Object)) {
  private ConcurrencyManager manager = new ConcurrencyManager();
```

Besides extending the `Synchronization` aspect, the `OptimisticSynchronization` aspect has to declare a `perthis` clause that creates an aspect instance associated with each object that is the currently executing object at any join point in the `synchronization-Points` pointcut. This is necessary because there must be a concurrency manager instance for each currently executing object at any join point. The aspect also defines advices to provide the concurrency manager behavior. Since the design pattern should use the semantics of the operation in order to synchronize only conflicting executions, the advices use a `getKey` abstract method in order to retrieve the `String` to be used as the blocking key.

```
    before(Object syncObj): synchronizationPoints(syncObj) {
      Object key = this.getKey(syncObj);
      manager.beginExecution(key);
    }
    after(Object syncObj): synchronizationPoints(syncObj) {
      Object key = this.getKey(syncObj);
      manager.endExecution(key);
    }
    protected abstract Object getKey(Object syncObj);
  }
```

The `before` advice calls the `beginExecution` method of the `ConcurrencyManager` class with an object key. This object key is the argument used by the concurrency manager to identify if this execution might conflict with any other execution, and therefore block it until the potentially conflicting execution finishes, which is notified by the `after` advice. Execution conflicts are the ones that use the same `String` as key. The `getKey` method should be defined in a concrete aspect together with the `synchronizationPoints` pointcut defining the semantics of the synchronization.

The next reusable concurrency control aspect is responsible for implementing the timestamp technique. The abstract aspect defines two auxiliary interfaces to identify basic classes and data collections that are affected by the aspects.

```
  abstract aspect Timestamp {
    interface TimestampedRepository {
      void updateTimestamp(TimestampedType object);
      long searchTimestamp(TimestampedType object);
    }
    interface TimestampedType {
      long getTimestamp();
    }
    private static final String MESSAGE = ...;
```

The aspect also declares a constant to be used in concurrency exceptions. The following piece of code adds, in the subtypes of `TimestampedType` interface, a `timestamp` field, and methods responsible for managing the field, using inter-type declaration mechanism.

```
    private long TimestampedType.timestamp;
    public long TimestampedType.getTimestamp() {
      return timestamp;
    }
    private void TimestampedType.setTimestamp(long timestamp) {
      this.timestamp = timestamp;
    }
```

A pointcut is defined to identify affected data collections `search` methods, in order to load the object timestamp after successfully (without raising an exception) retrieving the object from the repository, which is implemented by the `after returning` advice. The pointcut uses the designator `this` to expose the data collection, which is the currently executing object.

```
private pointcut managedSearchMethods(TimestampedRepository rep):
  execution(TimestampedType search(..)) && this(rep);

after(TimestampedRepository rep) returning(TimestampedType object):
    managedSearchMethods(rep) {
  long timestamp = rep.searchTimestamp(object);
  object.setTimestamp(timestamp);
}
```

This advice exposes the returned object and uses the searchTimestamp method to retrieve the object's timestamp. The next piece of code is responsible for guaranteeing the timestamp storage into the repository after successfully inserting the object.

```
pointcut managedInsertMethods(TimestampedRepository rep,
                              TimestampedType object):
  execution(void insert(TimestampedType)) &&
  this(rep) && args(object);

after(TimestampedRepository rep, TimestampedType object) returning:
    managedInsertMethods(rep, object) {
  rep.updateTimestamp(object);
}
```

The managedInsertMethods pointcut and its related advice are very similar to the previous pointcut and advice. After that, the abstract aspect defines a pointcut to identify affected data collections update methods

```
pointcut managedUpdateMethods(TimestampedRepository rep,
                              TimestampedType object):
  execution(void update(TimestampedType)) &&
  this(rep) && args(object);
```

and an advice to update the timestamp information if the object is successfully updated. Note the use of the synchronized block in the advice; this is necessary in order to guarantee serialization, for each data collection (repository), during timestamp checking [84, 81, 80].

```
void around (TimestampedRepository rep, TimestampedType object):
    managedUpdateMethods(rep, object) {
  synchronized(rep) {
    long timestamp = rep.searchTimestamp(object);
    if (object.getTimestamp() == timestamp) {
      object.setTimestamp(timestamp + 1);
      proceed(rep, object);
      rep.updateTimestamp(object);
```

if the timestamp object is different from the timestamp stored in the data collection, a concurrency control exception is raised

```
      } else {
        Exception ex;
        ex = new ConcurrencyControlException(MESSAGE);
        throw new SoftException(ex);
      }
    }
  }
}
```

Note that the exception is wrapped into an unchecked exception (`SoftException`). This unchecked exception should be handled in the user interface in order to show a message to the user, which is carried out by other aspects defined in Section 3.8.

Specializing abstract aspects

In order to implement concurrency control in a specific software, we should define concrete aspects to identify where and which concurrency control must be used. If this step is being performed as part of a restructuring process, we should use the information documented in the "Removing the concurrency control code" step in order to apply the same concurrency control applied before. Otherwise, we should apply the concurrency control implementation method [84, 81, 80] analysis to identify where and which controls to be used. In fact, it might be necessary to apply the concurrency control method even if performing a restructuring process in a software originally controlled; if that control was performed without using the method, and therefore, might not be safe.

Section 3.10 and Chapter 4 presents a progressive approach that supports both implementation and testing where persistence, distribution, and concurrency control are progressively implemented and maybe tested independently and with several combinations. In this way, we might provide concurrency control for nonpersistent data collection in order to support testing the system with distribution (multi-user environment) but without persistence.

As defined in the concurrency control implementation method [84, 81, 80] all methods of a nonpersistent data collection should be synchronized if concurrently used. Therefore, we defined the following aspect extending the `PessimisticSynchronization` aspect identifying which methods of which classes should be synchronized by implementing the `synchronizationPoints` pointcut.

```
aspect HWPessimisticSynchronization
    extends PessimisticSynchronization {
  private interface SynchronizedClasses {};
  declare parents: EmployeeRepositoryArray ||
                   ComplaintRepositoryArray
                   implements SynchronizedClasses;
  pointcut synchronizationPoints(Object syncObj):
    execution(* SynchronizedClasses+.*(..)) && this(syncObj);
}
```

The aspect defines an auxiliary interface that subtypes `EmployeeRepositoryArray` and `ComplaintRepositoryArray` classes through the use of the `declare parents` clause. To synchronize any other classes we just have to add them in the `declare parents` clause. Note that this aspect is responsible for synchronizing all methods of the `Synchronized–Classes` subtypes. When using those classes in a persistent environment, they are no longer be concurrently accessed, and therefore, we can easily unplug this aspect.

Another instance of concurrency control we implemented in the Health Watcher software uses the Concurrency Manager design pattern to avoid undesirable interferences when registering employees. The race condition occurs only when registering two employees with the same login. The `HWOptimizedSynchronization` aspect defines where the Concurrency Manager design should be applied by implementing the `synchronizationPoints` pointcut. The pointcut should also expose an object that has the semantics to be used in order to synchronize only potential conflicting execution.

```
aspect HWOptimisticSynchronization
    extends OptimisticSynchronization {
  protected pointcut synchronizationPoints(Object syncObj):
    execution(void EmployeeRecord.insert(Employee)) && args(syncObj);
```

The semantics to synchronize only potential conflicting executions is defined in the `getKey` method definition. In this case the concurrency manager uses the employee's login to avoid concurrent executions of the `insert` method only if they try to insert objects with the same login.

```
  protected Object getKey(Object syncObj) {
    Object response = null;
    if (syncObj instanceof Employee) {
      response = ((Employee)syncObj).getLogin();
    }
    return response;
  }
}
```

The body of this method should be modified every time a new object type is exposed in the `synchronizationPoints` pointcut in order to identify its synchronization key.

The last concurrency control applied in the Health Watcher software is to implement the timestamp technique. The `HWTimestamp` aspect defines the basic classes and their data collection, usually persistent, which have to implement the technique.

```
aspect HWTimestamp extends Timestamp {
  declare parents : Complaint implements TimestampedType;
  declare parents : ComplaintRepositoryRDBMS
                  implements TimestampedRepository;
  public void ComplaintRepositoryRDBMS.
      updateTimestamp(TimestampedType obj) {
    // update the object's timestamp in the complaint table
  }
  public long ComplaintRepositoryRDBMS.searchTimestamp(
                                         TimestampedType obj) {
    // retrieve the object's timestamp from the complaint table
  }
}
```

The aspect uses inter-type member declaration to implement `updateTimestamp` and `searchTimestamp` methods into the selected data collections. The methods implementation must access the persistent media to update and retrieve the object's timestamp information.

An alternative timestamp implementation

These timestamp aspects have additional database calls for each `insert`, `search`, and `update` method of the managed data collections. Since this extra overhead might be prohibitive, an alternative to the aspect previously defined is provided.

The following `OptimizedTimestamp` aspect is an alternative to `Timestamp` aspect. In common, they have two interfaces and the inter-type member declarations to add timestamp and related methods into timestamped objects. However, the `TimestampedRepository` interface has no methods in this alternative version. The interface is used only to identify what data collections the aspect affects.

```
abstract aspect OptimizedTimestamp {
  interface TimestampedRepository { }
  // inter-type member declarations to add timestamp
  // and related methods
```

This aspect has the same goal as the `Timestamp` aspect. It affects execution of `search`, `insert`, and `update` methods in order to retrieve, add and update, if consistent, the timestamp data. However, in order to avoid the additional calls made in the `Timestamp` aspect implementation, this aspect affects the executions of the JDBC types responsible for accessing the database and exposing their SQL commands.

```
  private pointcut executeQueryCall(String sql):
    call(ResultSet Statement.executeQuery(String)) &&
    args(sql);

  private pointcut executeUpdateCall(String sql):
    call(int Statement.executeUpdate(String,..)) &&
    args(sql);

  private pointcut managedSearchMethods(String sql):
    withincode(TimestampedType TimestampedRepository+.search(..)) &&
    executeQueryCall(sql);
```

These pointcuts affect calls to `Statement`'s `executeQuery` and `executeUpdate` methods of `TimestampedRepository` subtypes exposing their SQL commands. Those methods are used to select and either insert or update data in the database, respectively.

The following advice changes the exposed SQL command before retrieving data from the database in order to add the timestamp information and to retrieve it after the command execution. For simplicity, we omit the code responsible for manipulating the SQL string.

```
ResultSet around(String sql) throws SQLException:
    managedSearchMethods(sql) {
  ResultSet result = null;
  long timestamp;
  // code to change the sql String adding the timestamp information
  result = proceed(sql);
  result.next();
  timestamp = result.getLong("TIMESTAMP");
  result.beforeFirst();
  this.putTimestamp(timestamp);
  return result;
}
```

The `putTimestamp` method call stores the timestamp value in a hashtable using the executing thread as the hashtable key.

```
private Hashtable timestamps;
private void putTimestamp(long timestamp) {
  timestamps.put(Thread.currentThread(), new Long(timestamp));
}
private long getTimestamp() {
  Long l = (Long) timestamps.get(Thread.currentThread());
  return l.longValue();
}
```

There is also a `getTimestamp` method that retrieves the timestamp that the current executing thread stored, which is used by the following advice to update the searched object with the timestamp data.

```
after() returning(TimestampedType object):
    execution(TimestampedType TimestampedRepository+.search(..)) {
  object.setTimestamp(this.getTimestamp());
}
```

Similary, there is an advice to insert the timestamp information

```
int around(String sql) throws SQLException:
    managedInsertMethods(sql) {
  // code to change the sql String in order to add
  // the timestamp information to be inserted in
  // the database for the first time
  return proceed(sql);
}
```

and an advice to update the timestamp, which is responsible for applying the timestamp logic, by only updating objects consistently. If an inconsistency is identified, an exception is raised, as in the `Timestamp` aspect.

```
    int around (TimestampedType object, String sql) throws SQLException:
        managedUpdateMethods(object, sql) {
      long timestamp = object.getTimestamp();
      object.setTimestamp(timestamp + 1);
      // code to change the sql String in order to add the timestamp
      // information and the update condition of the timestamp technique
      int result = proceed(object, sql);
      if (result == 0) {
        Exception ex = new ConcurrencyControlException(MESSAGE);
        throw new SoftException(ex);
      }
      return result;
    }
  }
```

This alternative to the first timestamp aspect is less general, since the aspects are specific to data collections that use JDBC to access relational databases. In fact, the current implementation affects directly the SQL command to be submitted to the database and does not support the use of `PreparedStatement` [103, 57]. On the other hand, our first solution is technology-independent, despite generating database access overhead.

Concurrency control aspects class diagram

Figures 3.22 and 3.23 depict, respectively, how the synchronization and timestamp aspects affect the classes of the Health Watcher's software architecture.

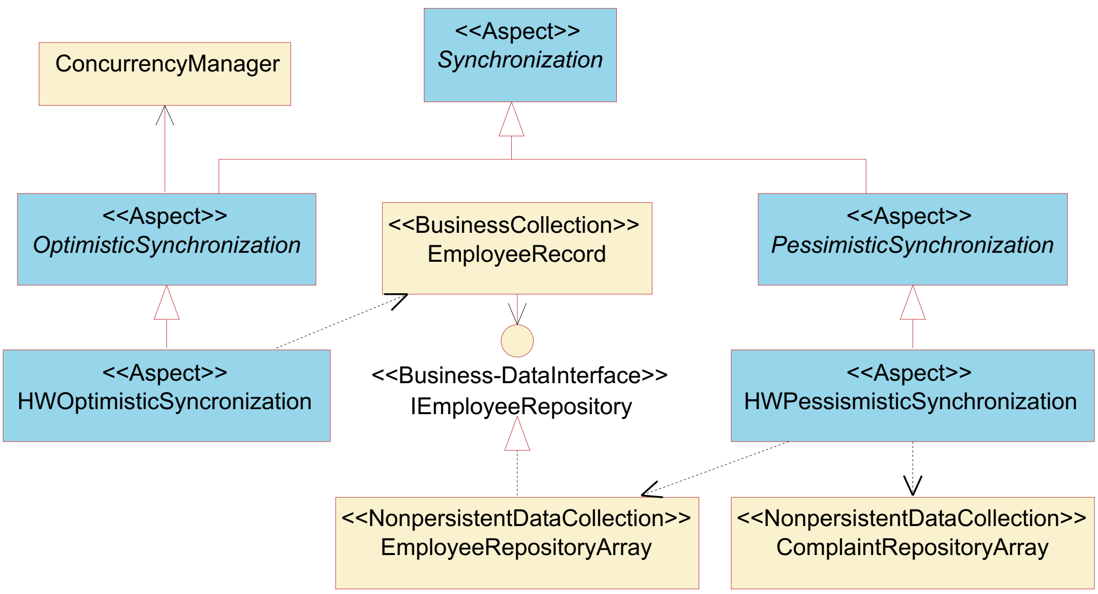

Figure 3.22: Synchronization aspects class diagram.

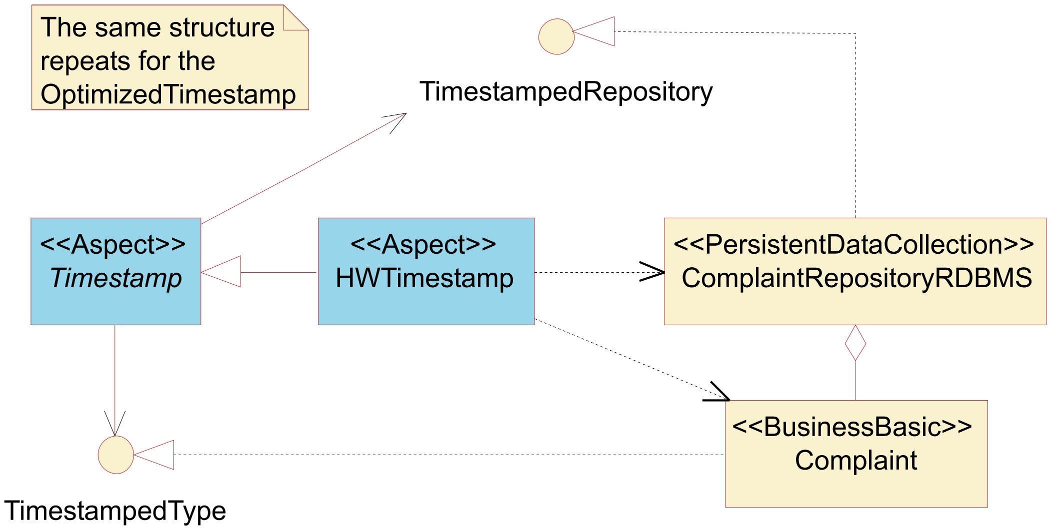

Figure 3.23: Timestamp aspects class diagram.

3.7.4 Concurrency control Framework

Figure 3.24 depicts the Concurrency Control Framework.

Despite being tailored to a specific software architecture, the synchronization aspects can be easily applied for other architectures, being necessary to identify which methods should be synchronized and which approach to use. The timestamp aspects could also be used in other architectures with some modifications, but they are harder to reuse since they depend on the architecture's data management organization.

3.7.5 Concurrency control dynamics

Figure 3.25 depicts the dynamics of synchronization aspects. The pessimistic synchronization approach is to synchronize any concurrent execution, whereas the optimistic approach synchronizes only the potentially conflicting concurrent executions.

The timestamp aspect depicted in Figure 3.26 adds members to store and manipulate the timestamp information into classes to be controlled. The aspect affects data collections of the controlled classes in order to guarantee that the timestamp information is inserted and retrieved with the objects. In addition, the aspect has to control data collections `update` methods, forbidding updates of potentially inconsistent objects. The dynamics of the optimized version is absolutely the same as this one. They have the same objective, differing only in the optimization.

3.8 Exception handling concern

As some of the advices presented so far might raise exceptions that are not handled by the advices themselves, we have 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

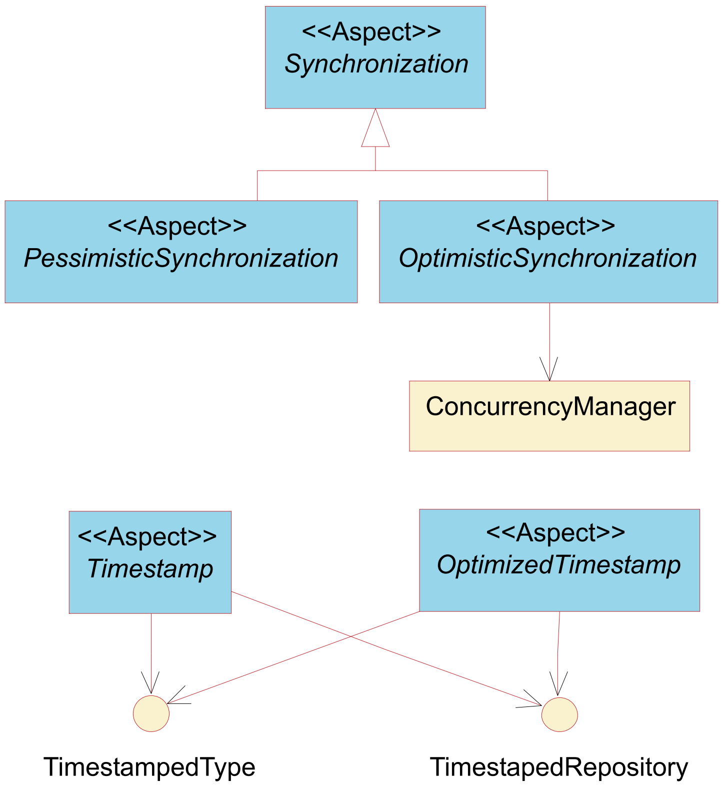

Figure 3.24: Concurrency control Framework.

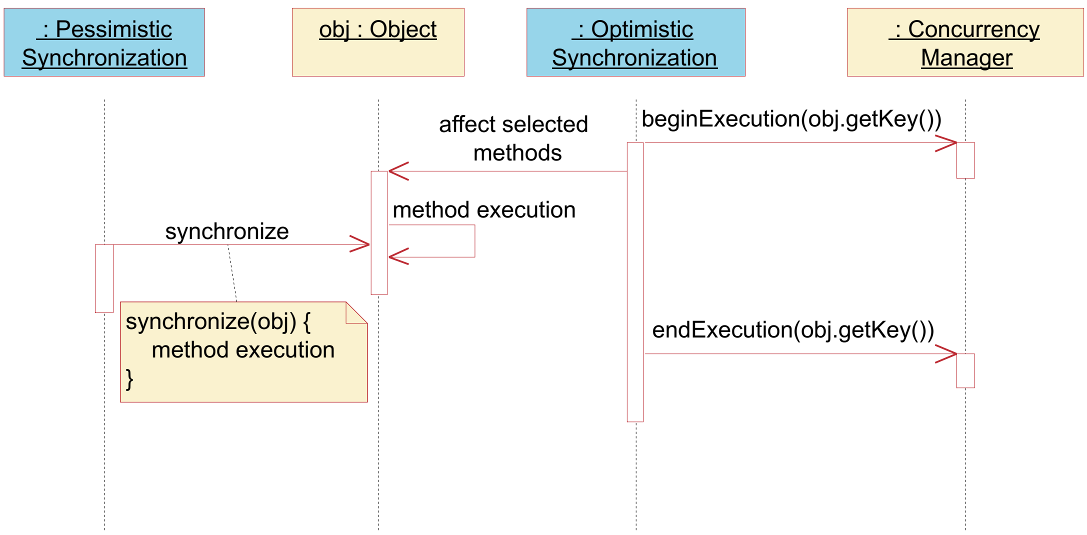

Figure 3.25: Synchronization dynamics.

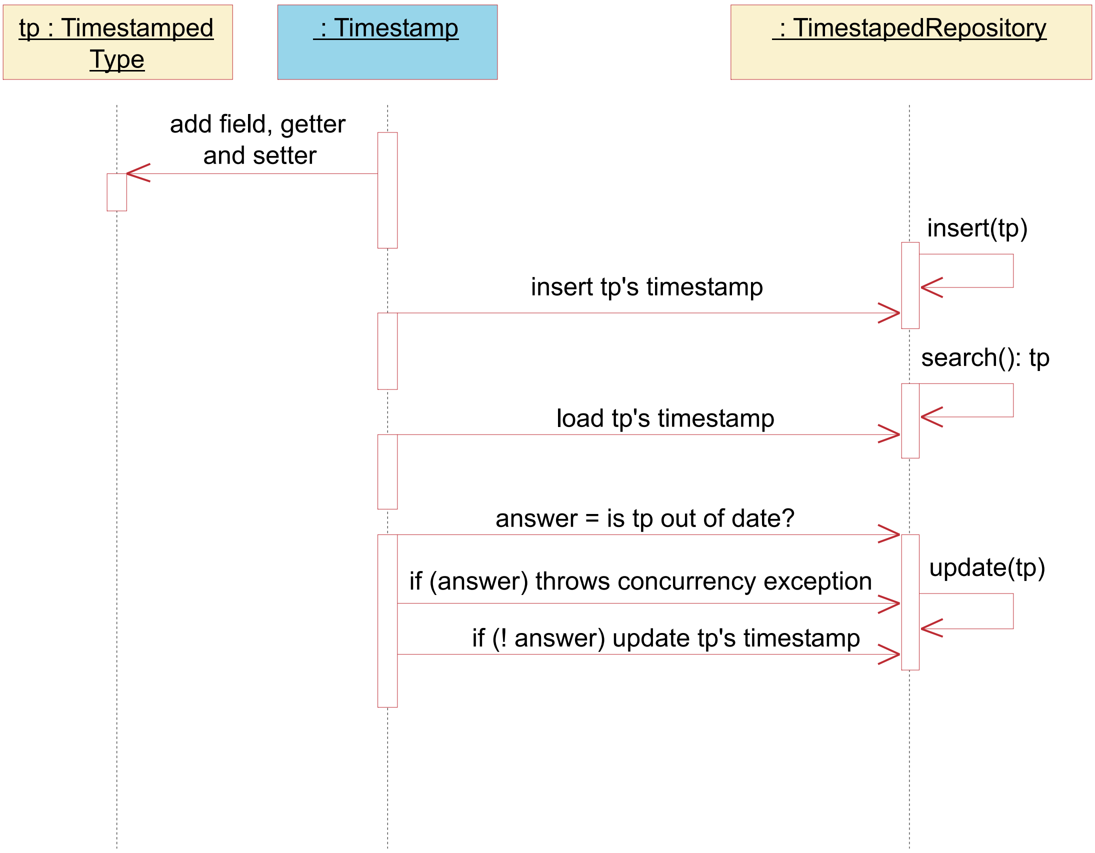

Figure 3.26: Timestamp dynamics.

to handle other types of exceptions as well. Although exception handling is a natural crosscutting concern, usually implemented with spread code, in our restructuring experience 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); }
    return o;
  }
  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 afterreturning 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 of the do∗ servlets methods). This is necessary because one PrintWriter object is created for each request received by a servlet. Therefore, when handling exceptions, we should make sure to use the right PrintWriter to notify the user.

This aspect also provides the concrete definition of the exceptionHandling method. It 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 3.27 presents a class diagram of the exception handling aspects and the Health Watcher classes and interfaces the aspects affect or use.

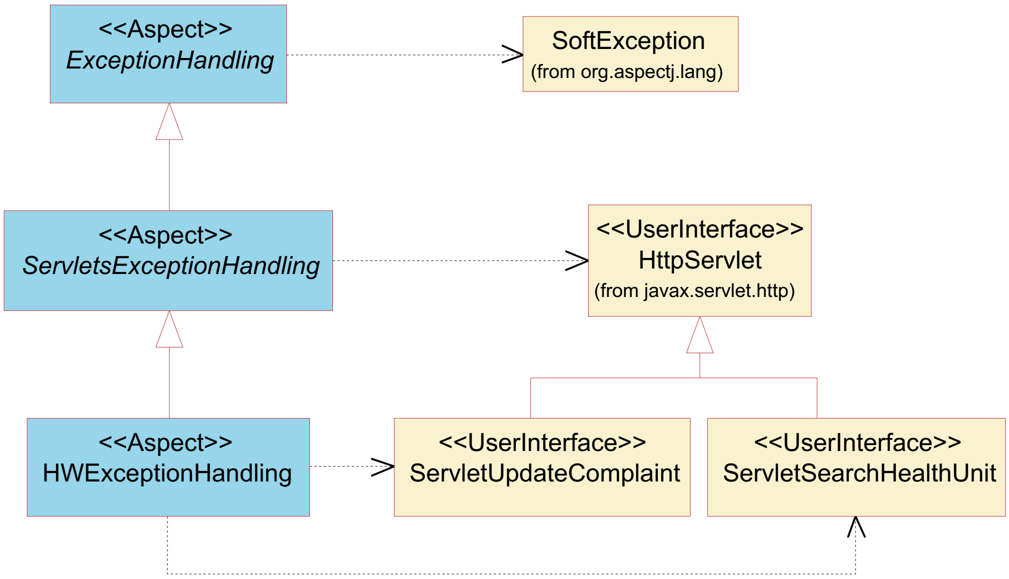

Figure 3.27: Exception handling aspects class diagram.

3.8.1 Exception handling framework

Figure 3.28 presents the abstract aspects to be reused in other aspect-oriented development. Similar to the distribution framework, the exception-handling framework is general enough to be reused in any kind of software and to handle any kind of exception, not necessarily the exceptions of the Health Watcher software.

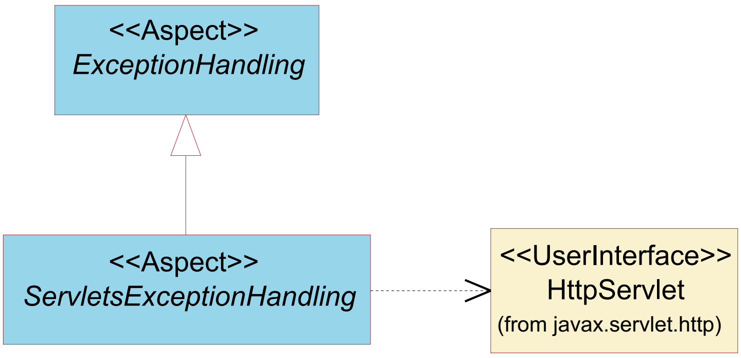

Figure 3.28: Exception-handling framework.

3.9 Interferences between aspects

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

- 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 of information is not transmitted from the client-side to the server-side.

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 3.6 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 3.29 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 [61]. 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.

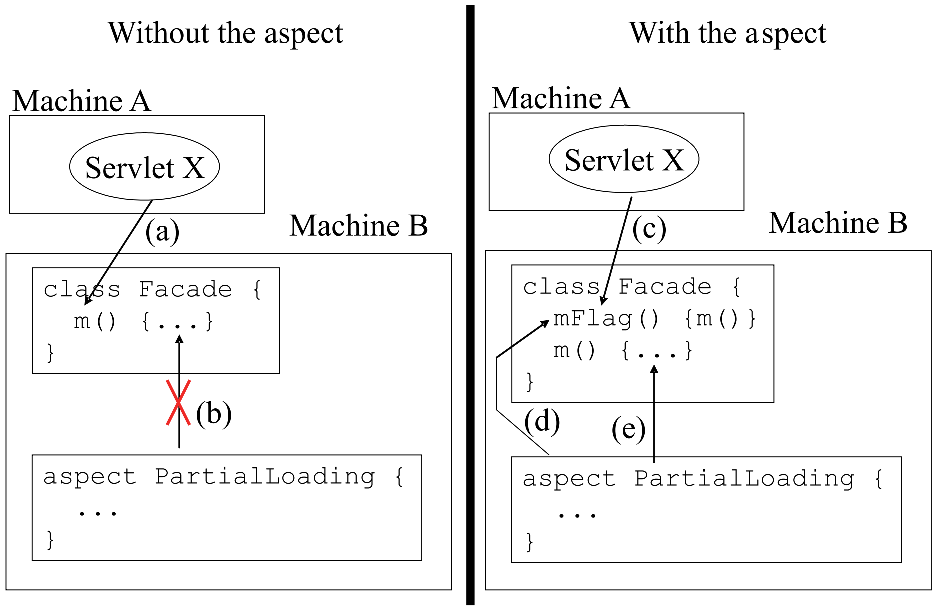

Figure 3.29: Interference problem and solution.

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 3.29.

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 as the one of the facade methods but with a prefix (`flag`). Whenever this method is called, the data collection `search` method must 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):
      cflow(healthUnitFetchDataFacadeIntroducedMethod()) &&
      target(huData) && call(HealthUnit search(int)) && args(code);
```

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` pointcut does. 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` pointcut to the added flag methods. This is our solution to identify what methods in the server-side are 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 `around` advice redirects 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);
      }
    }
  }
```

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 cannot be unaware of the other crosscutting concerns if they can interfere with each other.

Figure 3.30 shows a class diagram that presents three distribution 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.

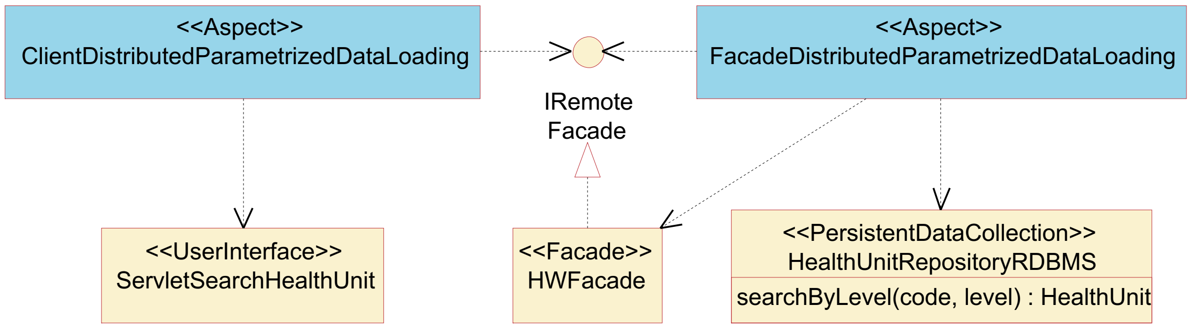

Figure 3.30: Interferences between aspects.

3.10 An alternative implementation approach

The previous sections describe guidelines to implement aspect-oriented software, considering data management, distribution, and concurrency control. These crosscutting concerns can be implemented in different ways and in a different order. They might be implemented at the same time as the functional requirements are being implemented. Another idea is to follow a progressive approach, where persistence, distribution, and concurrency control are not initially considered in the implementation activities, but are gradually introduced, preserving the system's functional requirements.

The progressive approach helps in decreasing the impact in requirement changes during the system development, since a great part of the changes might occur before the final version of the system is finished. This is possible because a completely functional prototype is implemented without persistence, distribution, and concurrency control, allowing requirements validation without interference of these non-functional requirements and without the effort to implement those. At this time, the system uses non-persistent data structures, such as arrays, vectors, and lists, and is executed in a single-used environment. Moreover, the progressive approach helps to deal with the inherent complexity of modern systems, through the support to gradual implementation and tests of the intermediate system versions.

Figure 3.31 depicts the dynamics difference between the progressive approach and the regular approach, so-called non-progressive approach. First of all, it is important to mention that despite the implementation approach used, the development is considered to be incremental. Therefore, a set of iterations should be planed to address a set of use cases of use-case scenarios.

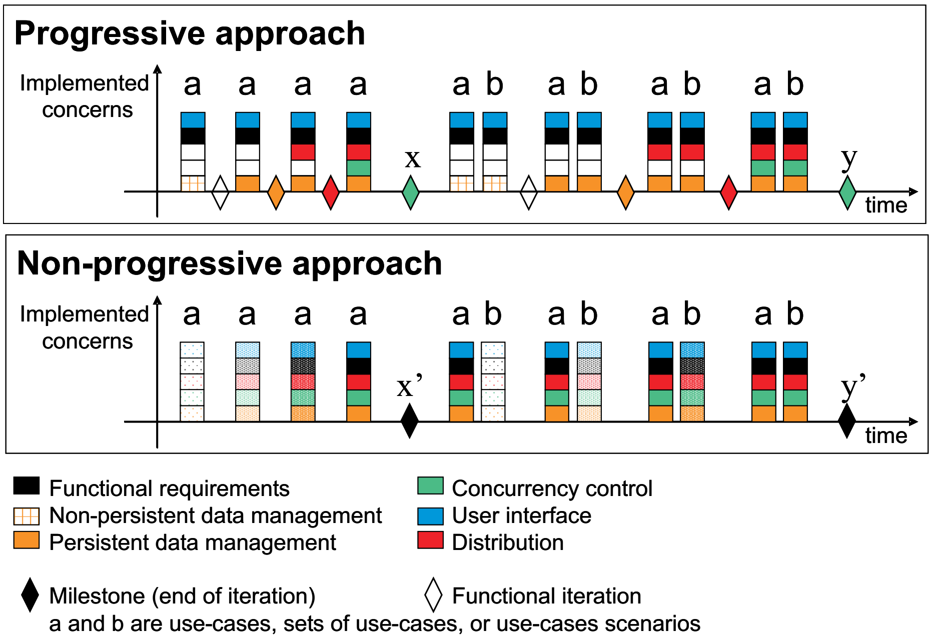

Figure 3.31: Progressive versus Non-progressive approach.

In the progressive approach, each iteration is subdivided into one functional and several non-functional iterations. First, the functional iteration is executed, where the functional requirements of the selected use cases are implemented as well as the user interface and nonpersistent data collections to support the implemented use cases. The result of this functional iteration is a functional prototype that can be validated with the customer. After validating the functional prototype, a non-functional iteration is placed to implement a single non-functional requirement and test this requirement in the end of the iteration. In our case, we are considering three non-functional requirements: persistence, distribution, and concurrency control, and therefore, we need to plan three non-functional iterations.

On the other hand, if not using the progressive approach, which means using a non-progressive approach, the functional and non-functional requirements are implemented in a same iteration. Therefore, the prototype is validated with the customer after implementing persistence, distribution, and concurrency control. If the customer requests any change at this moment, it might be necessary to change part of these non-functional requirements code. If using the progressive approach, the changes requested by the customer would not affect these non-functional requirements code, since their code was not yet implemented at the time the prototype is validated.

Chapter 4 presents how this approach affects a software development process.

3.11 Conclusion

In this chapter, we described the guidelines derived from a restructuring experience to implement distribution, data management, concurrency control, and exception handling aspects. This guidelines support the implementation method by guiding programmers on how implementing such concerns using aspect-oriented programming. This guide guarantees that all the effort made in requirements, analysis and design activities is not wasted during implementation activities.

Moreover, an aspect framework was defined to support the concerns implementation. The framework allows reusing part of the aspects behavior, also guiding the definition of the concrete aspects.

In general, the modularization achieved by the aspect-oriented version of the Health Watcher software is much better than the object-oriented one. In fact, Chapter 7 describe a related work that analyzed the difference between these two versions (Section 7.1) using some metrics they defined. In fact, their metrics show that the aspect-oriented version has several advantages over the object-oriented one.

On the other hand, the restructuring experience identified some AspectJ limitations that avoid reaching better results. A summary of such limitations are:

- The lack of support to add exceptions in methods throws clause;
- The use of soft exceptions to wrap checked exceptions into unchecked ones;
- The lack of support to remote join points;
- The property based pointcut definition, which makes aspects highly coupled with the coding standards and types names;
- The lack of support to identify the resulting join points defined by a pointcut;
- The lack of support to identify aspects interferences;

Some of these limitations demand the use of development tools to support programmers, allowing, for example, to identify if a method is affected by an aspect or what are the join points affected by an advice.

Next chapter describes the other part of the implementation method, which describes how implementation activities use the defined guidelines. In addition, the next chapter inserts these implementation activities in the context of a software development process, also describing changes on other process activities and dynamics.

Chapter 4

Integration with RUP

This chapter presents how the guidelines of the aspect-oriented implementation method defined in the previous chapter are related to a software development process, more specifically the Rational Unified Process (RUP).

As mentioned in Chapter 1, implementation methods are usually neglected by software engineers and researchers. If no commitment is made with implementation activities, the effort given to requirements and design may be wasted. Chapter 3 discusses some implementation guidelines and this chapter insert them in the context of a software development process, where we also have to consider management, requirements, analysis and design, and testing activities. The next sections present an overview of the Rational Unified Process — RUP [39], followed by changes made into the dynamics structure (phases) and into the static structure (disciplines/activities) of RUP. Those changes are necessary to adapt RUP in order to apply the implementation method also allowing the use of the progressive implementation approach.

A previous work [52] made similar changes to RUP to make it complying with an Object-Oriented Implementation method that considers the same concerns considered here. This chapter is based on that work, however, tailoring the process to an Aspect-Oriented development and to the use of aspect-oriented implementation guidelines defined in the previous chapter.

4.1 A RUP overview

The Rational Unified Process (RUP) is a well-known development process defined by "The Three Amigos" (Jacobson, Booch and Rumbaugh). RUP is the end product of three decades of development and practical use [39], and has achieved wide acceptance, including several software companies located in our area. In addition to our experience with this development process, those are sufficient reasons to choose RUP as target of our integration.

As a process, RUP describes activities and how to apply them in order to develop a software. In fact, RUP is a process framework, meaning that companies should instantiate RUP according to their needs, application area, and company organization. This chapter aims actually not in changing a specific process instantiated from RUP, but changing the process framework, allowing others to instantiate their process from this definition. RUP uses the Unified Modeling Language (UML) [10], a standard modeling language, to describe several models during the process flow.

The process is based on three pillars:

- Use-case driven — A use case is an artifact that specifies how an actor (users, other systems, or devices) interacts with the system to be developed. All the use cases of a system define the use-case model, which describes the system's requirements. This model should answer the question: "What is the system supposed to do for each actor?". The use-case model drives design, implementation and tests. Based on the use cases, developers create a series of design and implementation models that realize the use cases and review each model for conformance to the use-case model;

- Architecture-centric — The architecture structures the system, giving it a form. The execution platform, used frameworks, and non-functional requirements influence the architecture of a system. Although the architecture influences the selection of the use cases, the use-case model also drives the architecture. Every system has functions (use cases) and form (architecture), and these two forces should be balanced in order to achieve a successful development project. Use-case realization must fit in the architecture and the architecture must allow the use-case realizations. Use cases and architecture must evolve in parallel;

- Iterative and incremental — Linear development processes using a waterfall-like [74] approach are unrealistic. On the other hand, it is a good principle to divide to conquer, i.e., breaking the development project into sub-projects. Each sub-project is an iteration and each iteration increments the system. Requirements, analysis, design, implementation and test disciplines are executed in each iteration to accomplish the objectives planed for that iteration, which runs as a mini-project to realize selected use cases. In fact, there is a broader concept of phases to determine specific goals that have to be reached. The iterations of a phase should be planned in order to reach the phase's goal.

4.1.1 Lifecycle structure

The software lifecycle is broken into cycles, each working on a new generation of the product. One RUP development cycle consists of four phases with distinct goals:

- Inception — In this phase the executed activities are tailored to define the software major goals, propose a candidate architecture, and plan the whole development project, also estimating time and cost;

- Elaboration — The primary goals of Elaboration are to specify most use cases and to design and validate the software architecture. In order to allow this architectural validation, the most critical use cases should be realized. At the end of this phase, the project manager can estimate more solid amount of time and costs required to develop the software;

- Construction — During this phase the software is implemented. Minor architectural changes can be suggested as developers discover better ways to structure the software;

- Transition — At the end of Construction all the use cases are implemented, but they are not free of defects. During Transition phase, the alpha release moves to a beta release, the point at which developers and customers use the software in order to identify defects and deficiencies. These defects are corrected, and some of the suggested improvements are incorporated into the final release, while others should wait until the next releases.

Figure 4.1 depicts the emphasis of business modeling, requirements, analysis, design, implementation, test and deployment disciplines during each phase of RUP. Note that on the first phases there is a commitment with business modeling, requirement, and analysis and design activities; on the other hand, the last phases are mostly concerned with implementation, tests, and deployment activities. Despite having more emphasis on one or another activity, activities of any discipline can be potentially executed at any phase. For example, there are test activities that can be executed during Inception, such as **Plan Tests**.

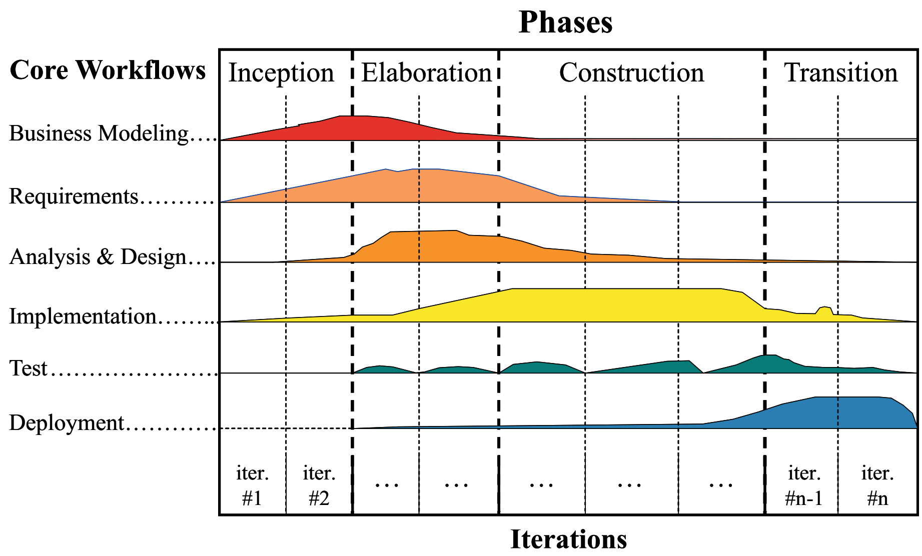

Figure 4.1: RUP disciplines taking place over phases [39].

The four phases are major milestones, while each phase is subdivided into iterations, where activities of each discipline can be executed, with different emphasis depending on the current phase. Each iteration flows like a waterfall process. Each software cycle executes these four phases resulting in a software release. During the software lifecycle, this cycle is repeated, yielding a release in each cycle.

4.2 Impact on RUP's dynamic structure

This section discusses changes that have to be made into RUP's phases and iterations, in order to allow the application of the aspect-oriented implementation method defined in Chapter 3.

Inception

"The Inception Phase launches the project" [39]. In this phase, the activities are driven by the definition of the project goals. The business case is defined in order to justify the project. The project scope is delimited in order to discern what the architecture has to cover, to define limits within the critical risks should be analyzed, and to provide boundaries of cost, schedule, and return on-investment, what is crucial to plan the other phases and their iterations, another goal of inception.

When planning the project iterations, the project manager has to consider the alternative implementation approach (progressive approach) allowed by the implementation method. In this way, the iterations that are progressively implemented have to be split into functional and non-functional iterations. During these iterations, new implementation activities should be executed. These activities are presented in Section 4.3, which gives more details on how to compose the progressive implementation approach with RUP.

Another change regards the need to propose a candidate architecture by the end of the phase. Since the implementation method is tailored to a specific software architecture, this architecture should be the candidate architecture.

During critical risk analysis, new risks due to the aspect-oriented implementation method should be considered:

1. *The programmers do not have experience with aspect-oriented programming* — A mitigation strategy to teams without enough experience is to schedule training in the chosen aspect-oriented programming language. Another strategy might be hiring expert programmers in the paradigm and language. In fact, the use of aspects can be delayed in the development cycle, also delaying this risk analysis.

2. *The specific software architecture might not match the customers's requirements* — In this context, a mitigation strategy might be selecting an important use case to completely realize during inception, however, without compromising the phase's goal. On the other hand, as previously mentioned, the software architecture used by the method can fit to several kinds of systems (see Chapter 3). Furthermore, as one of next phase's goals is to validate the software architecture, this control strategy can be delayed until the next phase.

The other activities should follow the regular RUP baseline, since neither they affect nor they are affected by the implementation method.

Elaboration

"The Elaboration Phase makes the architectural baseline" [39]. Before starting this phase, we had already identified a candidate architecture, the most critical risks, and established a business case stating that it is worth continuing the project. The goals of this phase are to capture around 80% of the requirements, establish a sound and robust architecture, monitor the critical risks also identifying some significant risks, and complete the project plan. As previously mentioned, the risk analysis regarding the use of aspect-oriented programming might have been delayed to this phase. All the activities executed in this phase are driven to establish the software architecture.

The major modification in this phase's dynamics is regarding the use of the progressive approach. In this way, we need to define how the progressive approach should be composed with Use-Case Driven Development (UCDD) [35], the development technique used by RUP.

In UCDD, developers create design and implementation models that realize the use cases. Moreover, other models should comply with the use-case model, and tests should ensure that the use cases are correctly implemented. Although UCDD does not incorporate the idea of progressive implementation, our progressive approach can be combined with UCDD.

In order to combine UCDD with our progressive approach, providing a use-case driven progressive development, we define how and when non-functional requirements are to be considered and implemented. When considering a progressive implementation, we have to change analysis and design models to comply with the implementation method.

To implement a use case, programmers should implement parts of the system that are necessary to realize that use case. However, when planning a progressive approach, some non-functional requirements implementation should be schedule after implementing and validating the functional part of the use cases and the user interface code. Therefore, use cases to be implemented in an iteration should be partially implemented in a functional iteration. In functional iterations only functional requirements, user interface, and non-persistent data management are implemented. At this moment, we have a functional prototype that should be validated and, if necessary, changes should be made. After validating the implemented functional code, the prototype evolves to a persistent and distributed software, with concurrency control, in three non-functional iterations, one for each non-functional requirement. These non-functional iterations also allow progressive testing, by separately testing persistence, distribution, and concurrency control.

In fact, the project manager can switch between using or not the progressive approach in different iterations. The results presented in Chapter 5 should be used in order to support this decision about when using the progressive approach. In general, requirement changes during software implementation result in bad productivity impact if not using the progressive approach.

Construction

"Construction leads to initial operational capability" [39]. Construction is the phase in which most of the use cases are implemented. In fact, the modification in construction's dynamics is related to the use of a progressive implementation approach. In this case, the modification is the same made to the Elaboration phase, which is planning functional and nonfunctional iterations if using the progressive approach.

Transition

"Transition completes product release" [39]. In this phase, the alpha release moves to a beta release. Once more, the modification in this phase's dynamics is the same made to Elaboration and Construction, which is planning functional and nonfunctional iterations if using the progressive approach.

4.3 Impact on RUP's static structure

This section discusses changes that have to be made into RUP's activities, in order to allow applying the aspect-oriented implementation method. We only mention new or modified activities.

4.3.1 Requirements

Figure 4.2 depicts the activities of the requirements discipline. Some of them have to be extended in order to support the implementation method.

The activities **Develop Vision** and **Elicit Stakeholder Needs** should consider the support provided by the aspect-oriented implementation method to some features and needs. For example, the software architecture to be used is concerned with providing extensibility. In addition, the implementation method aspect framework supports some crosscutting concerns, such as distribution, data management, concurrency control, and

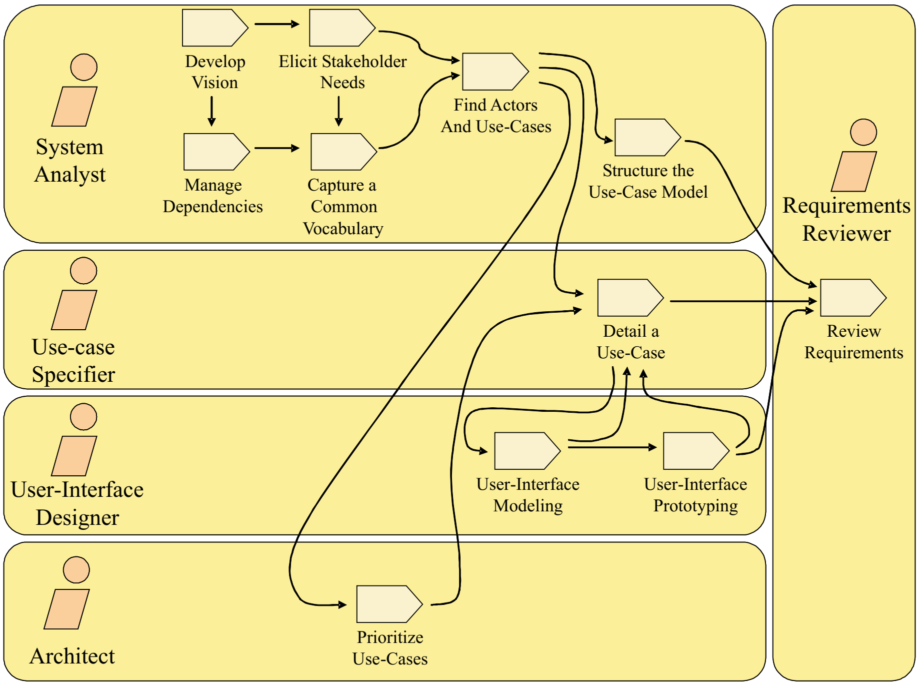

Figure 4.2: Requirement activities [39].

exception handling, besides providing tool support (see Chapter 6) to automate some aspects generation.

When executing the Activity **Detail a Use Case** it is necessary to consider the aspect-oriented development. During use-case specification, other crosscutting concerns might be identified, deriving candidate aspects. Usually, it is easier to identify non-functional requirements as crosscutting concerns; however, functional requirements that crosscut other concerns are also identifiable. This can help software analysis and design by suggesting requirements already defined as aspects. Additional artifacts and techniques can be used in order to early specify how the aspects (crosscutting concerns) affect or how they should be composed to other requirements [92]. In fact, there are related works [98, 64] concerned with the identification of aspects in this early stage towards crosscutting requirements. On the other hand, an alternative is to delay considering aspects in the first iterations.

Finally, the **User-Interface Prototyping** activity must consider the use of the progressive approach. Chapter 5 provides some number showing that by using the progressive approach a functional prototype can be rapidly delivered, at least 45% faster than not using the progressive approach, by abstracting some non-functional requirements, such as persistence, distribution and concurrency control. By using the progressive approach, the prototype can be quickly implemented without compromising quality principles, such as modularity and extensibility. This prototype technique contrasts with similar techniques, where prototypes are implemented faster, not considering modularity and extensibility. In the latter case, prototypes are eventually dropped after evaluation, whereas in our approach, prototypes are actually implemented to be the resulting software. The progressive implementation approach increases productivity by delaying the implementation of some non-functional requirements.

4.3.2 Analysis and Design

RUP does not make a clear distinction between analysis and design. Therefore, it is common to consider analysis and design as a single discipline, instead of considering two different disciplines. Figure 4.3 presents the activities of the analysis and design discipline.

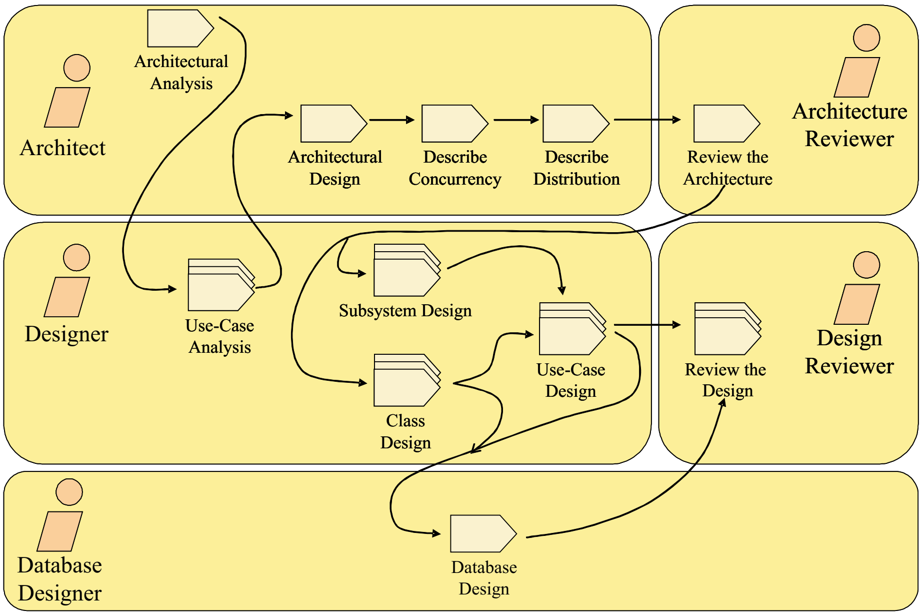

Figure 4.3: Analysis and design activities [39].

Architectural Analysis should be modified to consider the specific software architecture we use, see Chapter 3, in order to define the architectural patterns, mechanisms, and conventions for the system.

The **Use-Case Analysis** activity is concerned with identifying classes from the use-case specification. This activity uses stereotypes to identify three types of classes: boundary, control, and entity class. The boundary classes are interfaces to external systems or users (actors), control classes model the control flow of an operation, and entity classes are used to model information that is manipulated by the system and should be persistent. According to the specific architecture, the entity classes are analogous to basic classes (see Chapter 3).

Design mechanisms to support requirements, such as persistence, distribution, and concurrency control, should be identified during **Architectural Design**. Since the implementation method already supports those specific requirements, this activity should consider the support offered by the method. For example, the current version of the method offers JDBC to implement persistence, RMI to implement distribution, and uses different approaches to control concurrency (see Chapter 3). This activity should also move analysis classes into design classes. As previously mentioned, entity classes should be mapped into basic classes. The control classes should be composed into the facade class, or into controller classes, which should be composed into the facade class. Boundary classes are mapped to user interface classes, or interfaces to subsystems. At this moment, some aspects of our aspect framework should be identified to be used in the implementation activities. Those aspects are the ones related to data management and distribution. Those abstract aspects should be added to the class diagram in order to document their identification.

Since the implementation method already supports concurrency control and distribution, the activities **Describe Concurrency**, and **Describe Distribution** should also consider this support. The knowledge acquired while describing concurrency should be used when applying the concurrency control. The concurrency control method [84, 81, 80] uses this information to decide which parts of the software should be controlled and what is the most appropriate mechanism to use. When describing distribution we have to identify how the system functionality should be distributed. The implementation method already supports distributing the facade object. If another part of the software should be distributed, probably the method's framework and architecture can support that; however, some minor modifications might be necessary. This would happen if variations of the specific software architecture would be used, for example separating the software in two subsystems where each of them has a facade class. Another example might be any system that uses different software architectures and needs to distribute several classes, besides the facade class. Similar to the previous activity, this activity should identify aspects of the aspect framework in the software class diagram, designing the concurrency control policy and the distribution protocol to use, for example.

The last affected activity of the Analysis and Design Discipline is **Use-Case Design**. This activity defines use-case realizations in terms of interactions in order to better deal with and refine the requirements. Sequence diagrams can be used in order to describe the participant objects and the interactions among them to execute the designed use cases. These diagrams should specify the collaboration among the classes of the specific software architecture used by the implementation method. Using aspect-oriented programming, it is not necessary to present explicitly how aspects affect the architecture objects, since there is a framework to do it. Chapter 3 presents the framework, how to use it in order to add data management, distribution, and concurrency control, also presenting the impact of those aspects in static and dynamic structures of the software. The static impact is presented by several class diagrams and the dynamic impact by several sequence diagrams. In addition, Chapter 6 presents the tool support for the implementation method, also guiding the application of the framework. On the other hand, if additional concerns have to be implemented as aspects, it is necessary to depict how the aspects affect the software's static and dynamic structure, by defining class and sequence diagrams, in order to guide their definition.

4.3.3 Implementation

Figure 4.4 presents the implementation discipline. The **Structure the implementation model** activity implements the structure in which the implementation should reside. The basic classes can be automatically generated using a modeling tool, such as Rational Rose [76]. The method's tool support (see Chapter 6) can generate business collections, business-data interfaces, and the facade class with general methods already implemented[1]. If not using the progressive implementation approach, tool sup-

[1]Modeling tools, like Rose, can also generate those classes, however, without providing basic implementation for their methods.

port should be used in order to generate basic aspects and auxiliary classes to implement persistence, distribution, or concurrency. On the other hand, if using a progressive approach, only nonpersistent data collections and data management aspects should be generated in this activity. These non-persistence aspects and classes should be used by the functional prototype.

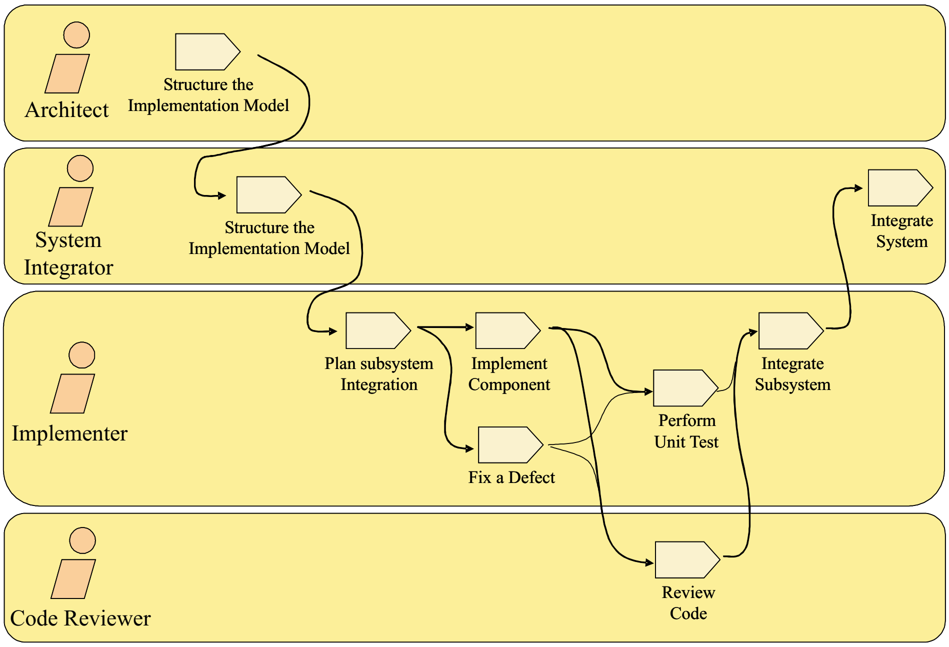

Figure 4.4: Implementation activities [39].

The **Implement component** activity can be modified in two forms. First, if the progressive approach will not be used, the activity has only to be modified in order to consider the framework that supports the implementation of persistence, distribution, and concurrency control concerns. Some aspects and auxiliary classes that use the framework can be generated using the method's tool support.

On the other hand, by using the progressive approach, besides modifying the **Implement Component** activity, new activities must be performed. Figure 4.5 presents the activities to be added to the implementation discipline, including a new version of the **Implement component** activity, in case the progressive implementation approach is used. The activities of this partial discipline replace the **Implement Component** activity on the original discipline. The incoming and outgoing flows of **Implement Component** are not changed and should be matched by "... in the beginning and in the end of the partial discipline.

The new version of the **Implement Component** activity and the new activities to be used in the progressive implementation approach are described using the RUP activity description style.

Activity: Implement Component (progressive approach version)

- **Purpose**: implementing the selected use cases or scenarios' functionality abstracting persistence, distribution, and concurrency control concerns.

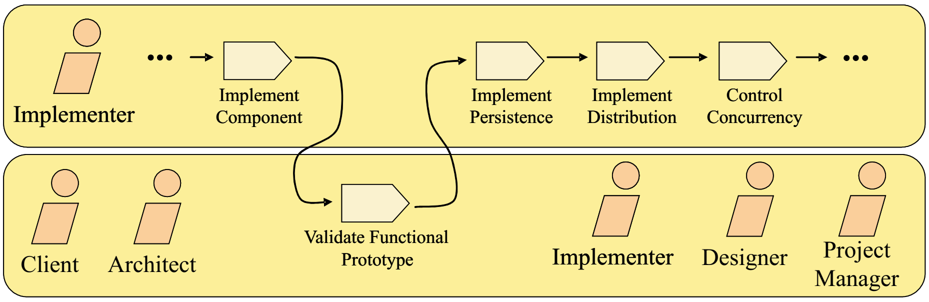

Figure 4.5: New implementation activities for progressive implementation.

- **Steps**:
  1. Turn off persistence, distribution, and concurrency control concerns, if already implemented.
  2. Turn on nonpersistent data management, if already implemented. Otherwise, generate the nonpersistent aspects and auxiliary classes, preferably using the method's tool support.
  3. Execute the steps of the original activity definition.
- **Input Artifacts**: the same of the original activity definition plus the aspects for nonpersistent data management.
- **Resulting Artifacts**: the functional prototype.
- **Worker**: Implementer.

We turn the aspects off by not composing the aspects responsible for persistence, distribution, and concurrency control concerns when weaving the system. Similarly, to turn on nonpersistent data management, we just weave them to the functional code. After implementing the functional prototype, it is necessary to execute the **Validate Functional Prototype** activity.

Activity: Validate Functional Prototype

- **Purpose**: validating the functional prototype in order to identify possible requirement changes.
- **Steps** (per worker):
  1. Implementer — Present the prototype.
  2. All — Suggest possible requirement changes.
  3. Project Manager — Evaluate the requirement changes in order to decide which should be performed.
  4. Project Manager — Review the project schedule, if necessary.
  5. Architect and Designer — Re-execute any analysis and design activities necessary to perform the change.

6. Implementer — Execute the steps of the original **Implement Component** activity in order to perform selected requirement changes.
7. All — Review the prototype after performing the requirement changes in order to validate the prototype.

- **Input Artifacts**: The prototype implemented in the **Implement Component** activity.
- **Resulting Artifacts**: A list of requirements to change or a document stating the validation of the prototype without any change.
- **Workers**: Implementer, Client, Architect, Designer, and Project Manager.

The architect or designer might also suggest some changes in order to achieve a better result. For example, an implementer may complain about problems in implementing a part of the prototype. In addition, some model may not be implemented as demanded, because of an implementer or designer mistake.

Changes resulting from implementation or design mistakes must be implemented by re-executing the **Implement Component** activity, and in case of a modeling mistake, the models should also be updated. Since those kinds of changes might suggest technical problems, the project manager should be aware of that in order to perform the necessary steps, like executing any contingency plan. The manager can also re-plan the schedule to accommodate this unexpected delay. On the other hand, changes requested by customers should be analyzed by the project manager in order to evaluate whether they can be accommodated into the current system scope. In case the changes are out of the defined scope, new negotiation on cost and schedule is necessary. These changes usually demands changes in analysis and design models.

Activity: Implement Persistence

- **Purpose**: implementing persistence in the functional prototype.
- **Steps**:
  1. Make sure distribution and concurrency control concerns are turned off, if already implemented.
  2. Turn off nonpersistent data management.
  3. Turn on persistent data management, if already implemented. Otherwise, generate the persistence aspects and auxiliary classes using the data management framework and the method's tool support (see Chapter 3).
  4. Complete the implementation of persistent data collections methods used by the implemented use cases or scenarios (see Chapter 3).
  5. Test persistence of the implemented use cases or scenarios.
- **Input Artifacts**: the validated functional prototype.
- **Resulting Artifacts**: the functional and persistent prototype.

- **Worker**: Implementer.

Note that the persistence aspects should be generated in the first iteration. When these persistence aspects are already generated, the main task is to implement data collections' methods. Since the functional prototype, used as input to this activity, is already tested, the tests made in the end of this activity can focus on testing persistence properties, yielding a functional and persistent prototype. More details on how to implement persistence are given in Chapter 3.

Activity: Implement Distribution

- **Purpose**: implementing distribution on the functional and persistent prototype.
- **Steps**:
  1. Make sure concurrency control aspects are turned off, if already implemented, and persistent data management is turned on.
  2. Turn on distribution aspects, if already implemented. Otherwise, generate the distribution aspects and auxiliary classes using the distribution framework and the method's tool support (see Chapter 3).
  3. Implement distribution in the selected use cases or scenarios (see Chapter 3).
  4. Test distribution in the implemented use cases or scenarios.
     (a) Turn off persistent data management (optional).
     (b) Turn on nonpersistent data management (optional).
     (c) Test distribution in the implemented use cases or scenarios (optional).
     (d) Turn off nonpersistent data management (optional).
     (e) Turn on persistent data management (optional).
     (f) Test distribution in the implemented use cases or scenarios.
- **Input Artifacts**: the functional and persistent prototype.
- **Resulting Artifacts**: the functional, persistent, and distributed prototype.
- **Worker**: Implementer.

If distribution was already implemented in previous iterations, there are only simple tasks to be performed, such as identifying new classes whose objects are remotely sent and serializing them. Since functional and persistence concerns are already tested, the tests made in this activity can focus on distribution properties. In fact, before testing distribution in the persistent software, some optional steps can be executed to test distribution independently from persistence, which might be useful for identifying possible interferences between them. However, if one optional step is executed, all of them must also be executed to ensure testing consistency. Chapter 3 presents details on how to implement distribution.

Activity: Control Concurrency

- **Purpose**: controlling concurrency in the functional, persistent and distributed prototype.
- **Steps**:
  1. Make sure persistence and distribution aspects are turned on.
  2. Turn on concurrency control aspects, if already implemented. Otherwise, generate the concurrency control aspects and auxiliary classes using the concurrency control framework and the method's tool support (see Chapter 3).
  3. Implement the necessary concurrency control in the implemented use cases or scenarios using the most appropriate mechanism (see Chapter 3).
     (a) Turn off persistent data management (optional).
     (b) Turn on nonpersistent data management (optional).
     (c) Turn off distribution data management (optional).
     (d) Test concurrency control in the implemented use cases or scenarios (optional).
     (e) Turn on distribution data management (optional).
     (f) Test concurrency control in the implemented use cases or scenarios (optional).
     (g) Turn off nonpersistent data management (optional).
     (h) Turn on persistent data management (optional).
     (i) Test concurrency control in the implemented use cases or scenarios.
- **Input Artifacts**: the functional, persistent and distributed prototype.
- **Resulting Artifacts**: the functional, persistent and distributed prototype with concurrency control.
- **Worker**: Implementer.

This activity should apply the concurrency control method [84, 81, 80] to analyze the software, identifying parts that should be controlled and the most appropriate control mechanism to use. Again, tests should be focused on the applied concurrency control. Similarly to the **Implement Distribution** activity, we can also test concurrency isolated from persistence and distribution in order to identify possible interferences between these aspects. Once more, if one optional step is executed all of them must be executed too to ensure testing consistency. Chapter 3 presents details on how to implement concurrency control.

The **Implement Persistence** activity might be scheduled after the **Implement Distribution** and **Control Concurrency** activities, since there is no dependency between them. However, if this happens, some optional steps should be added to the **Implement Persistence** activity in order to test persistence with and without distribution and concurrency control, similarly to the **Implement Distribution** and **Control Concurrency** activities. Since the **Control Concurrency** activity has to be performed only if the software is distributed, it should be always executed after the **Implement Distribution** activity.

4.3.4 Test

The test discipline presented by Figure 4.6 defines activities to be performed in order to test the software.

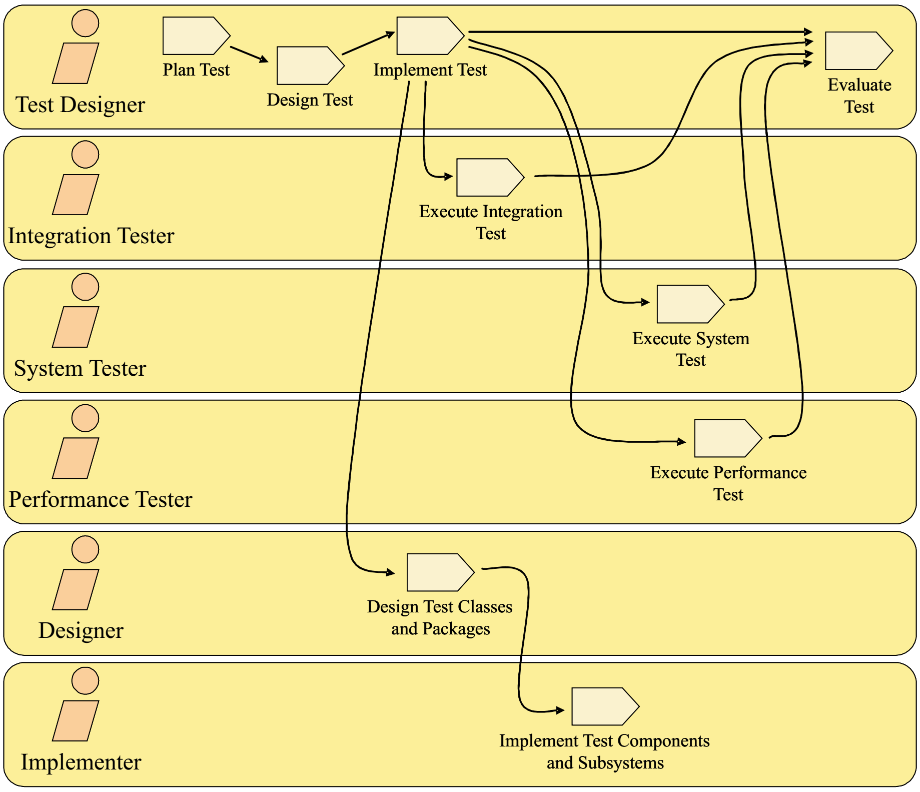

Figure 4.6: Test activities [39].

When executing the **Plan Test** activity it is necessary to describe the testing strategy. In fact, the testing strategy should consider the dynamics of the implementation activities in order to support progressive testing. In this way, as mentioned in the implementation discipline, when testing the functional prototype, the concerns related to persistence, distribution, and concurrency control must be turned off. In the same sense, concerns related to distribution and concurrency control must be turned off when testing persistence, and concerns related to concurrency control are turned off when testing distribution.

In fact, alternative strategies may be adopted, as in the optional steps of the added implementation activities, which allow combining different test approaches. For instance, aspects of a concern can be initially tested independently of other concerns. Next, several combinations can be applied to test this concern with others allowing identifying if their aspects affect each other. Actually, the aspect-oriented paradigm plays an important role in this progressive testing, since it makes easy and simple turning on and off the aspects.

4.4 Conclusion

This chapter makes changes into the dynamics structure (phases) and into the static structure (disciplines/activities) of the RUP software development process. Those changes actually do not aim in changing a specific process instantiated from RUP, but changing the process framework, allowing others to instantiate their process from this definition.

The modifications were made to allow using the specific software architecture, the implementation guidelines, and the aspect framework defined in the previous chapter. The composition of these with a development process result the implementation method. In fact, the implementation method's core is located in the implementation activities. However, it does not make sense to neglect other activities of the development process lifecycle. For example, several analysis and design activities are directly influenced by the aspects framework, where aspects that implement some concerns should be identified before implementation activities.

Therefore, besides the guidelines and the changed and added implementation activities, the implementation method is also composed of the changes made in other activities of the development process, since this is necessary to make them complying with the implementation method and its implementation guidelines.

Figure 4.7 depicts the implementation method in the context of a development process framework. It is important to mention that additional work should be done to address open issues related to aspects, such as requirements, modeling, and testing.

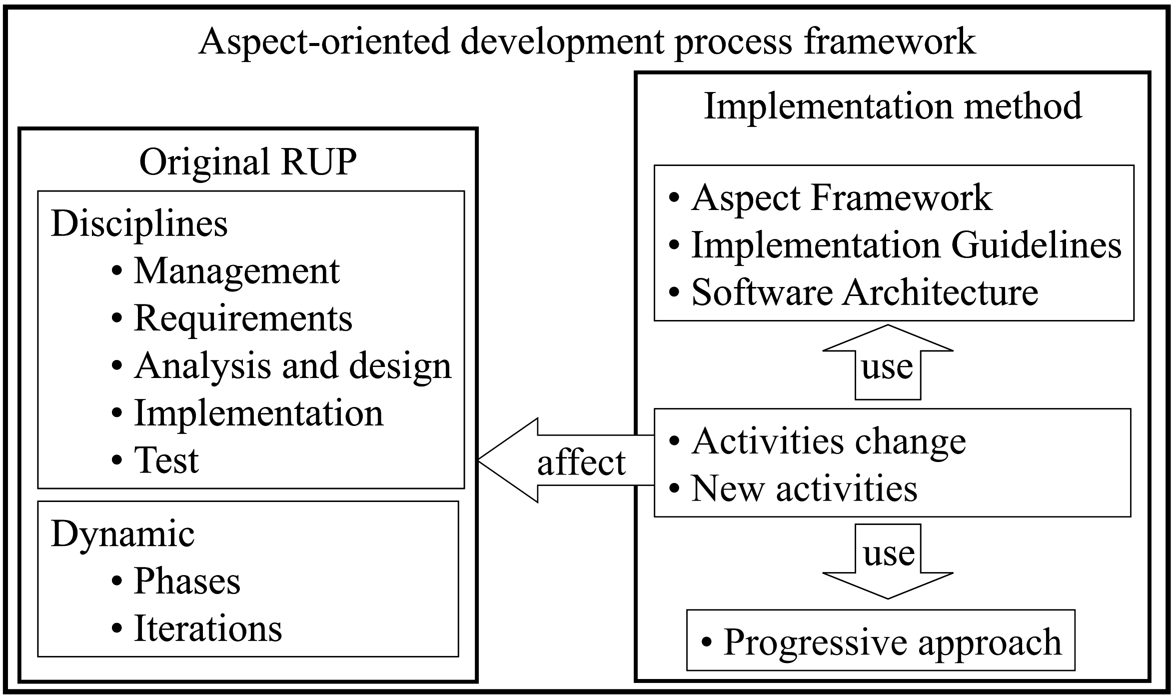

Figure 4.7: An aspect-oriented development process framework.

Chapter 5

Analysis of the progressive approach

This chapter reports an experimental study with the implementation method. This study analyzes an alternative approach to execute the implementation activities of the method, by suggesting when and how effective is using it.

Experimentation is a very important discipline for several research areas, which includes software engineering. Empirical studies are necessary to evaluate how effective or how promising processes, methodologies and techniques are in contrast to other approaches. Experimentation also allows determining the value of the claims about the subject of the study, supporting decision makers to choose which method, process or technique to use in a software development project.

Unfortunately, several software engineers do not give much attention to experimentation [104], which might hinder how reliable or promising is the methodology or the technique. In order to better understand the progressive approach impact on the implementation activities, we performed an experimental study using the implementation method and comparing the progressive approach versus the regular approach. In fact, this study does not cover the whole implementation method, since the concurrency control concern is not considered. Therefore, the study focuses on data management and distribution concerns.

There are several principles of experimentation and empirical studies that must be followed [5, 104, 69, 40], otherwise any conclusions derived of the so-called experiment might not be useful. Therefore, another contribution of this chapter is to provide a framework that allows performing other studies under the same conditions to confirm some results we derived, and others we could not achieve.

This chapter describe the study plan specifying the study goal, the hypotheses investigated, the treatment applied and analyzed by the study, the control object, which is the object to be compared with the application of the treatment, deriving the analyzes, the experimental object, the object to which the treatment is applied, and the subjects, which are the participants of the study. Moreover, the independent variables, which are variables that do not change their values during the study, and the dependent variables, which are variables to be affected by the study, and therefore, variables to be measured in order to evaluate the study are also defined. Finally, the design, preparation, treats of validity, execution, and analysis of the study are presented, following a format based on [100].

The next sections describe the designed study performed from January to March 2004 in order to characterize the impact of using a progressive approach (see Chapter 3) when executing the implementation activities of the implementation method (see Chapter 4). It is important to make clear that this experimental study is concerned to characterize the use of the progressive approach versus not using it, and does not try to make any discussion between using aspect-oriented programming or object-oriented programming.

5.1 Goal definition

The following sections describe the goal of this experimental study.

5.1.1 Global goal

Characterize the progressive implementation approach in the context of aspect-oriented software implemented using AspectJ.

5.1.2 Measurement goal

Considering the progressive implementation approach, we plan to characterize its difference from a regular approach, which we call the non-progressive approach, with respect to:

- Implementation time: time to implement selected use cases;
- Requirement changes time (during development): time to perform changes in selected use cases, after the customer or the software architect request the change;
- Test execution time: time to run test-cases of selected use cases (acceptance tests);
- Pre-validation prototype time: time to yield a first executable prototype of selected use cases, before validation;
- Post-validation prototype time: time to yield an executable prototype and validate it with the customer. This implies that requirements are validated and some of them might have changed. A mentor played the customer role.

5.1.3 Study goal

Analyze the progressive implementation approach for aspect-oriented software and AspectJ.

With the purpose of characterize the impact of adopting a progressive implementation approach.

With respect to the time to implement, change, testing, and yield pre and post-validation prototypes.

From the point of view of the software developer.

In the context of students of a software engineering graduate course, with industry expertise, implementing selected use cases of a real application.

5.1.4 Questions

Is the time to implement, change, testing, and yield pre and post-validation prototypes in a progressive way different from not using a progressive approach in the context of aspect-oriented software implementation?

5.1.5 Metrics

Data on how long it takes to implement, change, testing, and yield pre and post-validation prototypes of the selected use cases. The data unit is time in minutes.

5.2 Planning

This section describes the study plan showing how the study is designed. This allows running other studies using the same plan, which can confirm some of our results and derive new ones we could not derive. Actually, future studies can change some variables in order to measure their impact.

5.2.1 Hypotheses definition

Before presenting the hypotheses, it is necessary to introduce some symbols used through the rest of this chapter to denote the metrics to be collected and analyzed.

IE	—	Implementation time;
TE	—	Testing time;
CE	—	Change time;
PE	—	Time to yield a pre-validation functional prototype;
VE	—	Time to yield a post-validation functional prototype.

Each of these metrics has two variations, progressive and non-progressive. For example, there is implementation time using the progressive approach (IE_P) and the same metric using the non-progressive approach (IE_{NP}). The following hypotheses definition use these symbols.

The main hypothesis is the null hypothesis that states there is no difference between using or not the progressive approach. Therefore, the study tries to reject this hypothesis. There are five null hypotheses, one for each metric the study analyzes. The following is a composition of these five null hypotheses.

Null hypotheses ($H0_{1..5}$): The time to implement (1), testing (2), change (3), and yield pre (4) and post-validation (5) prototypes using a progressive approach for aspect-oriented development is not different from using a non-progressive approach.

$H0_1 : IE_P \cong IE_{NP}$
$H0_2 : TE_P \cong TE_{NP}$
$H0_3 : CE_P \cong CE_{NP}$
$H0_4 : PE_P \cong PE_{NP}$
$H0_5 : VE_P \cong VE_{NP}$

Additionally, alternative hypotheses are defined to be accepted when the corresponding null hypothesis is rejected. In fact, there are two different alternative hypotheses set.

Alternative hypothesis ($H1_{1..5}$): The time to implement (1), testing (2), change (3), and yield pre (4) and post-validation (5) prototypes using a progressive approach for aspect-oriented development is different from using a non-progressive approach.

$H1_1 : IE_P \neq IE_{NP}$
$H1_2 : TE_P \neq TE_{NP}$
$H1_3 : CE_P \neq CE_{NP}$
$H1_4 : PE_P \neq PE_{NP}$
$H1_5 : VE_P \neq VE_{NP}$

Alternative hypothesis ($H2_{1..5}$): The time to implement (1), testing (2), change (3), and yield pre (4) and post-validation (5) prototypes using a progressive approach for aspect-oriented development is smaller than using a non-progressive approach.

$H2_1 : IE_P < IE_{NP}$
$H2_2 : TE_P < TE_{NP}$
$H2_3 : CE_P < CE_{NP}$
$H2_4 : PE_P < PE_{NP}$
$H2_5 : VE_P < VE_{NP}$

5.2.2 Treatment

The treatment applied in this study is the progressive implementation approach, where persistence and distribution are not initially considered in the implementation activities, but are gradually introduced, preserving functional requirements. This is a one factor experimental study, since we have a single treatment. The absence of the treatment (the progressive approach) is called non-progressive approach, which is discussed in the following section.

5.2.3 Control object

In order to identify the impacts of using a progressive approach, the control object is the implementation of the experimental object without the progressive approach, which we call non-progressive approach. Therefore, in the non-progressive approach, persistence and distribution are implemented at the same time, in a single iteration, together with the functional requirements. In this way, the subjects are divided in two groups: the subjects of one group implement use cases using the progressive approach, whereas the subjects of the other do not use the progressive approach.

Since using the non-progressive approach means not using the progressive approach, we actually compare using to not using the progressive approach, and therefore, this is a one factor study.

5.2.4 Experimental object

The study used selected use cases of the Health Watcher software, a real software, that allows citizens to complain about diseases problems and retrieve information about the public health system, such as the location or the specialties of a health unit. The selected use cases cover several kinds of services, such as, recording, retrieving, and updating data. In addition to the use cases, we also consider requests for functional requirement changes, which are performed during the software implementation, and test-cases that guide the unit tests. We selected five of the nine use cases of the Health Watcher specification, and those five are representative since they cover the same kind of operation performed by the others, which are retrieving information, giving an employee different access grant, and cleaning the database. Moreover, the selected use cases are the most important use cases of the software, and therefore, they would be naturally selected as the first ones to be implemented. The use cases and their scenarios are

- RF01 Retrieve Information — describes several reports the software should implement such as retrieve information about complaints (1), diseases (2), specialties of a health unit (3), and which health units have a specific specialty (4);
- RF02 Record Complaint — describes how to register the different kinds of complaints: food complaint (5), animal complaint (6), and special complaint (7);
- RF10 Login — describes how an employee should login into the system in order to perform restricted operations (8);

RF11 Record Data — describes how to insert and update employees (9), and how to update health units (10);

RF12 Update Complaint — describes how to update data about how a complaint should be handled by the public health system (11).

Note that in these five use cases we identified eleven use-case scenarios to be implemented. In this study, those scenarios are distributed in three iterations, as shown in Section 5.6.

5.2.5 Experimental subjects

The participants of this study are MSc and PhD students taking a graduate course on advanced topics on programming language. The course were specifically offered to perform the study, and therefore, the participants were aware that they were participating of an experimental study and that their data would be used in the study analysis. We expected that most of them had experience in industrial software development. In fact, they answered a questionnaire in order to characterize their academic and industry expertise before the study. More information about their expertise is presented in Section 5.6.

5.2.6 Independent variables

- Implementation of information systems;
- Requirements and design expressed using use cases and UML diagrams;
- Iterative and incremental implementation;
- Aspect-oriented programming with Java and AspectJ, using specific design patterns and frameworks [91, 85, 53];
- Implementation order of persistence and distribution concerns. Although the implementation method does not demand a specific implementation order, we fixed the order to implement persistence before distribution to avoid undesirable bias, which means we are comparing times of these non-functional requirements implemented in a same order. Additional studies should be performed to evaluate the impact of the concerns implementation order.
- Specific technologies to implement web information systems. These technologies are: Java [27], AspectJ [41], databases, JDBC [103], distribution systems, Java RMI [59], Java Servlets [32], and object-oriented analysis and design.

5.2.7 Dependent variables

The time to implement, change, testing, and yield pre and post-validation functional prototypes. We can provide views of these data per use case, per use-case scenario, or per iteration.

5.2.8 Trials design

The software should be implemented using a Use-Case Driven Development (UCDD) approach, which is adopted by the Rational Unified Process (RUP) [39] and several other modern processes. According to UCDD, a software is designed, implemented, and tested based on its use cases. Chapter 4 gives a brief explanation on use cases. To implement a use case or a use-case scenario, programmers should implement parts of the system that are necessary to realize that specific use case or scenario.

Without using the progressive approach, the functional and non-functional requirements associated to a use case or scenario are usually implemented in the same iteration (the non-progressive approach).

On the other hand, when planning the progressive approach, the implementation of some non-functional requirements (persistence and distribution) is schedule after implementing the functional part of the use cases and the user interface. Therefore, use cases are partially implemented in functional iterations, until a functional prototype is finished. At this moment, this prototype is validated by the system stakeholders and, if necessary, changes are made. After validating, the functional prototype evolves to a persistent and distributed system, in two different non-functional iterations.

To avoid undesirable bias we enforced the implementation of persistence before implementing distribution in both approaches. This assures that we are comparing times of these non-functional requirements implemented in a same order. Additional studies should be performed to evaluate the impact of the concerns implementation order.

Figure 5.1 shows the two different development dynamics. In the figure, times x and y have no relation with times x' and y'. Note that, as explained before, when implementing a use case using a progressive approach the functional requirements and nonpersistent (volatile) data management are implemented in a first iteration, called functional iteration. After validating this functional prototype, persistent data management and distribution are implemented in two special iterations (non-functional iterations). When implementing the next iteration, the concerns implemented in the non-functional iterations should be turned-off, in other to implement the new use cases and test the whole system using only nonpersistent data collections. By using this approach a functional prototype, without persistence and distribution, is delivered at the end of each functional iteration to be validated.

On the other hand, if the use case is being implemented without using a progressive approach, both non-functional requirements (persistence and distribution) are implemented in the same iteration of the functional requirements. When implementing the next iteration, non-functional requirements already implemented might affect the new requirements implementation and testing, since they are not turned-off like in the progressive approach.

It is important to mention that the resulting software in both approaches is the same, they have the same functionality and the same non-functional requirements implemented. The only difference between the implementation approaches is the order non-functional requirements are implemented and tested, and the need to implement nonpersistent data collections if using a progressive approach. Therefore, at the end of each iteration (x or x' and y or y') we should have the same software, independent of what implementation approach was used.

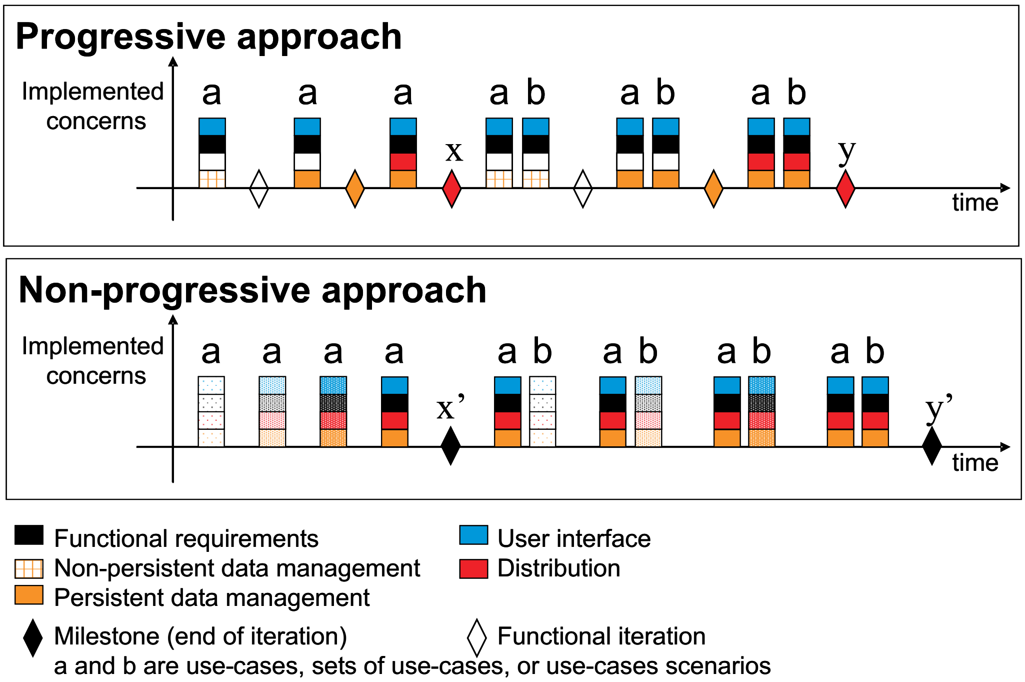

Figure 5.1: Iterations using progressive and non-progressive approaches.

5.3 Preparation

Before starting the study, the subjects answered the questionnaire (Appendix A) about their profiles and experience with software development. The subjects were aware that their data would be used by the experimental study. The subject's data is presented in Section 5.6.

5.4 Analysis

The study analysis compares the collected data from the experimental object trials and control object trials in order to see if the null hypotheses can be rejected.

The study analyzes the impact of the progressive approach on

- Implementation time;
- Requirement changes time (during development);
- Tests execution time;
- Pre-validation functional prototype time;
- Post-validation functional prototype time.

As we have a completely randomized one factor study [104, 40], we use a t-test [40, 104] to analyze the results. The t-test is a statistical test that compares the mean values of two sets of data, and analyzes if their differences are significant. In fact, we can perform two different analyses using the t-test.

5.4.1 Yes-No decision

The idea when analyzing the collected data is to try to reject the null hypotheses by showing that the expected mean values are not the same for a given significance. In this way, we perform the t-test with the samples of the progressive and non-progressive approach to implement the use cases. We consider this a one factor study since using non-progressive approach actually means implementing the use cases without using the progressive approach. The so-called non-progressive approach is our control object. In this way, this test is a yes-no decision, since we can only say if there is a difference between the sample means.

Consider $x_1, x_2, ..., x_n$ the samples (times) of applying the progressive approach, and $y_1, y_2, ..., y_m$ the samples of applying the non-progressive approach. We first define the mean values ($\bar{x}$ and $\bar{y}$) of the samples:

$$\boxed{\begin{array}{l} \bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i \\[1em] \bar{y} = \frac{1}{m}\sum_{i=1}^{m} y_i \end{array}}$$

Now, following the t-test, we define the distribution t_0 in the following way:

$$\boxed{t_0 = \frac{\bar{x}-\bar{y}}{s_p\sqrt{\frac{1}{n}+\frac{1}{m}}}, \text{ where } S_p = \sqrt{\frac{(n-1)S_x^2+(m-1)S_y^2}{n+m-2}}}$$

where S_x^2 and S_y^2 are the variances of the samples:

$$\boxed{\begin{array}{l} S_x^2 = \frac{(\sum_{i=1}^{n} x_i^2)-n\bar{x}^2}{n-1} \\[1em] S_y^2 = \frac{(\sum_{i=1}^{m} y_i^2)-m\bar{y}^2}{m-1} \end{array}}$$

The null hypotheses expect the same mean for time values. Therefore, in order to reject the null hypothesis we want to find if the difference of the means is different from zero, which means they are different, comparable. The test consists in checking if $|t_0| > t_{\alpha,f}$, where $t_{\alpha,f}$ is the t distribution at a α percentage point, or significance level, using f degrees of freedom, where $f = n + m - 2$ [104]. The distribution is tabulated elsewhere [40, 104]. We perform the test at the 0.05 significance level, since we had few participants in the experiment. In other words, we use a confidence level of 95%, which means that we are working with the 95% probability of using values that are close to the mean values.

However, this test based in a yes-no decision might not be enough. There is a more effective way to analyze data than just saying if the samples mean are different or not.

5.4.2 Confidence interval

We can also analyze the collected data by determining the confidence interval [40] for the mean difference of progressive and non-progressive times ($\bar{x} - \bar{y}$). If the interval contains zero, which means that zero is a possible value for the difference, the samples are not significantly different. This is a more effective way to analyze data than just saying if the samples mean are different or not. A narrow confidence interval indicates a high degree of precision. On the other hand, a wide confidence interval indicates that the precision is not high.

We first used hypothesis testing to identify if the samples are significantly different. When the null hypothesis is false, the sample means are significantly different and it is not necessary any additional analysis. On the other hand, when the null hypothesis is true, instead of just saying that the sample means are not significantly different, we also determine the confidence interval to analyze the degree of precision of this analysis. The confidence interval is defined in the following way:

1. Compute the samples means $\bar{x}$ and $\bar{y}$, as defined in Section 5.4;
2. Compute the sample variance S_x^2 and S_y^2, also from Section 5.4;
3. Compute the mean difference: $\bar{x} - \bar{y}$;
4. Compute the standard deviation of the mean difference:

$$S = \sqrt{\tfrac{S_x^2}{n} + \tfrac{S_y^2}{m}}$$

5. Compute the effective number of degrees of freedom:

$$v = \frac{(S_x^2/n \, + \, S_y^2/m)^2}{\frac{1}{n+1}(S_x^2/n)^2 + \frac{1}{m+1}(S_y^2/m)^2} - 2$$

6. Compute the confidence interval for the mean difference:

$$(\bar{x} - \bar{y}) \mp t_{[1-\frac{\alpha}{2};v]} S$$

   where $t_{[1-\frac{\alpha}{2};v]}$ is the $(1 - \frac{\alpha}{2})$-quantile of a t-variate with v degrees of freedom;

7. If the confidence interval includes zero, the difference is not significant at $100(1-\alpha)\%$ confidence level. Similar to the null hypothesis test, we used a significance level at 0.05 that correspond to a 95% confidence level, which means that we are working with a 95% probability that the population mean are in the interval.

This alternative analysis does not change the result of the previous hypothesis test. The idea is to provide additional data to decision-makers, in this case the degree of precision.

5.5 Threats to Validity

This section discusses how valid are the results and if we can generalize them to a broad population. There are four kinds of validity. Internal validity defines if the collected data in the study resulted from the dependent variables and not from an uncontrolled factor. Conclusion validity is related to the ability to reach a correct conclusion about the collected data, to the used statistical test, and how reliable are the measures and the collected data. Construct validity is concerned to assure that the treatment reflects the cause and the results reflect the effect, for example, without being affected by human factors. Finally, external validity is concerned with the ability to generalize the results to an industrial environment.

5.5.1 Internal Validity

The experimental subjects are MSc and PhD students of a graduate course. The students are from the Software Engineering area, so they have some experience in developing software. In fact, most of them have experience in the software industry, which contributes for being a representative set of software developers. However, despite most of the subject have experience in the software industry, their experience is no more than five years for most of them.

Despite being separated in two groups, one that uses progressive approach and the other that does not use, both subjects group used the same aspect-oriented technology, AspectJ. Therefore, we did not expect the subjects to be unhappy or discouraged in performing or not the treatment, since the resultant software is essentially the same for both situation. In addition, the study execution is necessary for the conclusion of the course, and therefore, we did not expect anybody to quit the study.

One confounding factor could be the subject's experience. In fact, the subjects filled a questionnaire about their experience and expertise in academia and industry. These data, presented in Section 5.6, are used in order to identify this possible confounding factor. Since there were few subjects to perform the study (6 in one group and 7 in the other), we randomly distributed the subjects to treatments (see Section 5.2) instead of defining blocks, which would decrease the number of samples to be compared.

5.5.2 Conclusion Validity

A mentor was always present during the study execution to guarantee the correct data collection and implementation of the treatment. The study uses a t-test to compare the data between applying the treatment (progressive implementation) and the control object (non-applying the treatment, also called non-progressive implementation). The t-test is more suitable to this study since it is concerned in comparing the mean values of the sample data of the two groups, instead of comparing a sample from one group with a sample from the other, with can provide undesirable bias. For example, data from two specific subjects cannot be compared to each other since they might have different expertise. On the other hand, when considering the mean value, we are comparing data from one group with the data from the other, and not specific subjects, which can decreases the subject's expertise impact.

5.5.3 Construct Validity

The subjects applied the treatment to selected use cases of a real information system using an execution plan, which explains how to apply the treatment. In addition, they performed a dry run to make clear how the treatments should be implemented and how data should be collected.

In fact, we performed a previous study, however, without the concern to provide exact guides to the subjects on how to implement the software, and how to collect the data. This actually resulted in incomparable data, since the subjects collected data in different way. This shows how useful are these threats of validity in order to guarantee the validity of the study results.

5.5.4 External Validity

One expected result of this study is to guide programmers on when to use the progressive approach. As we used randomization to separate the subjects in two groups, we expect to decrease the confounding factors, since the most important is the subjects' expertise. However, the limited number of subjects does not allow generalization outside the scope of the study. On the other hand, we expect that the results, including the subjects' feedback, can be used as guidelines to better implement aspect-oriented software.

The terminology, implementation process, and technology used in the study are currently used in the software industry, and therefore, are adequate to our objective. We did not have problems with temporary issues, since the study was performed in one of the university laboratories, specially reserved to the study execution.

Although the results are limited by the narrow scope we have, we believe that a considerable contribution is the study design. This framework can guide other studies in order to evaluate the progressive approach with more general and conclusive results and can also support other kind of studies, for example to identify the impact of other factors variation, evaluating alternative approaches to implement aspect-oriented software.

5.6 Execution

As previously mentioned, this study was performed during a graduate course. In the first half of the course we discussed several papers about aspect-oriented programming [42, 45, 46, 47, 43, 22, 54], AspectJ [41, 83, 30, 97], specific design patterns for the kind of system they implemented in the study [53, 2, 82, 86], and the use of these design patterns with AspectJ [91, 85, 48]. We provided exercises[1] about AspectJ, JDBC, and RMI, in order to give the subjects a minimum knowledge about these technologies, in particular to the ones that did not have enough contact with them. We also performed a dry run to give the subjects a chance to familiarize with data collection and plan execution.

The study was executed during the second half of the course, according to the schedule in Table 5.1. As previously mentioned in the experimental object definition (Section 5.2), there are 11 use-case scenarios to be implement.

[1]Those exercises can be found at `www.cin.ufpe.br/~scbs/talp1/nivelamento`.

Use case	Scenario to implement	Scenario id
Iteration 1 — 23/01/2004		
[RF02] Register Complaint	Food complaint	01
[RF01] Retrieve Information	Specialties of a health unit	02
Iteration 2 — 06/02/2004		
[RF02] Register Complaint	Animal complaint	03
[RF10] Login		04
[RF11] Register Data	Insert employee	05
[RF01] Retrieve Information	Complaint	06
Iteration 3 — 20/02/2004		
[RF12] Update Complaint		07
[RF01] Retrieve Information	Health units with a specialty	08
[RF01] Retrieve Information	Disease	09
[RF02] Register Complaint	Special	10
[RF11] Register Data	Update health unit	11
12/03/2004		

Table 5.1: Study schedule.

We provided several documents to the teams, including the selected Health Watcher's use-cases specification, class diagram, test-cases, and implementation plan stating the order in which the use-case scenarios should be implemented in each iteration (Table 5.1). We also provided the user interface code, which are HTML documents and Java Servlets, in order to speed up the study, since it should be performed during the second half of the course. Since the progressive approach demands implementing the user interface code at the same time the functional requirements are implemented, whether the progressive approach or the non-progressive approach is used, the user interface code is implemented at the same point. Therefore, this should not affect the study results. Moreover, since at the time of the study the tool support was not implemented, we also provided nonpersistent data collections (see Section 3.2) that would be automatically generated (see Chapter 6).

As previously mentioned, we also simulated requirement changes during development. The requirement changes are the following:

- In order to simulate a customer's request to change the requirement after using an implemented use-case scenario for the first time, we requested changes to add attributes to classes
  - After implementing the Food complaint scenario (01) of the use case Register Complaint ([RF02]);
  - After implementing the Animal complaint scenario (03) of the use case Register Complaint ([RF02]).
- In order to simulate a change in the system design after presenting the implemented use case for the first time, we requested a change to extract a class from three existing classes

– After implementing the Food complaint scenario (01) of the use case Register Complaint ([RF02]).

In fact, there are two change requests, since two of them are requested at the same time during Scenario 01 implementation. Those changes were chosen because they are pretty common changes, like adding new information or modifying the design model. It is important to mention that as the resulting software is the same independent of the used approach, we did not perform any maintenance change. In fact, the progressive approach tries to increase productivity by allowing the identification of requirement changes during the software development.

5.6.1 Questionnaire data

The first step before starting the study is to apply the questionnaire to the subjects in order to collect information about their experience. As previously mentioned, all the subjects were aware that the collected data would be used in an experimental study. In the following pages, Figure 5.2 depicts the academic expertise and Figure 5.3 depicts the industry expertise of the thirteen subjects according to the score in Table 5.2.

Score	**Expertise**	
1	Only in this course	(OITC)
2	Less than 6 months	(< 6m)
3	Between 6 months and 2 years	(6m-2y)
4	Between 2 and 4 years	(2y-4y)
5	Between 4 and 6 years	(4y-6y)
6	More than 6 years	(> 6y)

Table 5.2: Expertise scores.

It is interesting to notice that most of the subjects had their first contact with AspectJ in this course. On the other hand, two of the subjects used AspectJ in academic environment in the past six months, and other subject between six months and two years. A fourth subject actually used AspectJ in the past six months at industrial environment.

All the subjects have at least between six months and two years of experience in industrial software development. However, some of them had no contact with some technologies used during the study, such as RMI and JDBC. As previously mentioned, we provided exercises to help subjects that did not have enough experience with JDBC and RMI, reducing possible confounding factors.

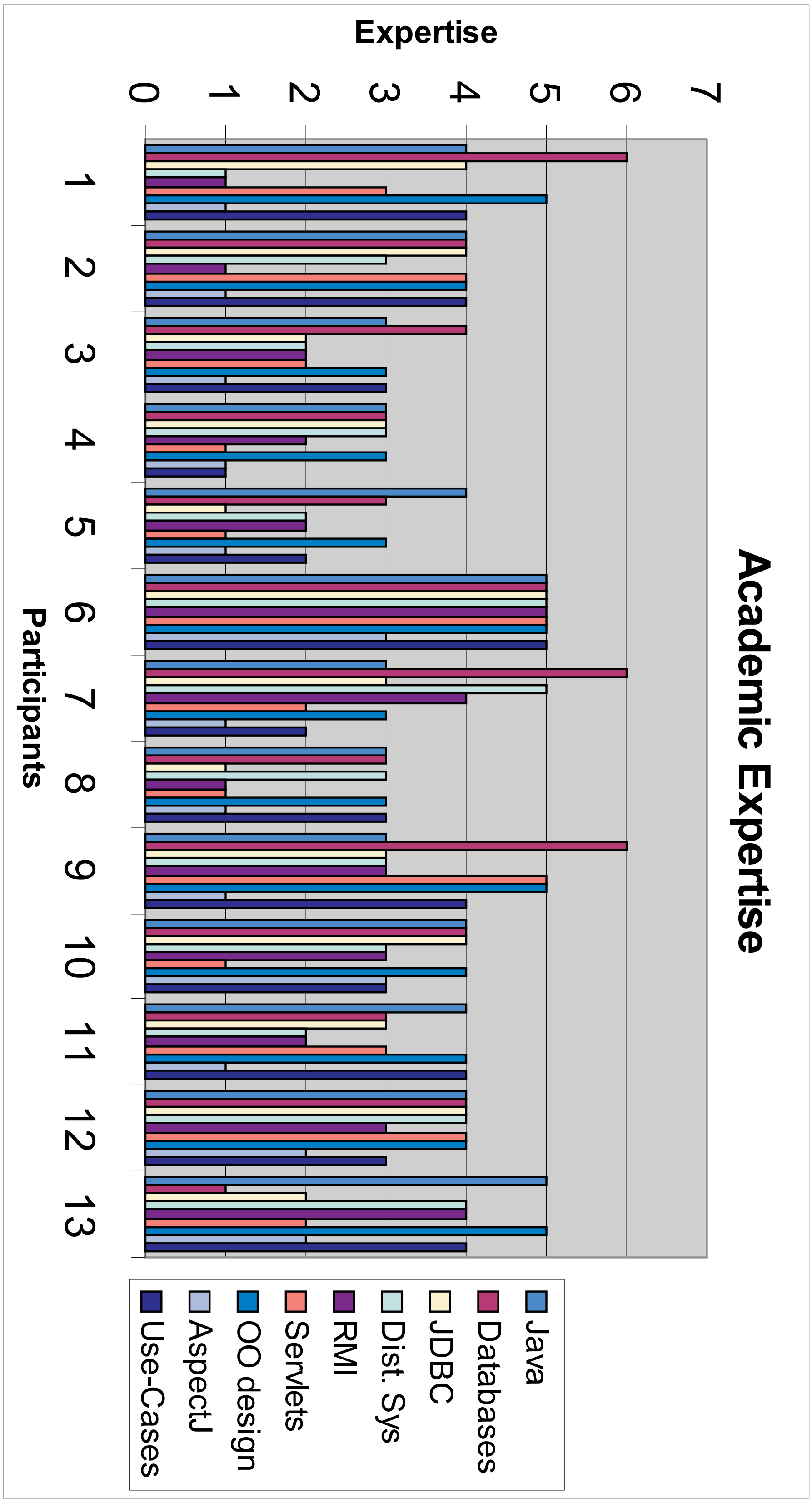

Figure 5.2: Subjects's academic expertise.

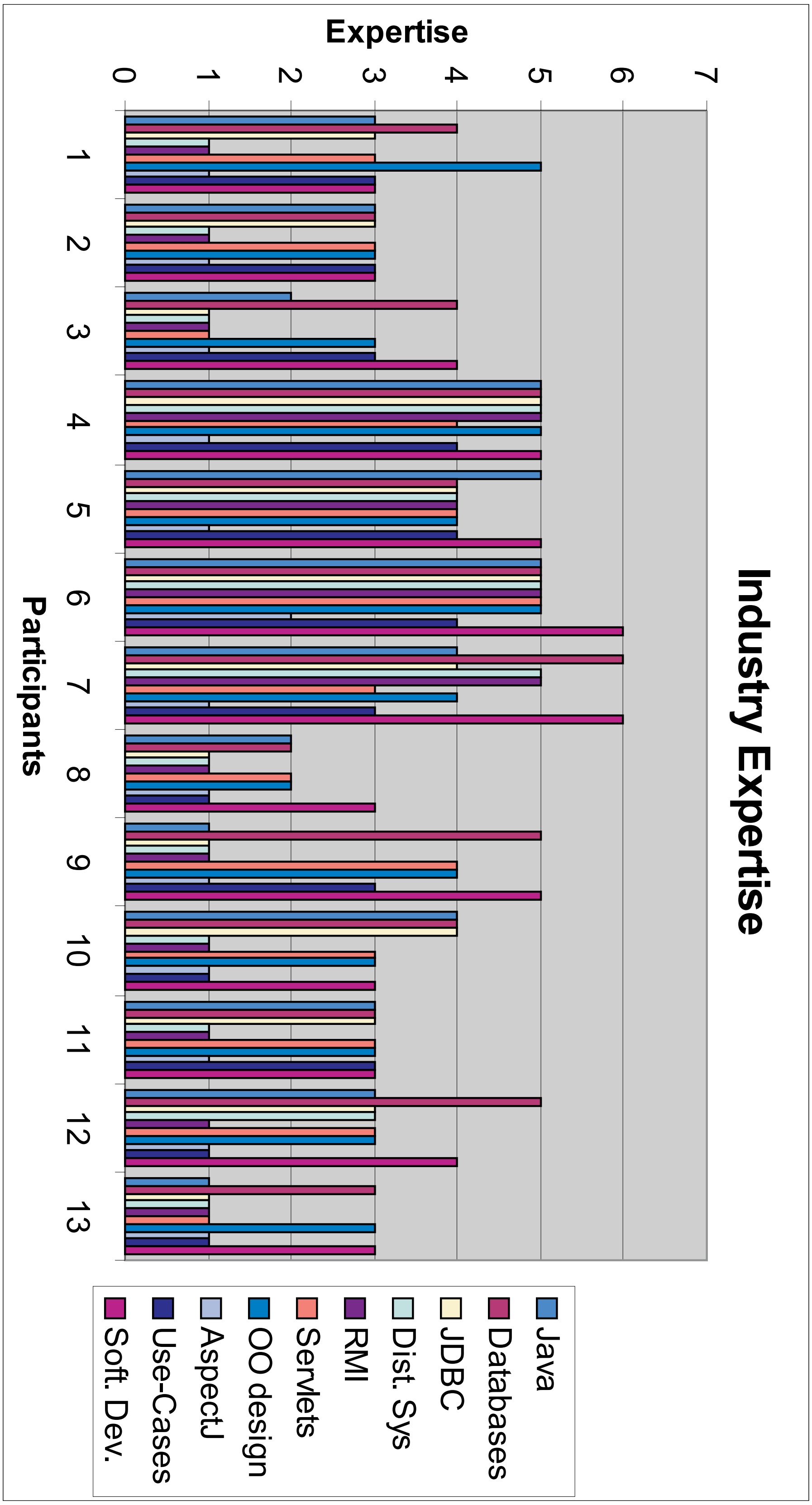

Figure 5.3: Subjects's industry expertise.

5.6.2 Study data

Tables 5.4, 5.5, and 5.6 present each subject's time, in minutes, to implement, test, and change requirements during the three iterations. The total time to complete each iteration is presented by Table 5.7. Tables 5.8 and 5.9 present the time, in minutes, to yield a functional prototype per use-case scenario for each subject. Table 5.10 presents the time in minutes to yield a functional prototype after requesting a requirement change in Scenarios 1 and 3. These tables use the legend presented in Table 5.3.

Code	Description
S. ID	Suject ID
Impl	Implementation
Test	Test
RC	Requirement change
Mean	Sample mean
Diff (%)	Difference in % from the other approach

Table 5.3: Tables legend.

NON-PROGRESSIVE			
S. ID	**Impl**	**Test**	**RC**
1	605	170	198
2	497	530	279
3	987	143	593
5	377	155	152
9	885	381	333
10	234	47	237
11	784	337	377
Mean	624	252	310
Diff (%)	+13.4	+41.4	+789.5

PROGRESSIVE			
S. ID	**Impl**	**Test**	**RC**
4	451	117	30
8	705	191	38
12	694	251	38
7	745	279	57
6	323	101	28
13	385	130	18
Mean	551	178	35
Diff (%)	-11.8	-29.3	-88.8

Table 5.4: Iteration 1 data.

The difference (Diff (%)) from the other approach can be read as an increasing or decreasing in time from the other approach. For example, Table 5.4 presents a increase of 789.5% in the requirement changes time when using the non-progressive approach, or a decrease of 88.8% in the requirement changes time when using the progressive approach.

NON-PROGRESSIVE			
S. ID	**Impl**	**Test**	**RC**
1	252	150	18
2	372	294	39
3	405	228	30
5	199	124	22
9	316	127	20
10	180	93	14
11	462	229	59
Mean	312	178	29
Diff (%)	+3.4	+10.7	+124.9

PROGRESSIVE			
S. ID	**Impl**	**Test**	**RC**
4	359	110	13
8	273	334	14
12	294	151	11
7	347	94	12
6	276	123	16
13	263	152	11
Mean	302	161	13
Diff (%)	-3.3	-9.7	-55.5

Table 5.5: Iteration 2 data.

NON-PROGRESSIVE		
S. ID	**Impl**	**Test**
1	218	69
2	242	218
3	358	110
5	163	63
9	171	274
10	237	61
11	529	91
Mean	274	127
Diff (%)	-3.7	+19.0

PROGRESSIVE		
S. ID	**Impl**	**Test**
4	229	123
8	368	167
12	234	118
7	300	124
6	271	50
13	306	56
Mean	285	106
Diff (%)	+3.9	-16.0

Table 5.6: Iteration 3 data.

NON-PROGRESSIVE			
S. ID	**IT 1**	**IT 2**	**IT 3**
1	973	420	287
2	1142	691	460
3	1718	661	468
5	674	338	226
9	1599	458	445
10	501	283	298
11	1427	743	620
Mean	1147.7	513.4	400.6
Diff (%)	+51.7	+8.9	+2.4

PROGRESSIVE			
S. ID	**IT 1**	**IT 2**	**IT 3**
4	592	479	352
8	921	614	535
12	975	452	352
7	1078	452	424
6	447	409	321
13	526	424	362
Mean	756.5	471.7	391.0
Diff (%)	-34.1	-8.1	-2.4

Table 5.7: Total iterations data.

	Use-case scenarios										
S. ID	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
1	603	172	38	45	131	188	53	53	70	61	50
2	628	235	76	76	315	185	100	80	123	82	75
3	937	188	67	100	159	305	62	102	146	87	71
5	358	164	64	48	81	123	46	28	66	51	35
9	920	346	51	75	134	178	71	116	107	119	32
10	281	23	50	40	109	70	31	126	85	35	21
11	839	211	85	49	264	286	97	166	158	130	69
Mean	652.3	191.3	61.6	61.9	170.4	190.7	65.7	95.9	107.9	80.7	50.4
Diff (%)	+480	+131	+83	+117	+459	+423	+412	+233	+141	+156	+159

Table 5.8: Times to yield pre-validation prototype without progressive approach.

	Use-case scenarios										
S. ID	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
4	107	89	68	35	34	37	9	28	43	28	51
8	136	136	34	41	31	37	17	38	60	58	13
12	136	11	13	15	51	35	20	22	42	23	15
7	141	179	32	32	18	77	6	30	52	28	8
6	74	18	24	27	28	17	10	24	33	42	7
13	81	63	31	21	21	16	15	31	39	10	23
Mean	112.5	82.7	33.7	28.5	30.5	36.5	12.8	28.8	44.8	31.5	19.5
Diff (%)	-83	-57	-45	-54	-82	-81	-81	-70	-58	-61	-61

Table 5.9: Times to yield pre-validation prototype with progressive approach.

NON-PROGRESSIVE		
S. ID	**Scenario 1**	**Scenario 3**
1	801	56
2	907	115
3	90	97
5	510	86
9	1253	71
10	478	64
11	1216	144
Mean	750.7	90.4
Diff (%)	+398.3	+94.5

PROGRESSIVE		
S. ID	**Scenario 1**	**Scenario 3**
4	137	81
8	174	48
12	174	24
7	198	44
6	102	40
13	119	42
Mean	150.7	46.5
Diff (%)	-79.9	-48.6

Table 5.10: Times to yield post-validation prototype.

5.6.3 Statistical analysis

As mentioned in Section 5.4 we use a t-test [40, 104] to analyze the data. The test consists in checking if the samples' mean difference is significant. This can be made by a hypothesis testing, and complemented by determining the confidence interval, as mentioned in Section 5.4.

Iteration	$H0_1$	$H0_2$	$H0_3$	$H0_4$	$H0_5$
1	TRUE	TRUE	FALSE	FALSE	FALSE
2	TRUE	TRUE	FALSE	FALSE	FALSE
3	TRUE	TRUE	—	FALSE	FALSE

Table 5.11: Null Hypotheses test.

We first used hypothesis testing to identify if the samples are significantly different. Table 5.11 presents the null hypotheses tests per iteration, where FALSE means the null hypothesis in question is false. When the null hypothesis is false, the sample means are significantly different. According to the presented results, at all iterations, implementation and tests time using a progressive approach are not significantly different from the non-progressive approach. On the other hand, the statistical test rejected the null hypotheses for the time to change requirements, and yield pre and post-validation prototypes.

Table 5.12 presents the values of the $|t_0|$ distribution of the t-test for each null hypothesis per iteration. The test consists in checking if $|t_0| > t_{\alpha,f}$, where $t_{\alpha,f}$ is the t distribution at a α percentage point, or significance level, using f degrees of freedom, where $f = n + m - 2$, in our case, $f = 6 + 7 - 2$ and $\alpha = 0.05$. Therefore, we should compare $|t_0|$ values with the $t_{0.05,11}$ distribution, which is 2.201. The $H0_4$ and $H0_5$ values in the third iteration are the same since there were no requirement changes.

Iteration	$H0_1$	$H0_2$	$H0_3$	$H0_4$	$H0_5$
1	0.555	0.985	4,551	5.960	3.497
2	0.221	0.385	2.470	4.649	4.858
3	0.188	0.522	—	4.601	4.601

Table 5.12: $|t_0|$ values for the Null Hypotheses test.

The $|t_0|$ values for the requirement changes in the first iteration and for the pre and post-validation prototypes in all three iterations are significantly different even using a higher confidence level. The $t_{0.025,11}$ distribution, which is the distribution for a 99% confidence level, or at the 0.025 significance level, is 3.106.

In order to allow a more effective analysis, we also determined the confidence interval, at the 0.05 significance level, presented by Table 5.13, for the mean difference when the null hypotheses could not be rejected by the test. We can notice wide confidence intervals, which indicate that the null hypotheses tests were obtained with a low precision. In fact, we expected benefits from implementing and testing the software with the progressive approach. When progressively implementing and testing functional requirements, the programmer does not have to worry with persistence and distribution issues and vice-versa, decreasing implementation and test complexity. These wide confidence intervals suggest that other studies should be performed to better evaluate the progressive approach impact in implementation and tests.

Iteration	$H0_1$ — implementation time	$H0_2$ — test time
1	(-206.6 ; 353.9)	(-86.2 ; 233.6)
2	(-88.6 ; 109.2)	(-81.5 ; 115.9)
3	(-131.2 ; 109.9)	(-61.2 ; 101.6)

Table 5.13: Confidence interval for Hypotheses $H0_1$ and $H0_2$.

We can notice some interesting data at the table's data. Before commenting these data, it is important to mention that the first iteration is the hardest one, which can be noticed by the mean values of amount of time of each iteration, presented in Table 5.7. Notice the time to implement Iteration 1, which is higher than the sum of the others, in the non-progressive approach, and almost the sum of the other in the progressive approach.

Despite not being significantly different, there are some interesting data about implementation and testing times that might help understanding the study. For example, we can clearly notice a discrepancy at Table 5.4 regarding Subject 10's implementation and testing time, which were much faster than the others. In fact, implementation time was more than twice faster than the group mean, and testing time was more than five times faster. He definitely has superior programming skills, which can be confirmed by his academic and industrial expertise (Figures 5.2 and 5.3) that show he is one of the most experienced with AspectJ. On the other hand, his skills did not make difference when performing requirement changes, which reinforces the huge impact requirement changes might have on the development process. Such discrepancy did not happened with the progressive approach group.

Table 5.4 also presents a huge difference (789.5%) between the approaches' requirement changes time. This happened because the requirement changes at this iteration had a great impact in the persistence code. For example, one of them demanded changing the database scheme as well as writing migration scripts to move the data already inserted to the new scheme. On the other hand, the progressive approach does not implement persistent code until validating the functional requirement, when the changes are usually required. However, despite this huge difference at requirement changes, when considering the whole iteration time (see Table 5.7), the difference decreases to 51.7%, which is significantly different when performing a t-test. However, this value could be worst if we had more requirement changes.

Besides the null hypotheses there are also alternative hypotheses (see Section 5.2). Table 5.14 shows the alternative hypotheses test based on the values of the $|t_0|$ distribution already presented by Table 5.12. There are two alternative approaches set ($H1_{1..5}$ and $H2_{1..5}$) stating that the time to implement, test, change, and yield pre and post-validation prototypes using a progressive approach is different ($H1_{1..5}$), and smaller ($H2_{1..5}$) than using a non-progressive approach. When there is a significance difference between the approaches, the progressive approach times are smaller than the

Iteration	$H1_1$	$H1_2$	$H1_3$	$H1_4$	$H1_5$
1	FALSE	FALSE	TRUE	TRUE	TRUE
2	FALSE	FALSE	TRUE	TRUE	TRUE
3	FALSE	FALSE	—	TRUE	TRUE
Iteration	$H2_1$	$H2_2$	$H2_3$	$H2_4$	$H2_5$
1	FALSE	FALSE	TRUE	TRUE	TRUE
2	FALSE	FALSE	TRUE	TRUE	TRUE
3	FALSE	FALSE	—	TRUE	TRUE

Table 5.14: Alternative Hypotheses test.

non-progressive approach times (see sample data tables). For example, Table 5.6 shows that the implementation time of Iteration 3 is 3.9% worst when using the progressive approach. However, neither the null hypothesis nor the alternatives considered the mean differences for implementation time of Iteration 3 significantly different. In fact, this is the only case where the collected data shows a progressive time, in minutes, higher than the non-progressive time.

The data presented by Tables 5.5 and 5.6 show that these iterations were simpler than the first one. In fact, Iterations 2 and 3 had less use-case scenarios to implement and most of the aspects were implemented in the first iteration, which requires only simple modifications, as the software is incremented.

There was another requirement change in the second iteration, however, this change did not demand the same kind of tasks performed in the changes of the first iteration, such as changing the database scheme, and therefore, writing a data migration script. Despite being simpler than the first iteration, the single requirement change performed at the second iteration was also significantly different, according to Table 5.14, showing that the software development can be benefited from the progressive approach when there requirement changes during implementation activities. Therefore, although there is not a significant difference between the progressive and non-progressive approaches during implementation and tests activities, the progressive approach can be used to avoid unnecessary delay if there are requirement changes. Moreover, the progressive approach decreases the implementation complexity, since it does not deal with all the concerns at the same time, as discussed at the end of this chapter.

Another important data was about the times to yield pre and post-validation prototypes. The t-test showed that the times to yield pre and post-validation prototypes of all use-case scenarios (see Tables 5.8, 5.9, and 5.10) were significantly different, and therefore, the progressive approach has unbeatable results. In fact, this was expected since early validation of functional requirements is one of the pillars of the progressive approach, which is reached by first abstracting the implementation of some non-functional requirements. This early validation anticipates requirement changes, also helping to understand the problem before implementing some non-functional requirements. Moreover, the effort to create such prototype is lower, decreasing the budget impact, for example, if the requirements were not well understood by the requirements engineering or the customers had just change his mind.

5.6.4 Qualitative data

After finishing the study, we applied another questionnaire with more general questions in order to get some feedback from the subjects about the study, the technology used, and the implementation approaches. From a total of 13 subjects, 12 (92%) said that AOP and AspectJ helped the development, and 1 (8%) stated that AOP involves several new constructs, such as join points and pointcuts, that complicate debugging. He concludes that it is maybe better to use OO and design patterns. According to this subject, he had this feeling because he has a great experience with OO. The difficulty to learn a new paradigm (AO) was reported by 11 (85%) subjects. Another interesting information collected about the AspectJ language is about the debugging support, where 5 (38%) subjects complained about AspectJ's debugging. In fact, this is a limitation of the AspectJ tool used, but new versions of the tool have debugging support.

We also asked the subjects if they felt the progressive approach increases productivity. An expressive number of subjects, 7 (54%), explicitly said that they believe so, 2 (15%) suggested to believe, 3 (23%) gave neutral opinions, and 1 (8%) suggested that the progressive approach does not increase productivity.

Additionally, we asked to the 6 subjects that used the progressive implementation approach if they would use the approach in a real software development process. Again, the progressive approach had a great feedback from the subjects, where 4 (67%) would use the progressive approach just like they used in the study, and 2 (33%) would use variations of it.

5.7 Conclusions

In order to evaluate if the progressive approach is appropriate for a given project, a manager or development leader should balance the following forces derived from the study:

- The progressive approach helps to increase implementation productivity by early validating functional requirements, before implementing non-functional requirements. The study showed that there is a great difference between using and not using the progressive approach when there are requirement changes. In fact, requirement changes are common during implementation activities.

- Requirement changes might emerge from different stakeholders, specially after functional requirements validation, and the progressive approach avoids wasting effort writing non-functional code that is wrong or is supposed to be changed. Examples of causes of requirement changes are the following: requirement faults that lead to design faults, and therefore, to implementation faults; underestimated requirements that lead to bad design decisions; the lack of experience of developers; the customer keeps changing his mind, mainly when seeing a functional prototype.

- In fact, good practices and techniques in requirements, analysis, and design help to avoid some of these requirement changes. On the other hand, some requirement changes cannot be avoided by using good practices. For example, some faults during requirements, analysis, and design activities cannot be identified until implementing the software. Furthermore, customer requests after seeing the prototype are unpredictable.

- The study showed that the time to yield a functional prototype using a non-progressive approach is larger to using the progressive approach in every use-case scenario, effectively providing early validation of functional requirements.

- Although the times to implement and test software using the progressive approach and the non-progressive approach in this study were not different, this means that there is not an overhead in the progressive approach implementation and testing activities. Therefore, the progressive approach can be used to perform requirement changes earlier, if technical problems are detected or customer requests are made after seeing the prototype, which avoids wasting effort writing non-functional code that is wrong or is supposed to be changed.

- Qualitative data from the subjects gave an important feedback about the progressive approach, which might suggest its use.

Besides all those conclusions, another contribution of this study is the documentation of the performed study, resulting in a study framework that allows replication in order to give more evidence of our results. In fact, some variables of this study might be changed to evaluate other factors.

Chapter 6

Tool support

This chapter presents the first version of a tool that supports the aspect-oriented implementation method by automatically generating part of the aspects and classes necessary to use the aspect framework. The tool aims to increase the method's productivity.

Code generation and refactoring tools have been quite useful for developing object-oriented systems [24, 17]. They increase development productivity by automating tedious, repetitive, and error-prone tasks. By reducing the number of programming errors, they also help to improve software quality.

Based on our experience [91] developing AspectJ systems, we believe that aspect-aware code generation and refactoring tools can bring similar benefits for the development of aspect-oriented systems as well. Although aspect-oriented languages such as AspectJ provide some of the power of metaprogramming constructs, code generation tools are still necessary.

In fact, code generation tools can generate part of the implementation of specific AspectJ patterns, since they might follow the same structure in several systems, such as the aspect framework and the aspect patterns identified in Chapter 3.

Our approach followed the same approach used by others [13, 19], where they consider program transformation as a unifying concept for code generation and refactoring [24]. A refactoring comprises several behavior preserving changes on the program, but does not add new functionalities. A generator, on the other hand, introduces new functionalities. With such a unifying view, AspectJ's transformations may introduce new code and modify existing one as long as the semantics of the original program is preserved. Similarly, we define transformations that manipulate both aspects and classes. This is the main point to consider when adapting a previous work [19, 13] on developing similar tools for Java.

In fact, the scope of this thesis contains a tool to automatically generate aspects, instead of and does not address aspects refactoring. Actually, as our approach considers program transformation as a unifying concept for both code generation and refactoring, the structure we provide can also be used for defining refactorings. However, in order to define a refactoring, one has the non-trivial task of analyzing the code to evaluate some preconditions. This work has been carried out elsewhere [14, 15].

6.1 Java transformations

Our approach to implement such a tool is to extend an existing Java transformation tool (JaTS) [13, 19] in order to provide the ability to manipulate AspectJ programs. In fact, JaTS stands for Java Transformation System. This system consists of a transformation language and a transformation engine to perform the program transformation. The language is a Java extension with meta-programming constructs, such as meta-variables, optional, conditional, iterative constructs, and executable declarations that can have access to lower level code structures. The transformation engine is responsible for performing the transformation itself.

A transformation defines *source* and *target* templates using the transformation language, and the transformation engine uses the source template to match the source code to be transformed. When there is a match, the engine uses the target template in order to generate a type as specified by the transformation language constructs. The generated type can be a new type or a new version of the original type.

For example, the following piece of code specifies a source template T1_lhs.JaTS[1].

```
#[ PackageDeclaration: #PD; ]#
ImportDeclarationSet: #IDS;
ModifierList: #M class #NAME #[ extends #SC_NAME ]#
                             #[ implements NameList: #I_LST]# {

  FieldDeclarationSet: #ATTRS;
  MethodsDeclarationSet: #MTDS;
}
```

The construct PackageDeclaration:#PD matches the package clause of a Java type, which is referenced in the example by the JaTS variable #PD. Similarly, #IDS, #M, #NAME, #SC_NAME, #I_LST, #ATTRS, #MTDS are, respectively, variables that reference the set of imports declarations, list of modifiers, class name, super-class name, implemented interfaces name list, set of field declarations, and set of methods declarations of a Java class. Actually, these variables can be used in the context of any Java type; however, the use of the class Java word restricts the matches to Java classes. Any construct that appears between #[and]# is optional, which means the source template should match types with this construct or not. In the example above, the classes matched by the source template might have or not package, extends, or implements clauses. As the import declaration set might be empty, the matched type does not need to define any import.

The target template T1_rhs.JaTS[2] is implemented by the next piece of code.

```
#[ PackageDeclaration: #PD; ]#
public class #< #NAME.addSuffix("RepositoryArray") >#
      implements #< #NAME.addPrefix("I").addSuffix("Repository") >#  {
  private #NAME[] #< #NAME.addSuffix("s").toVariableName() >#;
  private int index;
  // methods to insert, remove, update and retrieve
  // objects of type #NAME using a Java array
}
```

This target template generates a class that has its name based on the name of the class matched by the source template. The generated class defines a Java array to store objects of the matched type. Actually, the matched type would be a basic class and the generated type a nonpersistent data collection. Note the use of the implements clause in the target template that forces the generated type to implement a business-data interface, which is also generated by a similar transformation.

When applying this transformation to the following Account class

```
package accounts;
import util.Address;
public class Account {
  // ...
}
```

[1] lhs stands for left hand side in a transformation.

[2] rhs stands for right hand side.

JaTS generates the `AccountRepositoryArray` class.

```
package accounts;
public class AccountRepositoryArray implements IAccountRepository {
  private Account[] accounts;
  private int index;
  // insert, remove, update and retrieve accounts in a Java array
}
```

6.2 AspectJ transformations

The approach for developing refactoring and code generation tools for AspectJ is the same as the one presented in the previous section. JaTS is extended in order to support AspectJ's constructs, yielding AJaTS — AspectJ Transformation System.

JaTS uses JavaCC [56] to define a parser that creates an abstract syntax three (AST) of Java objects as nodes representing the parsed program, source and target templates. Visitors [26] visit these nodes in order to find a match according to the source template and perform the transformation according to the target template.

In order to obtain AJaTS, we have to extend the language and the engine. In this way, the JaTS parser and its nodes are extended to add AspectJ syntax and the nodes representing AspectJ's constructs. The visitors responsible for manipulating the AST, performing the engine operations, are also extended. We used bootstrapping to modularize the extensions made to JaTS. A weaver was implemented in order to integrate AspectJ's constructs to the JaTS parser yielding the AJaTS parser. Likewise, AspectJ aspects are defined to modify some JaTS AST nodes and visitors. Besides that, new nodes to model AspectJ's constructs are separately defined. There is a software company currently improving JaTS and we expect to easily extend improved versions by using aspects, auxiliary classes, and the parser weaver.

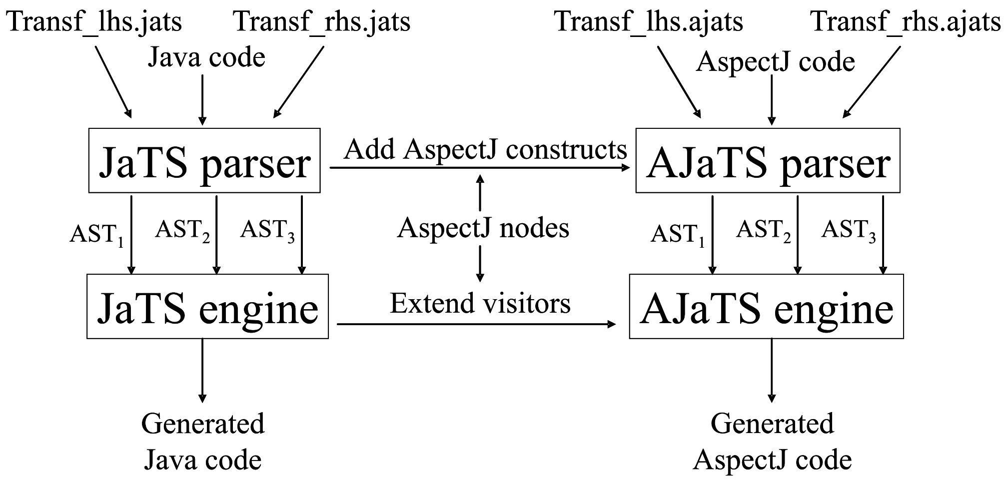

Figure 6.1: JaTS versus AJaTS.

Figure 6.1 depicts the differences between JaTS and AJaTS through the extension made and the code generation support they provide.

Since AspectJ is a superset of Java, AJaTS is a superset of JaTS. Any transformation supported by JaTS is also supported by AJaTS.

6.2.1 Generating aspects with AJaTS

As explained in Chapter 3 the distribution concern is divided in two aspects: client-side and server-side. In order to generate these aspects we defined transformations in AJaTS. Transformations related to the distribution concern use the facade class to generate the aspects and the remote interface.

The following source code matches the facade class and is used as the left hand side of distribution aspects generation.

```
PackageDeclaration:#PD_FACADE;
ImportDeclarationSet:#IDS_FACADE;
public class #FACADE #[ extends #SC_FACADE ]#
                     #[ implements NameList:#IFS_FACADE ]#  {
  FieldDeclarationSet:#ATTRS_FACADE;
  InitializerSet:#IS_FACADE;
  ConstructorDeclarationSet:#CDS_FACADE;
  MethodDeclarationSet:#MTDS_FACADE;
}
```

In fact, this source template cannot distinguish the facade class from other classes. This template potentially matches any class of the program. Therefore, it is necessary an interaction with the programmer in order to identify the facade class, or more generally, the class whose object should be distributed.

Server-side aspect

Before presenting the server-side target template, two other target templates should be executed in order to generate a remote facade interface and to implement the singleton design pattern in the facade class. The following target template is responsible for generating a remote interface.

```
package ##TARGET_PACKAGE##;
ImportDeclarationSet:#IDS_FACADE;
import java.rmi.RemoteException;
import java.rmi.Remote;
public interface #< #FACADE.addPrefix("IRemote") >#
      extends Remote {
  forall #md in #MTDS_FACADE {
    #< #md.addException("RemoteException") >#;
    #< #md.toInterfaceMethodDeclaration()  >#;
  }
}
```

This template uses an iterative declaration (`forall`) to access the facade methods adding the `RemoteException` in their `throws` clause and changing their declaration according to interface definition, where only the signature is presented.

Note that the construction between a pair of ##'s is not an AJaTS construction. This construct has to be replaced by a value specific to the system being generated. In this case, the programmer has to provide the target package of the aspects. Another part of the tool support presented in the following section interacts with the programmer in order to collect this information and perform the replacement.

The next piece of code defines the target template that implements the singleton design pattern.

```
PackageDeclaration:#PD_FACADE;
ImportDeclarationSet:#IDS;
public class #FACADE #[ extends #SC_FACADE ]#
                     #[implements NameList:#IFS_FACADE ]#  {
  private static #< #FACADE ># singleton;
  FieldDeclarationSet:#ATTRS_FACADE;
  InitializerSet:#IS_FACADE;
  forall #cd in #CDS_FACADE {
    #< #cd.getModifiers().removeAllModifiers() >#;
    #< #cd.getModifiers().
              addModifier(java.lang.reflect.Modifier.PRIVATE) >#;
  }
  ConstructorDeclarationSet:#CDS_FACADE;
```

the template adds the `singleton` static field, makes facade constructs `private` using a iterative declaration (`forall`), and adds the `getInstance` method

```
  public #< #FACADE ># getInstance() {
    if (singleton == null) {
      singleton = new #< #FACADE >#();
    }
    return singleton;
  }
  MethodDeclarationSet:#MTDS_FACADE;
}
```

Note that this template is quite general and can be used to implement the singleton design pattern for any class.

Chapter 3 presents the distribution framework. The `ServerSide` aspect (see Section 3.4.1) provides common behavior on exporting and binding the object to be remotely accessed instead of executing its main method. Therefore, the concrete server-side aspect should specialize the abstract pointcut of the `AbstractServerSideAspect` aspect to identify the facade's `main` method execution, should define how to initialize the facade class, and should provide a name to bind the facade object to.

The following target template generates the server-side aspect that extends the framework abstract aspect.

```
package ##TARGET_PACKAGE##;
import cin.aspects.framework.distribution.ServerSide;
import java.rmi.Remote;
public aspect #< #FACADE.addSuffix("ServerSideAspect") >#
    extends ServerSide {
  public static final String SYSTEM_NAME     = "##SYSTEM_NAME##";
  public static final String RMI_SERVER_NAME = "##RMI_SERVER_NAME##";
```

Note that this template also has constructs between a pair of `##`'s that should be replaced by a system-specific value before performing the code generation. Next, the template generates two `declare parents` constructs to make the facade class implement the generated remote interface and to make classes whose objects are sent over the communication channel implement the `Serializable` interface.

```
  declare parents: #< #FACADE >#
                   implements #< #FACADE.addPrefix("IRemote") >#;
  declare parents: ##SERIALIZABLE_TYPE_LIST##
                   implements java.io.Serializable;
  Remote initFacadeInstance() {
    return #< #FACADE >#.getInstance();
  }
```

The `initFacadeInstance` method defines how to initialize a facade object and is used by the super-aspect. In the same way, this template defines how to obtain the name the facade object should be bound to, and the pointcut that identifies facade's `main` method execution.

```
  String getSystemName() {
    return SYSTEM_NAME;
  }
  pointcut facadeMainExecution(String[] args):
    execution(static void #< #FACADE >#.main(String[])) &&
    args(args);
  public static void #< #FACADE >#.main(String[] args) {
    // just to enable the super-aspect
    // if there is not a main method yet
  }
}
```

Note that these templates are general enough to be used in other situations, where other classes should have their objects distributed using RMI.

Similar to the server-side, there is a client-side target template that generates an aspect using the distribution aspect framework like the aspect presented in Chapter 3.

The set of AJaTS transformations files used to generate distribution, data management, and concurrency control aspects, auxiliary aspects and classes, and types of the specific software architecture are presented in Appendix B.

6.2.2 Interacting with the programmer

As previously mentioned, some transformations need to interact with the programmer. In fact, most of them need some information from the programmer to identify the target of the transformation. For example, the distribution transformations require the programmer to identify the facade class. Likewise, the base transformations for generating nonpersistent data collection, business collections, business-data interface, and facade class require the programmer to identify the basic classes.

In order to provide such interaction in a user friendly fashion we developed a plug-in for the Eclipse platform [25, 33] in order to apply the transformations and collect the necessary information from the programmer. In fact, the plug-in guides the transformations, being an additional support for programmers.

Figure 6.2 shows a snapshot of the plug-in running on Eclipse and requesting the identification of the facade class to generate the distribution aspects. Note that the plug-in contributes with a new item in the menu bar, where there are options to generate the aspects. The current plug-in version generates distribution aspects.

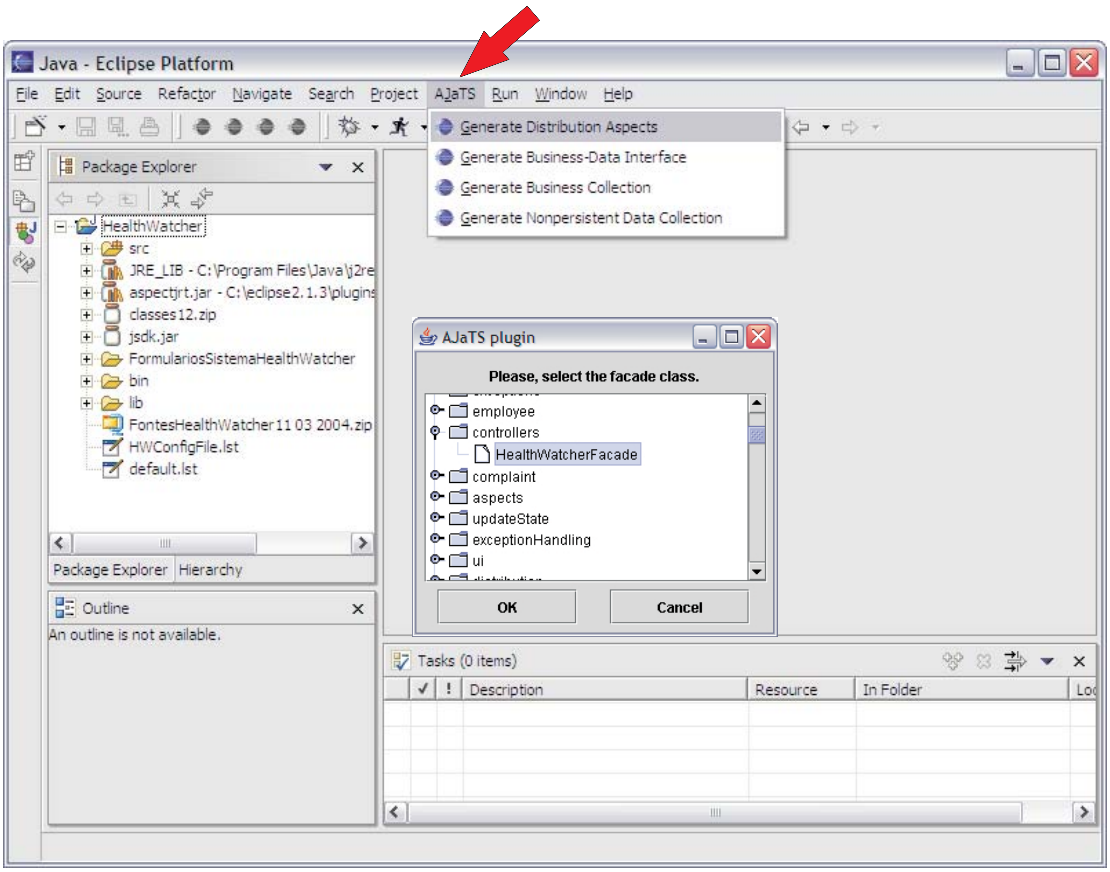

Figure 6.2: Snapshot of the plug-in execution.

After collecting the requested information, the plug-in uses a set of source and target transformation files and the AJaTS engine to generate the aspects and auxiliary types from the type identified by the user. Regarding the distribution aspects, the types generated by the tool are two aspects that specializes the client and the server-side aspects and the remote interface.

As an Eclipse plug-in, the tool can be easily installed in the Eclipse platform. In fact, to install the tool is only necessary to extract a zip file into Eclipse's plug-in directory. The zip file is available at *http://www.cin.ufpe.br/∼jats/AJaTS*

6.3 Conclusion

This chapter presented a tool to generate code supporting the aspect-oriented implementation method. The tool generates either Java and AspectJ types and is composed of an Eclipse plug-in, which is actually the tool's front-end, and of a transformation engine, called AJaTS. The current version of AJaTS can generate Java and AspectJ.

The tool uses transformation files that guide the code generation. In fact, the back-end of the tool is not tailored to the implementation method, and therefore can be used for transforming and generating any kind of Java and AspectJ code. We can easily change the plug-in to consider user-defined transformation files, instead of only the pre-defined transformations.

This tool is especially useful when adopting the progressive approach. When executing the functional iteration, persistence, distribution, and concurrency control code are not considered. At this moment, the tool can generate nonpersistent data collections for testing the functional prototype before implementing persistent data collections.

In addition, the tool should be extended in order to perform the non-trivial task of analyzing the code to evaluate some preconditions, allowing refactorings definition. In fact, this is been carried out elsewhere [14, 15].

Chapter 7

Related work

This chapter discusses related work related to aspect-oriented software development and to implementation methods.

This chapter presents works related to aspect-oriented software development. In particular, some of these works are also related to the implementation of persistence, distribution, or concurrency control concerns, and to the implementation method.

7.1 Evaluating distribution and persistence concerns implementation using AspectJ

As mentioned in Chapter 3, we restructured an object-oriented version of the Health Watcher software to an aspect-oriented one. Kulesza et al. [44] evaluate the differences between the object-oriented and the aspect-oriented versions of the Health Watcher software using the following set of metrics:

- Separation of concerns
  - Concern diffusion over components (CDC) — number of components (classes and aspects) to implement a concern;
  - Concern diffusion over operations (CDO) — number of methods and advices of the components that implement a concern;
  - Concern diffusion over LOC (CDLOC) — number of transition points to each concern through the lines of code. This metric captures how tangling and spreading is a concern in the software.
- Coupling
  - Coupling between components (CBC) — how coupled is a component with respect to other components;
  - Depth of inheritance tree (DIT) — classes and aspects hierarchy depth.
- Cohesion
  - Lack of cohesion in operations (LCO) — measures the cohesion of the classes and aspects through how their methods and advices manipulate their fields.
- Size
  - Vocabulary Size (VS) — number of classes and aspects;
  - Lines of code (LOC) — classes and aspects lines of code;
  - Number of attributes (NOA) — classes and aspects fields;
  - Weighted operations per component (WOC) — complexity measure for operations, based on the parameters number of methods or advices.

They give evidence of several advantages of the aspect-oriented (AO) solution over the object-oriented (OO). The AO version decreased the number of lines of code by 9%. When considering all the components, the OO Health Watcher is 12% less cohesive than the AO, and when considering each component, the mean difference of cohesion was 51%, in favor of the AO software. This difference is, in part, related to the tangling code between business and distribution and persistence concerns. There is some equality between the two versions with respect to coupling, which shows that the OO software was well designed. In fact, the mean coupling per component was 8% smaller in the AO version, because there are more components (aspects). The hierarchy depth was also similar between the OO and the AO software.

As expected, the separation of concerns metrics show a great advantage of the AO version. The number of components (classes and aspects) to implement a concern (CDC) is around 77% smaller for distribution, 51% for persistence, and 31% for concurrency control. Regarding the number of methods and advices of the components that implement a concern (CDO), the AO version had numbers 47% smaller for distribution, 41% for persistence, and 17% for concurrency control. Finally, the numbers of transition points to each concern through the lines of code (CDLOC) are more than 77%, 95%, 98% smaller for distribution, persistence and concurrency control. These numbers show how spread and tangled are those concerns in the OO version, and how effective is the modularizations achieved by the AO approach.

The size metrics show that the OO version is around 4% less complex (WOC) than the AO one, however, when considering the components mean, the AO version is 4% less complex. On the other hand, the AO software has 9% more components (VS) than the OO software. The number of fields (NOA) of the two versions is equal.

The general results conclude that there is a relative advantage of the AO Health Watcher with respect to the OO Health Watcher. The AO advantage is mainly related to the separation of concerns and lines of code metrics. More studies are being performed in order to identify discrepancies in the individual components results. More details about this study can be found elsewhere [44].

7.2 Use-case Driven Development and Aspect-Oriented Software Development

Jacobson [38, 36, 37] considers Aspect-Oriented Programming (AOP) the missing link to allow effective modeling with use cases. When working with use cases with the current UML [10], neither analysis and design artifacts, such as collaboration, components and class diagrams, nor implementation languages, such as Java and C#, support the use of use-case extensions.

Jacobson considers that UML use-case extensions are equivalent to AOP aspects. Actually, some extensions do not change the base use-case, while others demand invasive changes in the base use case. Aspects are used in order to implement extensions that otherwise would be invasive. For example, consider an abstract use case, called *ManageAccounts*, that defines the operations to manage account objects in a storage medium. A concrete use case would extend the abstract to manage account objects in a relational or object-oriented database, file system, or another medium. This is an example of a *generalization* relationship, and can be used to model our data management concern. Another kind of extension relationship is *extend*, where a use case add behavior (extends) to a base use case. In those extension relationships, it is necessary to define extension points in the base use case. Those extension points can be mapped to join points, and the added behavior to advices.

Jacobson's work is primarily concerned in refining the way systems are modeled, towards an Aspect-Oriented modeling, also establishing a mapping between model and implementation. On the other hand, our work is primarily concerned with implementation, also providing an alternative implementation approach (progressive approach) to early identify requirement changes. Despite being an implementation method, our work also defines how the method can be used in a development process, by defining how composing it with use-case driven development and RUP. In fact, Jacobson's work is complementary to ours.

7.3 Persistence as an Aspect

Another work [75] 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, and to allow early validation of functional requirements, which is possible by validating a nonpersistent, monolithic, and single-user version of the system. After being validated, the prototype evolves to the persistent, distributed, and multi-user version.

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 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 that 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.

7.4 Concurrency and Transactions

A related work is another AspectJ implementation of transactions, which is independently developed in the context of the OPTIMA framework for controlling concurrency and failures with transactions [43]. 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 restructuring experience 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, but they do not explore the variations that we present, such as separating read and write transactions. Those variations can actually avoid the performance problems they mentioned. They also faced the same problem we have 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.

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 than for the transaction aspects. It seems that our AspectJ implementation of transactions can usually have significant advantages over pure Java implementations. That is certainly the case for systems such as the one used in our restructuring experience and experimental study presented in Chapters 3 and 5.

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.

7.5 Concurrent Object Programming

This related work deals with separation of concurrency concerns using design patterns, pattern languages, and object-oriented framework [18]. The work covers several issues related to concurrent programming, including how to generate, control, and guarantee interaction between concurrent objects. On the other hand, our thesis scope is limited to how to control concurrency in a specific software architecture. In fact, our distribution aspects generate the concurrency that is controlled by the concurrency aspects.

The work states the need for an incremental approach in order to first implement and test functional requirements, before implementing concurrency, similar to our progressive approach. However, as object-oriented programming is used instead of aspect-oriented programming, it has some problems when adopting an incremental approach. For example, there might be conflicts between using an incremental approach and the software reusability. Instead of adding concurrency between two objects by only changing their implementation, it might be necessary to change the interfaces between them, decreasing their reusability. If the work used aspect-oriented programming, those interface changes could be made by aspects, guaranteeing business objects reusability.

Another problem mentioned by the work is the inheritance anomaly when using synchronization. When classes include synchronization code, it might not be trivial reusing through inheritance. The anomaly happens when new subclasses, with new methods or new method implementations demand changing synchronizations constraints in the superclass methods. Our concurrency control concerns do not have such problem, since there is no specific concurrency control in classes that are specialized in the specific software architecture used. The classes that use to have subclasses in the architecture are the basic classes. The concurrency control applied to them is based in its fields' type, if a single object can be concurrently accessed, or if it is possible two copies of a same object be concurrently updated in the system. In both cases, the inheritance anomaly does not materialize. When analyzing the fields of a basic object, each class defines their own fields, and the concurrency control should be normally applied for the super and subclass separately. In the case of two copies of a same object being concurrently updated, new properties or behavior added by a subclass might request to control the concurrent update of this object. However, this control is not made in the superclass or in the subclass, the control is applied in other class of the architecture, the data collection class. In fact, this control demands adding a timestamp field in the class to be controlled, which it is made by an aspect. By using aspect-oriented programming, the concurrency control is transparent for the business objects not compromising their reuse.

Similar to our work, there are several design patterns and a framework to implement the concurrency concerns. Additionally, the work also has a pattern language to describe how to compose the concurrency concerns (classes) with the sequential software. In our approach, this composition of concerns is made by the AspectJ weaver that uses information in the aspects that describe how they should be composed with the software.

This related work provides general concurrency patterns and framework that can be applied in several kinds of software. Whereas our aims in defining patterns and framework specific to a software architecture. Despite being specific, this software architecture can be used to implement several kinds of software (see Chapter 3) and allows a more precise definition of the patterns, framework, and guidelines on how to use them. Besides that, is also allows automatic generation of some aspects and classes necessary to implement the software architecture and to use the aspect framework, which also guides the concerns implementation.

7.6 D: A language framework for distributed programming

D [50] is a domain specific language framework that consists of three languages: Jcore, an object-oriented language to express basic functionality of the software (a subset of Java), Cool, a language to express coordination of threads, and Ridl, a language to express remote access strategies. The framework uses an aspect-oriented approach in order to achieve separation of the distribution concerns from the basic software. Furthermore, Cool and Ridl are aspect languages. A weaver is responsible for combining programs written in these three different languages yielding the executable distributed software.

The work uses RMI and Java concurrency control primitives to exemplify how they tend to cut across the implementation of business objects. On the other hand, our work presents an aspect-oriented approach that uses a general-purpose language, AspectJ, to separate those crosscutting concerns. Our approach is not tailored to the use of RMI, which allows changing the distribution protocol. The same happens with the synchronization policies. Moreover, our approach allows a fine grain implementation, for example, instead of using the RMI API the programmer might need to use the Socket API, for performance purposes. Although we have an aspect framework, other aspects can be written to implement other distribution protocols, design patterns [30], and concurrency control polices.

In addition, JCore removes from Java the synchronization mechanisms (the `synchronized` modifier, and `wait`, `notify`, and `notifyAll` methods), interfaces, and method overloading, which makes Jcore a quite restrictive language. Interfaces and method overloading are basic object-oriented constructs and `wait`, `notify`, and `notifyAll` methods might be necessary in order to implement business rules, and therefore, should not be separated from the basic software.

The use of Cool to control concurrency and Ridl to insert distribution introduces two new languages the programmer must know. Our approach also requires the programmer to know AspectJ, in addition to Java, however, this knowledge is used to implement several concerns, not only concurrency control and distribution. In addition, our approach uses the Java synchronization mechanisms, which were tested for innumerous programmers and researchers, instead of implementing our own synchronization mechanisms. Complementary to our approach, a concurrency control implementation method [80, 84] guides precisely how to control concurrency in a software using the specific architecture of Health Watcher, which is not provided by the D language framework. In fact, the D framework is a general framework, whereas ours approach is architecture specific. However, as previously mentioned, this specificity allows precise guidelines and automatic code generation, which is not supported by D.

7.7 EJB

Another related work is The Enterprise JavaBeans (EJB) architecture [60, 93]. 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 improvements, which happens in our approach, where the developers have to write their own code to implement transactions, database connectivity, distribution, and multi-user safety, dealing with different APIs.

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

Our implementation method does not aim to provide implementation transparency, but to improve software modularity. This modularity allows the use of a progressive implementation approach to early identify functional errors and does not mix these errors with persistence, distribution and concurrency control 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, distribution, and concurrency control 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 other benefits of AOP, is using code generation tools that would generate aspects, similar to EJB that generates classes.

7.8 Other related works

The implementation of distribution and persistence concerns in pure object-oriented systems is explored elsewhere, leading to specific design patterns [2, 53]. 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 a level of separation of concerns that is possible to achieve with aspect-oriented programming (AOP); for instance, using these design patterns, the distribution and persistence exception handling are tangled with user interface and business code. There is also spread 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 is 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 [106]. That 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 [65] middleware, by factoring out aspects that were identified in CORBA. The identified aspects are implemented in AspectJ to increase the middleware configurability, since the aspects can be chosen at compile-time. That work is another example where AOP and AspectJ are effectively used to modularized crosscutting concerns. However, that 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 [8]. The motivation is 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 ours in the restructuring experience (see Chapter 3), one of the issues is pointcut fragility, which is the pointcut dependence on the system code. This demands refactoring 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, communication, and concurrency control) in contrast to their concerns (tracing and logging, event reporting, error handling, and performance monitoring).

Regarding distribution and aspects, another work [95] proposes a tool for supporting aspect-oriented distributed programming. They have the same goal of implementing distribution without changing the core system code. However, that 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 and does not worry in define where the objects are located, but uses an API to implement remote methods call from the user interface to the system facade [26].

The need for Quality of Service (QoS) aspects in distributed programs are discussed elsewhere [6]. That 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 restructuring experience (see Chapter 3) is 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.

The JST [79] language introduces an object synchronization aspect for the Java language. The language allows programmers to define synchronization classes for Java. JST is based on a state/transition language, where the basic idea is to define in the synchronization classes all possible states for a concurrent class and what methods can be executed in these states. The language has its own aspect weaver, implemented in C++, which produces an OpenJava [96] meta-class for each synchronization aspect. OpenJava is a compile-time reflective extension of Java. JST is more general than our approach, which defines concurrency control aspects tailored to a specific software architecture. However, by being specific, our aspects definition can be more precise and might be more efficient [84] than a general approach. In addition, the guidelines derived from our aspect definition supports programmers to control concurrency, which does not happen with JST, where the programmers should figure out by themselves what controls to apply in a system.

"Concurrent Programming in Java" [49] proposes models for implementing Java concurrent programs, design patterns to guarantee a safety execution in concurrent environments, and some rules to insert and to remove method synchronization. The approach in that work insinuates that concurrency control must be applied during the implementation of the system functional requirements. This increases the implementation complexity because the programmer has to worry about the concurrency control and the implementation of the other requirements at the same time and in the same place (tangled), which decreases the software maintainability. On the other hand, our approach allows implementing concurrency control separately (untangled), also separating the reasoning about concurrency control. Our approach also allows using a progressive approach [11, 86] where concurrency control is delayed until functional requirements validation, in order to reduce the impact caused by requirement changes during development. Another differential of our work is that is based on a specific software architecture, which facilitates the definition of precise guidelines, also allowing a high automatization level.

Chapter 8

Conclusions

This chapter presents conclusions made about this thesis work and suggests future work.

This work defines an aspect-oriented implementation method that modularizes data management, distribution and concurrency control concerns. By using AspectJ, an aspect-oriented extension of Java, these concerns implementation are separated from business and user interface source code, which are written using Java. The aspects implementation is presented through activities 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, concurrency control, and data management concerns are physically separated from each other and from the business and user interface concerns, resulting in a software more modular than the object-oriented version. 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.

A contribution in deriving such activities is to validate the use of AspectJ for implementing several data management, distribution, concurrency control, and exception handling 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 is reported elsewhere [22]. The activities 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. Examples of such alternatives are defined elsewhere [51]. 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 data management 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 show that the distribution aspects affect the persistence aspects by breaking some of then, not allowing them to work. Therefore, additional aspects are implemented to solve those problems when using persistence in a distributed environment. This is an example of crosscutting concerns that are physically, but not semantically separated.

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 [43], 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 [102]. In fact, it is important to clarify the use of the word "abstraction" in the aspect-oriented programming context. This has noting to do with transparency, in the sense that any concern is semantically tangled with business or other concerns; otherwise, they are not crosscutting concerns. Aspect-Oriented Programming, in the beginning, and now Aspect-Oriented Software Development have never tried to mean that, as some might believe.

Some of the aspects implemented in our restructuring experience 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 (see Chapter 3). In fact, some of them can be easily used for other architectures, mainly concurrency control and distribution aspects (see Chapter 3).

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 [85], we document such an aspect pattern to implement distribution aspects in an object-oriented application. These pattern structures can be encoded in code generation tools and automatically generated for different applications, increasing productivity, as we made with AJaTS, a tool support for the aspect-oriented implementation method (see Chapter 6).

In addition, the aspect-oriented implementation method proposes an alternative for implementing software, the progressive implementation approach. The approach aims in validating functional requirements before implementing persistence, distribution and concurrency control, towards to increase productivity. Moreover, an experimental study was performed in order to characterize this alternative approach identifying when it is useful to adopt it (see Chapter 5). Although the study suggests important advantages when using the progressive implementation approach, additional experimental studies should be performed to allows generalizing the results to others development teams.

Despite defining an implementation method, this work is also committed with the whole development process. Therefore, we identify modifications that should be performed to management, requirements, analysis and design, and test activities in order to support the aspect-oriented implementation method and the progressive implementation approach. This is actually done in Chapter 4, where he we discussed how the implementation method can be composed with RUP and Use-Case Driven Development. Chapter 3 used UML for modeling aspects dynamics.

8.1 Future Work

Most examples on aspect-oriented programming and aspect-oriented software development are non-functional requirements, but aspects are not limited to non-functional requirements. Jacobson's work [38, 36, 37] discussed on the related work chapter, provides a good starting point for identifying functional crosscutting concerns and write them as aspects. Any use-case extension might be an aspect, if the extension demands invasive changes in the base (extended) use case. This probably should be identified by analysis and design models, and the current Health Watcher models have not such situation. It might be necessary to refine Health Watcher analysis or look at a more complex system.

More sophisticated distribution aspects can be written, for instance to guarantee fault tolerance, caching, and automatic object deployment for load balancing. This would also make necessary more sophisticated concurrency control, since an operation might start in one server and finish in another. Imagine implementing transaction control and other concurrency controls needed for such operations. The distribution aspects can also be extended to consider QoS (quality of service) [6].

The tool support for the aspect-oriented implementation method generates classes and aspects. More generally, AJaTS is a transformation tool, and therefore, can be used to implement object-oriented and aspect-oriented refactorings. However, more work should be done in order to define behavior-preserving transformations, maybe extending the tool to provide some static analysis. In fact, a masters work in progress [14] is investigating refactorings for aspect-oriented programming and will use and extend AJaTS to perform those refactorings.

More experimental studies should be performed to better evaluate the progressive implementation approach. In fact, experimental studies should be designed to evaluate several results of this work. For example, studies to evaluate the aspect framework and the tool support. Another interesting study can evaluate different instances of the modified version of RUP that complies with the implementation method, aspect-oriented development, and progressive approach. A third study can compare the aspect-oriented implementation method with the object-oriented one [52]. Another study that can be performed is to evaluate the use of the progressive implementation approach with other development techniques.

An important future work is to investigate alternatives to some open issues and workarounds used in this work. For example, some defined aspects use reflection to increase the aspect reuse, however decreasing code legibility and reliability. On the other hand, alternatives, such as aspect parameterization or code generation tools can provide a much more reliable and elegant solution. Another future work is to investigate alternative to define remote pointcuts, which is one of the worst impacts of the distribution aspects, which actually had break some data management aspects.

Since the implementation method uses a specific software architecture, another future work is to generalize the software architecture, also executing case studies and experimental studies to evaluate such proposals. In addition to the architecture generalization, the implementation method and the activities impacted by the method should also be generalized to be applied in a wider range of software.

Appendix A

Experiment Questionare

Answer the following questionnaire in order to collect information about your profiles and experience with software development.

1. Name: ____

2. Fill the following form with the appropriate information about your experience with the following items. **ATENTION**: You should fill the form using the letters **A** and **I** to define your experience in academia (A) and in the industry (I) or mark the OITC square if your experience is only in this course. For example, if you have 5 months of experience in academia and 3 years in industry in one technology you should fill the column "< 6m" with A and column "2-4y" with I.

```
OITC  - only in this course
< 6m  - less than 6 months
6m-2y - Between 6 months and 2 years
2y-4y - Between 2 and 4 years
4y-6y - Between 4 and 6 years
> 6y  - more than 6 years
```

Item	OITC	< 6m	6m-2y	2-4y	4-6y	> 6y
Java						
Databases						
JDBC						
Distributed systems						
RMI (Remote Method Invocation)						
Java Servlets						
Object-oriented design						
AspectJ						
Use Cases						
Software development projects						

3. What is your current situation?

- M.Sc. student.
- Ph.D. student.
- Other. Please, specify:

Appendix B

AJaTS templates

This appendix presents transformation files used to generate code with AJaTS in order to support the aspect-oriented implementation method. The templates use a naming convention adding _lhs and _rhs suffixes standing for left hand-side (source template) and right hand-side (target template), and named with the extension .ajast.

B.1 Software architecture

The following transformation files generate types of the spectific software architecture used by the implementation method.

B.1.1 Basic class source template

```
//BasicClass_lhs.ajats
#[PackageDeclaration:#PD_BASIC;]#
ImportDeclarationSet:#IDS_BASIC;
ModifierList:#M_BASIC class #BASIC #[ extends #SC_BASIC ]#
                                #[implements NameList:#IFS_BASIC ]#  {
    FieldDeclarationSet:#ATTRS_BASIC;
    InitializerSet:#IS_BASIC;
    ConstructorDeclarationSet:#CONS_BASIC;
    MethodDeclarationSet:#MTDS_BASIC;
}
```

B.1.2 Business-data interface target template

```
//BusinessDataInterface_rhs.ajats
package #< #BASIC.toVariableName() >#
ImportDeclarationSet:#IDS_BASIC;
import java.rmi.RemoteException;
ModifierList:#M_BASIC interface #< #BASIC.addPrefix("I")
                                        .addSuffix("Repository") ># {
    forall #md in #SELECTED_MDS {
       MethodDeclaration:#< #md.toInterfaceMethodDeclaration() >#;
    }
}
```

B.1.3 Business collection target template

```
//BusinessCollection_rhs.ajats
package #< #BASIC.toVariableName() >#
public class #< #BASIC.addSuffix("Record", false) ># {
    private #< #BASIC.addPrefix("I").addSuffix("Repository") >#
                #< #BASIC.addSuffix("s", false).toVariableName() >#;
    public #< #BASIC.addSuffix("Record", false) >#() {
        this.#< #BASIC.addSuffix("s", false).toVariableName() ># =
                new #< #BASIC.addPrefix("I")
                    .addSuffix("Repository") >#();
    }
    public  void cadastrar( #BASIC #< #BASIC.toVariableName() >#) {
        if(#< #BASIC.addSuffix("s", false).toVariableName() >#
                .has(#< #BASIC.toVariableName() >#.getId())) {
            throw new #< #BASIC.addSuffix("AlreadyRegisteredException",
                                          false) >#();
        } else {
            #< #BASIC.addSuffix("s", false).toVariableName() >#
                    .insert(#< #BASIC.toVariableName() >#);
        }
    }
    public  void update( #BASIC #< #BASIC.toVariableName() >#) {
        #< #BASIC.addSuffix("s", false).toVariableName() >#
                .update(#< #BASIC.toVariableName() >#);
    }
    public  void remove( String id) {
        #< #BASIC.addSuffix("s", false).toVariableName() >#.remove(id);
    }
    public  #BASIC search( String id) {
        return #< #BASIC.addSuffix("s", false).toVariableName() >#
                .search(id);
    }
    public  boolean has( String id) {
        return #< #BASIC.addSuffix("s", false).toVariableName() >#
                    .has(id);
    }
}
```

B.1.4 Facade target template

```
//Facade_rhs.ajats
#[PackageDeclaration:#PD_FACADE;]#
ImportDeclarationSet:#IDS_OPTIONAL;
ImportDeclarationSet:#IDS_FACADE;
public class #FACADE #[ extends #SC_FACADE ]#
                     #[implements NameList:#IFS_FACADE ]#  {
    private #< #BASIC.addPrefix("Cadastro").addSuffix("s", false) >#
                   #< #BASIC.addSuffix("s", false).toVariableName() >#;

    FieldDeclarationSet:#ATTRS_FACADE;
    InitializerSet:#IS_FACADE;
    public #FACADE() {
          Block:#CONST_BLOCK;
          #< #BASIC.addSuffix("s", false).toVariableName() ># =
                   new #< #BASIC.addPrefix("Cadastro").addSuffix("s", false) >#();
    }
    ConstructorDeclarationSet:#CDS_FACADE;
    public  void #< #BASIC.addPrefix("cadastrar") >#(
                #BASIC #< #BASIC.toVariableName() >#) {
        #< #BASIC.addSuffix("s", false).toVariableName() >#
                .cadastrar(#< #BASIC.toVariableName() >#);
    }
    public  #BASIC #< #BASIC.addPrefix("procurar") >#( String id) {
        return #< #BASIC.addSuffix("s", false).toVariableName() >#
                       .procurar(id);
    }
    public  void #< #BASIC.addPrefix("remove") >#( String id) {
        #< #BASIC.addSuffix("s", false).toVariableName() >#.remove(id);
    }
    public  void #< #BASIC.addPrefix("update") >#(
            #BASIC #< #BASIC.toVariableName() >#) {
        #< #BASIC.addSuffix("s", false).toVariableName() >#
                .update(#< #BASIC.toVariableName() >#);
    }
    MethodDeclarationSet:#MTDS_FACADE;
}
```

B.2 Data management templates

The following target templates use the same basic class source template used by the software architecture templates.

B.2.1 Array data collection target template

```
//RepositoryArray_rhs.ajats
#[PackageDeclaration:#PD_BASIC;]#
public class #< #BASIC.addSuffix("RepositoryArray", false) ># implements
        #< #BASIC.addPrefix("I").addSuffix("Repository") ># {
    private #BASIC[] #< #BASIC.addSuffix("s", false).toVariableName() >#;
    private int index;
    public #< #BASIC.addSuffix("RepositoryArray", false) >#() {
        #< #BASIC.addSuffix("s", false).toVariableName() ># =
            new #BASIC[100];
        index = 0;
    }
    public  void insert( #BASIC #< #BASIC.toVariableName() >#) {
        #< #BASIC.addSuffix("s", false).toVariableName() >#[index] =
                #< #BASIC.toVariableName() >#;
        index = index + 1;
    }
    public  void update( #BASIC #< #BASIC.toVariableName() >#) throws
            #< #BASIC.addSuffix("NotFoundException", false) ># {
        int i = getIndex(#< #BASIC.toVariableName() >#.getId());
        if(i == index) {
            throw new #< #BASIC.addSuffix("NotFoundException", false) >#();
        } else {
            #< #BASIC.addSuffix("s", false).toVariableName() >#[i] =
                    #< #BASIC.toVariableName() >#;
        }
    }
    public void remove( String id) throws #< #BASIC.addSuffix(
            "NotFoundException", false) ># {
        int i = getIndex(id);
        if(i == index) {
            throw new #< #BASIC.addSuffix("NotFoundException", false) >#();
        } else {
            index = index - 1;
            #< #BASIC.addSuffix("s", false).toVariableName() >#[i] =
                #< #BASIC.addSuffix("s", false).toVariableName() >#[index];
        }
    }
```

```
    public #BASIC search( String id) throws #< #BASIC
            .addSuffix("NotFoundException", false) >#  {
        #BASIC response;
        int i = getIndex(id);
        if(i == index) {
            throw new #< #BASIC.addSuffix("NotFoundException", false) >#();
        } else {
            response = #< #BASIC.addSuffix("s", false)
                .toVariableName() >#[i];
        }
        return response;
    }
    public  boolean has( String id) {
        boolean response;
        int i = getIndex(id);
        if(i == index) {
            response = false;
        } else {
            response = true;
        }
        return response;
    }
    private int getIndex( String id) {
        boolean found = false;
        int i = 0;
        while((!found) && (i < index)) {
            if(#< #BASIC.addSuffix("s", false).toVariableName() >#[i]
                    .getId().equals(id)) {
                found = true;
            } else {
                i = i + 1;
            }
        }
        return i;
    }
}
```

B.2.2 List data collection target template

```
//RepositoryList_rhs.ajats
#[PackageDeclaration:#PD_BASIC;]#
public class #< #BASIC.addSuffix("RepositoryList", false) ># implements
        #< #BASIC.addPrefix("I").addSuffix("Repository") ># {
    private #BASIC #< #BASIC.toVariableName() >#;
    private #< #BASIC.addSuffix("RepositoryList", false) ># next;
    public #< #BASIC.addSuffix("RepositoryList", false) >#() {
        #< #BASIC.toVariableName() ># = null;
        next = null;
    }
    public  void insert( #BASIC #< #BASIC.toVariableName() >#) {
        if(this.#< #BASIC.toVariableName() ># != null) {
            next.insert(#< #BASIC.toVariableName() >#);
        } else {
            this.#< #BASIC.toVariableName() ># =
                #< #BASIC.toVariableName() >#;
            this.next =
                new #< #BASIC.addSuffix("RepositoryList", false) >#();
        }
    }
    public void update( #BASIC #< #BASIC.toVariableName() >#) throws
            #< #BASIC.addSuffix("NotFoundException", false) ># {
        if(this.#< #BASIC.toVariableName() ># != null) {
            if(this.#< #BASIC.toVariableName() >#.getIdentificador()
                    .equals(#< #BASIC.toVariableName() >#
                    .getIdentificador())) {
                this.#< #BASIC.toVariableName() ># =
                    #< #BASIC.toVariableName() >#;
            } else {
                next.update(#< #BASIC.toVariableName() >#);
            }
        } else {
            throw new #< #BASIC.addSuffix("NotFoundException", false) >#();
        }
    }
    public void remove( String id) throws
            #< #BASIC.addSuffix("NotFoundException", false) ># {
        if(this.#< #BASIC.toVariableName() ># != null) {
            if(this.#< #BASIC.toVariableName() >#.getIdentificador()
                    .equals(id)) {
                this.#< #BASIC.toVariableName() ># =
                    next.#< #BASIC.toVariableName() >#;
                this.next = next.next;
            } else {
                next.remove(id);
```

```
            }
        } else {
            throw new #< #BASIC.addSuffix("NotFoundException", false) >#();
        }
    }
    public #BASIC search( String id)
            throws #< #BASIC.addSuffix("NotFoundException", false) ># {
        #BASIC response;
        if(this.#< #BASIC.toVariableName() ># != null) {
            if(this.#< #BASIC.toVariableName() >#.getIdentificador()
                    .equals(id)) {
                response = this.#< #BASIC.toVariableName() >#;
            } else {
                response = next.search(id);
            }
        } else {
            throw new #< #BASIC.addSuffix("NotFoundException", false) >#();
        }
        return response;
    }
    public boolean has( String id) {
        boolean response;
        if(this.#< #BASIC.toVariableName() ># != null) {
            if(this.#< #BASIC.toVariableName() >#.getIdentificador()
                    .equals(id)) {
                response = true;
            } else {
                response = next.has(id);
            }
        } else {
            response = false;
        }
        return response;
    }
}
```

B.2.3 Relational database data collection target template

```
//RepositoryBDR_rhs.ajats
#[PackageDeclaration:#PD_BASIC;]#
import java.sql.*;
import java.util.Vector;
import java.util.Collection;
import #EXCEPTION_PCKG.PersistenceSoftException;
import #EXCEPTION_PCKG.#< #BASIC.addSuffix("NotFoundException", false) ># ;
import #EXCEPTION_PCKG.ExceptionCode;
import #POOL_PKG.PersistenceMechanismRDBMS;
import #POOL_PKG.TransacaoException;
import #POOL_PKG.PersistenceMechanismException;
ImportDeclarationSet:#IDS_BASIC;
public class #< #BASIC.addSuffix("RepositoryRDBMS", false) ># implements
        #< #BASIC.addPrefix("I").addSuffix("Repository") ># {
    private PersistenceMechanismRDBMS pm;
    public #< #BASIC.addSuffix("RepositoryRDBMS", false) >#() {
        try {
            pm = PersistenceMechanismRDBMS.getInstance();
        } catch(TransacaoException ex){
            throw new PersistenceSoftException(ex);
        }
    }
    public void insert( #BASIC #< #BASIC.toVariableName() >#) {
        String sql = #SQL_I;
        Statement stmt = null;
        try {
            stmt = (Statement)pm.getCommunicationChannel();
            stmt.execute(sql);
        } catch(SQLException e) {
            throw new PersistenceSoftException(e);
        } catch(PersistenceMechanismException e) {
            throw new PersistenceSoftException(e);
        }
        finally {
            try {
                stmt.close();
                pm.releaseCommunicationChannel();
            } catch(SQLException e) {
                throw new PersistenceSoftException(e);
            } catch(PersistenceMechanismException e) {
                throw new PersistenceSoftException(e);
            }
        }
    }
```

```
public void update( #BASIC #< #BASIC.toVariableName() >#) throws
        #< #BASIC.addSuffix("NotFoundException", false) ># {
    Statement stmt = null;
    String sql = #SQL_A;
    try {
        stmt = (Statement)pm.getCommunicationChannel();
        int resultadoAtualizacao = stmt.executeUpdate(sql);
        if(resultadoAtualizacao == 0) {
            throw new #< #BASIC.addSuffix("NotFoundException",
                                          false) ># ();
        }
    } catch(SQLException e) {
        throw new PersistenceSoftException(e);
    } catch(PersistenceMechanismException e) {
        throw new PersistenceSoftException(e);
    }
    finally {
        try {
            stmt.close();
            pm.releaseCommunicationChannel();
        } catch(SQLException e) {
            throw new PersistenceSoftException(e);
        } catch(PersistenceMechanismException e) {
            throw new PersistenceSoftException(e);
        }
    }
}
public void remove( #BASIC #< #BASIC.toVariableName() >#) throws
        #< #BASIC.addSuffix("NotFoundException", false) ># {
    Statement stmt = null;
    String sql = #SQL_R;
    try {
        stmt = (Statement)pm.getCommunicationChannel();
        int resultadoAtualizacao = stmt.executeUpdate(sql);
        if(resultadoAtualizacao == 0) {
            throw new #< #BASIC.addSuffix("NotFoundException",
                                          false) ># ();
        }
    } catch(SQLException e) {
        throw new PersistenceSoftException(e);
    } catch(PersistenceMechanismException e) {
        throw new PersistenceSoftException(e);
    }
    finally {
        try {
```

```
            stmt.close();
            pm.releaseCommunicationChannel();
        } catch(SQLException e) {
            throw new PersistenceSoftException(e);
        } catch(PersistenceMechanismException e) {
            throw new PersistenceSoftException(e);
        }
    }
}
public #BASIC search( #BASIC #< #BASIC.toVariableName() >#) throws
        #< #BASIC.addSuffix("NotFoundException", false) ># {
    Statement stmt = null;
    ResultSet rs = null;
    #BASIC #< #BASIC.toVariableName().addSuffix("Return") ># = null;
    String sql = #SQL_P;
    try {
        stmt = (Statement)pm.getCommunicationChannel();
        rs = stmt.executeQuery(sql);
        if(rs.next() == false) {
            throw new #< #BASIC.addSuffix("NotFoundException",
                                              false) ># ();
        }
        #< #BASIC.toVariableName().addSuffix("Return") ># =
            new #BASIC();
        forall #fd in #ATTRS_BASIC {
             forall #vd in #< #fd.getVariables() ># {
                 #if( #fd.getTypeCode() !=
                          cin.jats.engine.parser.nodes.JType.OBJECT) {
                     #< #BASIC.toVariableName().addSuffix("Return") >#
                        .#< #vd.addPrefix("set") >#(
                            rs.#< #fd.getDefaultQueryMethod() >#(
                                #<#MAP_TABLE.getTableEntry(
                                    #<#vd.getName()>#)># ));
                 } else {
                     #< #BASIC.toVariableName().addSuffix("Return") >#
                        .#< #vd.addPrefix("set") >#((#< #fd.getType() >#(
                            rs.getObject(
                                #<#MAP_TABLE.getTableEntry(
                                    #<#vd.getName()>#)># ) );
                 }
             }
        }
    } catch(SQLException e) {
        throw new PersistenceSoftException(e);
    } catch(PersistenceMechanismException e) {
        throw new PersistenceSoftException(e);
```

```
    } finally {
        try {
            rs.close();
            stmt.close();
            pm.releaseCommunicationChannel();
        } catch(SQLException e) {
            throw new PersistenceSoftException(e);
        } catch(PersistenceMechanismException e) {
            throw new PersistenceSoftException(e);
        }
    }
    return #< #BASIC.toVariableName().addSuffix("Return") >#;
}
public Collection searchAll() throws TransacaoException{
    Statement stmt = null;
    ResultSet rs = null;
    #BASIC #< #BASIC.toVariableName() ># = null;
    Collection collection = new Vector();
    String sql = #SQL_PT;
    try {
        stmt = (Statement)pm.getCommunicationChannel();
        rs = stmt.executeQuery(sql);
        while(rs.next()) {
            #< #BASIC.toVariableName() ># = new #BASIC();
            forall #fd in #ATTRS_BASIC {
               forall #vd in #< #fd.getVariables() ># {
                   #if( #fd.getTypeCode() !=
                         cin.jats.engine.parser.nodes.JType.OBJECT) {
                       #< #BASIC.toVariableName() >#
                          .#< #vd.addPrefix("set") >#(
                              rs.#< #fd.getDefaultQueryMethod() >#(
                                  #<#MAP_TABLE.getTableEntry(
                                      #<#vd.getName()>#)># ));
                   } else {
                       #< #BASIC.toVariableName() >#
                          .#< #vd.addPrefix("set") >#(
                              (#< #fd.getType() >#(
                                  rs.getObject(
                                      #<#MAP_TABLE.getTableEntry(
                                          #<#vd.getName()>#)># ));
                   }
               }
            }
            collection.add(#< #BASIC.toVariableName() >#);
        }
    } catch(SQLException e) {
```

```
            throw new PersistenceSoftException(e);
        } catch(PersistenceMechanismException e) {
            throw new PersistenceSoftException(e);
        }
        finally {
            try {
                rs.close();
                stmt.close();
                pm.releaseCommunicationChannel();
            } catch(SQLException e) {
                throw new PersistenceSoftException(e);
            } catch(PersistenceMechanismException e) {
                throw new PersistenceSoftException(e);
            }
        }
        return collection;
    }

}
```

B.3 Distribution templates

The following templates generate distribution aspects and auxiliary types. They use a source template that matches the facade class.

B.3.1 Facade source template

```
//Facade_lhs.ajats
PackageDeclaration:#PD_FACADE;
ImportDeclarationSet:#IDS_FACADE;
public class #FACADE #[ extends #SC_FACADE ]#
        #[ implements NameList:#IFS_FACADE ]#  {
    FieldDeclarationSet:#ATTRS_FACADE;
    InitializerSet:#IS_FACADE;
    ConstructorDeclarationSet:#CDS_FACADE;
    MethodDeclarationSet:#MTDS_FACADE;
}
```

B.3.2 Singleton target template

```
//Singleton_rhs.ajats
PackageDeclaration:#PD_FACADE;
ImportDeclarationSet:#IDS;
public class #FACADE #[ extends #SC_FACADE ]#
                     #[implements NameList:#IFS_FACADE ]#  {
    private static #< #FACADE ># singleton;
    FieldDeclarationSet:#ATTRS_FACADE;
    InitializerSet:#IS_FACADE;
    forall #cd in #CDS_FACADE {
       #< #cd.getModifiers().removeAllModifiers() >#;
       #< #cd.getModifiers()
          .addModifier(java.lang.reflect.Modifier.PRIVATE) >#;
    }
    ConstructorDeclarationSet:#CDS_FACADE;
    public #< #FACADE ># getInstance() {
        if (singleton == null) {
            singleton = new #< #FACADE >#();
        }
        return singleton;
    }
    MethodDeclarationSet:#MTDS_FACADE;
}
```

B.3.3 Server-side target template

```
//Server_side_rhs.ajats
package ##TARGET_PACKAGE##;
import cin.aspects.framework.distribution.AbstractServerSideAspect;
import java.rmi.Remote;
public aspect #< #FACADE.addSuffix("ServerSideAspect", false) >#
        extends AbstractServerSideAspect {
    public static final String SYSTEM_NAME     = "##SYSTEM_NAME##";
    public static final String RMI_SERVER_NAME = "##RMI_SERVER_NAME##";
    declare parents: #< #FACADE ># implements
                     #< #FACADE.addPrefix("IRemote", false) >#;
    declare parents: ##SERIALIZABLE_TYPE_LIST##
                     implements java.io.Serializable;
    Remote initFacadeInstance() {
        return #< #FACADE >#.getInstance();
    }
    String getSystemName() {
        return SYSTEM_NAME;
    }
    pointcut facadeMainExecution(String[] args):
        execution(static void #< #FACADE >#.main(String[])) &&
    args(args);
    public static void #< #FACADE >#.main(String[] args) {
         // just to enable the super aspect
    }
}
```

B.3.4 Client-side target template

```
//ClientSideAspect_rhs.ajats
package ##TARGET_PACKAGE##;
import cin.aspects.framework.distribution.AbstractServerSideAspect;
import cin.aspects.framework.updateState.UpdateStateControl;
import javax.servlet.http.HttpServlet;
import org.aspectj.lang.SoftException;
public aspect #< #FACADE.addSuffix("ClientSideAspect", false) >#
        extends AbstractClientSideAspect {
    private #< #FACADE.addPrefix("IRemote", false) >#
                #< #FACADE.toVariableName() >#;
    void setRemoteFacade(#< #FACADE.addPrefix("IRemote", false) ># remote) {
    #< #FACADE.toVariableName() ># = remote;
    }
    public Object getRemoteFacade() {
        prepareFacade();
        return #< #FACADE.toVariableName() >#;
    }
```

```
    pointcut facadeCallers():
        this(HttpServlet) ||
        within(UpdateStateControl);
    pointcut facadeCalls():
        call(* #< #FACADE.addPrefix("IRemote", false) >#+.*(..)) &&
        !call(static * #< #FACADE.addPrefix("IRemote", false) >#+.*(..));
    pointcut facadeLocalCalls():
        facadeCallers() &&
        facadeCalls();
    Object around():
        facadeCallers() &&
        call(* #< #FACADE.addSuffix("ServerSideAspect", false) >#
                .getInstance()) {
            return null;
    }
     private synchronized void prepareFacade() {
        if (#< #FACADE.toVariableName() >#  == null) {
            try {
                System.out.println("About to lookup...");
                #< #FACADE.toVariableName() ># =
                    (#< #FACADE.addPrefix("IRemote", false) >#)
                        java.rmi.Naming.lookup("//" +
                        #< #FACADE.addSuffix("ServerSideAspect", false) >#
                            .RMI_SERVER_NAME + "/" +
                        #< #FACADE.addSuffix("ServerSideAspect", false) >#
                            .SYSTEM_NAME);
                System.out.println("Remote #FACADE found");
            } catch (Throwable ex) {
                throw new SoftException(ex);
            }
        }
     }
}
```

Bibliography

[1] Vander Alves. Progressive development of distributed object–oriented applications. Master's thesis, Informatics Center — Federal University of Pernambuco, Brazil, February 2001.

[2] 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.

[3] Vander Alves and Paulo Borba. An Implementation Method for Distributed Object-Oriented Applications. In *XV Brazilian Symposium on Software Engineering*, pages 161–176, Rio de Janeiro, Brazil, October 2001.

[4] Scott Ambler. *Process Patterns-Building Large-Scale Systems Using Object Technology*. Cambridge University Press, 1998.

[5] Victor Basili, Richard Selby, and David Hutchens. Experimentation in Software Engineering. *IEEE Transactions on Software Engineering*, SE-12(7):733–743, July 1986.

[6] C. Becker and K. Geihs. Quality of service - aspects of distributed programs. In *International Workshop on Aspect Oriented Programming (ICSE 1998)*, February 1998.

[7] Lodewijk Bergmans and Mehmet Aksit. Composing Crosscuting Concerns Using Composition Filters. *Communications of the ACM*, 44(10):51–57, October 2001.

[8] 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.

[9] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings, second edition, 1994.

[10] Grady Booch, Ivar Jacobson, and James Rumbaugh. *Unified Modeling Language — User's Guide*. Addison-Wesley, 1999.

[11] 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.

[12] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *A System of Patterns: Pattern-Oriented Software Architecture*. John Wiley & Sons, 1996.

[13] Fernando Castor and Paulo Borba. A language for specifying Java transformations. In *V Brazilian Symposium on Programming Languages*, pages 236–251, Curitiba, Brazil, 23th–25th May 2001.

[14] Leonardo Cole. Deriving Refactorings for AspectJ. Master's thesis, Centro de Informática – Universidade Federal de Pernambuco, 2004. To appear.

[15] Leonardo Cole and Paulo Borba. Deriving Refactorings for AspectJ. In *Poster at OOPSLA'2004*, October 2004. To appear.

[16] D. Coleman, D. Ahs, B. Lowther, and P. Oman. Using Metrics to Evaluate Software System Maintainability. *IEEE Computer*, 24(8):44–49, August 1994.

[17] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison–Wesley, 2000.

[18] M. Ferreira Rito da Silva. *Concurrent Object–Oriented Programming: Separation and Composition of Concerns using Design Pattern, Pattern Languages and Object–Oriented Frameworks*. PhD thesis, Technical University of Lisbon, 1999.

[19] 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.

[20] Ramez Elmasri and Shamkant Navathe. *Fundamentals of Database Systems*. Addison–Wesley, second edition, 1994.

[21] Tzilla Elrad, Robert Filman, and Atef Bader. Aspect–Oriented Programming. *Communications of the ACM*, 44(10):29–32, October 2001.

[22] Robert Filman and Daniel P. Friedman. Aspect–Oriented Programming is Quantification and Obliviousness. In *Workshop on Advanced Separation of Concerns, OOPSLA'00*, 2000.

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

[24] Martin Fowler et al. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[25] Erich Gamma and Kent Beck. *Contributing to Eclipse: Principles, Patterns, and Plug-ins*. Addiso Wesley, first edition, October 2003.

[26] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[27] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, second edition, 2000.

[28] R. Graddy. Succesfully Applying Software Metrics. *IEEE Computer*, 27(9):18–25, September 1994.

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

[30] Jan Hannemann and Gregor Kiczales. Design Pattern Implementations in Java and Aspectj. In *Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications, OOPSLA'02*. ACM Press, November 2002.

[31] Erik Hilsdale and Gregor Kiczales. Aspect–oriented programming with AspectJ. In *OOPSLA'01, Tutorial*, Tampa FL, 2001.

[32] Jason Hunter and Willian Crawford. *Java Servlet Programming*. O'Reilly & Associates, Inc., first edition, 1998.

[33] Object Technology International Inc. Eclipse Platform Technical Overview. White Paper. Disponível em http://www.eclipse.org/, Julho 2001.

[34] Lieberherr K. J., Silva-Lepe I., and et al. Adaptive Object–Oriented Programming Using Graph–Based Customization. *Communications of the ACM*, 37(5):94–101, 1994.

[35] Ivar Jacobson. Object-oriented development in an industrial environment. In *Proceedings of the OOPSLA'87 conference on Object-oriented programming systems, languages and applications*, pages 183–191. ACM Press, December 1987.

[36] Ivar Jacobson. Aspects: The missing link. *Software Development*, October 2003. Avaliable at http://www.sdmagazine.com.

[37] Ivar Jacobson. The case for aspects. *Software Development*, September 2003. Avaliable at http://www.sdmagazine.com.

[38] Ivar Jacobson. Use cases and aspects - working seamlessly together. *Journal of Object Technology*, July/August 2003. Avaliable at http://www.jot.fm.

[39] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.

[40] Raj Jain. *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation and modeling*. Wiley, 1991.

[41] 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.

[42] 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.

[43] 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.

[44] Uirá Kulesza, Cláudio Sant'Anna, Alessandro Garcia, Carlos Lucena, and Arndt von Staa. Evaluating distribution and persistence concerns implementation using AspectJ. Technical report, Informatics Department, PUC-Rio, 2004. To appear.

[45] Ramnivas Laddad. I want my AOP!, Part 1: Separate software concerns with aspect-oriented programming. *JavaWorld*, January 2002. Avaliable at http://-www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect.html.

[46] Ramnivas Laddad. I want my AOP!, Part 2: Learn AspectJ to better understand aspect-oriented programming. *JavaWorld*, March 2002. Avaliable at http://www.-javaworld.com/javaworld/jw-03-2002/jw-0301-aspect2.html.

[47] Ramnivas Laddad. I want my AOP!, Part 3: Use AspectJ to modularize crosscutting concerns in real-world problems. *JavaWorld*, April 2002. Avaliable at http://www.javaworld.com/javaworld/jw-04-2002/jw-0412-aspect3.html.

[48] Eduardo Laureano. Persistence implementation with AspectJ. Master's thesis, Informatics Center — Federal University of Pernambuco, Brazil, January 2002.

[49] Doug Lea. *Concurrent Programming in Java.* Addison–Wesley, second edition, 1999.

[50] Cristina Lopes and Gregor Kiczales. D: A language framework for distributed programming. Technical report, Xerox Palo Alto Research Center, 1997.

[51] 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, July 2004.

[52] Tiago Massoni. A Software Process with Progressive Implementation Support (in portuguese). Master's thesis, Informatics Center — Federal University of Pernambuco, Brazil, February 2001.

[53] 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.

[54] Hidehiko Masuhara and Gregor Kiczales. Modular Crosscutting in Aspect-Oriented Mechanisms. In *European Conference on Object-Oriented Programming, ECOOP'2003.* Springer-Verlag, July 2003.

[55] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice-Hall, second edition, 1997.

[56] Sun Microsystems. JavaCC Project. Available at https://javacc.dev.java.net.

[57] Sun Microsystems. Package java.sql. Available at http://java.sun.com/products/-jdk/1.2/docs/api/java/sql/package-summary.html.

[58] Sun Microsystems. Java Remote Method Invocation (RMI). Disponível em http://java.sun.com/products/jdk/1.2/docs/guide/rmi, 2001.

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

[60] Richard Monson-Haefel. *Enterprise JavaBeans*. Oreilly, second edition, 2000.

[61] 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.

[62] Gail Murphy, Robert Walker, and Elisa Baniassad. Evaluating Emerging Software Development Technologies: Lessons Learned from Assessing Aspect-Oriented Programming. *IEEE Transactions on Software Engineering*, 25(4):438–455, July/August 1999.

[63] 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.

[64] Bashar Nuseibeh. Crosscutting requirements. In *Proceedings of the 3rd international conference on Aspect-oriented software development, AOSD'04*, pages 3–4. ACM Press, March 2004.

[65] Robert Orfali and Dan Harkey. *Client/Server Programming with Java and CORBA*. Wiley, 1998.

[66] H. Ossher, M. Kaplan, A. Katz, W. Harrison, and V. Kruskal. Specifying suject–oriented composition. *TAPOS*, 2(3):179–202, 1996. Special Issue on Subjectivity in OO Systems.

[67] Harold Ossher and Peri Tarr. Using subject–oriented programming to overcome common problems in object–oriented software development/evolution. In *International Conference on Software Engineering, ICSE'99*, pages 698–688. ACM, 1999.

[68] Harold Ossher and Peri Tarr. Hyper/J: multi–dimensional separation of concerns for Java. In *22nd International Conference on Software Engineering*, pages 734–737. ACM, 2000.

[69] Shari Pfleeger. Design and Analysis in Software Engineering, Part 1: The Language of Case Studies and Formal Experiments. *Software Engineering Notes*, 19(4):16–20, October 1994.

[70] Shari Pfleeger. Experimental Design and Analysis in Software Engineering, Part 2: How to Set Up an Experiment. *Software Engineering Notes*, 20(1):22–26, January 1995.

[71] Shari Pfleeger. Experimental Design and Analysis in Software Engineering, Part 3: Types of Experimental Design. *Software Engineering Notes*, 20(2):14–16, April 1995.

[72] Shari Pfleeger. Experimental Design and Analysis in Software Engineering, Part 4: Choosing an Experimental Design. *Software Engineering Notes*, 20(3):13–15, July 1995.

[73] Shari Pfleeger. Experimental Design and Analysis in Software Engineering, Part 5: Analyzing the Data. *Software Engineering Notes*, 20(5):14–17, December 1995.

[74] Roger Pressman. *Software Engineering: A Practitioners Approach*. McGraw-Hill, 5th edition, 2000.

[75] 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.

[76] Rational. Rational Web pages, Rational Software Corporation. At http://www.-rational.com.

[77] Yau S. and Collofello J. Some Stability Measures for Software Maintenance. *TSE*, SE–6, 1980.

[78] Carolyn Seaman. Qualitative Methods in Empirical Studies of Software Engineering. *IEEE Transactions on Software Engineering*, 25(4):557–572, July/August 1999.

[79] L. Seinturier. Jst: An object synchronization aspect for java. In *Workshop on Advanced Separation of Concerns (ECOOP 1999)*, June 1999.

[80] Sérgio Soares. Progressive Develpoment of Object Oriented Concurrent Programs (in portuguese). Master's thesis, Informatics Center (CIn) — Federal University of Pernambuco (UFPE) — Brazil, February 2001.

[81] Sérgio Soares and Paulo Borba. Concurrency Control with Java and Relational Databases (in portuguese). In *V Brazilian Symposium on Programming Languages*, pages 252–267, Curitiba, Brazil, May 23-25 2001.

[82] Sérgio Soares and Paulo Borba. Concurrency Manager. In *First Latin American Conference on Pattern Languages of Programming — SugarLoafPLoP*, pages 221–231, Rio de Janeiro, Brazil, October 2001. Published in UERJ Magazine: Special Issue on Software Patterns.

[83] Sérgio Soares and Paulo Borba. AspectJ — Aspect-Oriented Programming in Java (in portuguese). In *Tutorial in SBLP 2002, VI Brazilian Symposium on Programming Languages.*, pages 39–55, PUC-Rio, Rio de Janeiro, Brazil, June 5-7 2002.

[84] Sérgio Soares and Paulo Borba. Concurrency Control with Java and Relational Databases. In *Proceedings of 26th Annual International Computer Software and Applications Conference*, pages 834–849, Oxford, England, August 2002. IEEE Computer Society Press.

[85] 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.

[86] 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.

[87] Sérgio Soares and Paulo Borba. Progressive implementation with aspect–oriented programming. In *The 12th Workshop for PhD Students in Object–Oriented Systems, ECOOP'02*, volume 2548 of *LNCS (Lecture Notes in Computer Science)*, pages 44–54, Málaga, Spain, June 2002. Springer Verlag.

[88] Sérgio Soares and Paulo Borba. An aspect-oriented implementation method. *Student Research Extravaganza (Poster Session), International Conference on Aspect-Oriented Software Development, AOSD 2004.* http://www.aosd.net/2004/extravaganza.php. Lancaster, UK, March 2004.

[89] Sérgio Soares, Marcelo d'Amorim, Denise Neves, Marcelo Faro, Luciana Valadares, Gibeon Soares, and Antonio Valenca. Implementing Object-Oriented Web Systems Using Java Servlets (in portuguese). In *IV Brazilian Symposium on Programming Languages*, pages 290–299, Recife, Brazil, May 17-19 2000.

[90] Sérgio Soares, Eduardo Laureano, and Paulo Borba. Distribution and Persistence as Aspects. *Software: Practice & Experience*. Submitted, August 2004.

[91] 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).

[92] Geórgia Sousa, Sérgio Soares, Paulo Borba, and Jaelson Castro. Separation of Crosscutting Concerns from Requirements to Design: Adapting the Use Case Driven Approach. In Bedir Tekinerdoan, Ana Moreira, Jo ao Araújo, and Paul Clements, editors, *Early Aspects 2004: Aspect-Oriented Requirements Engineering and Architecture Design. Workshop at International Conference on Aspect-Oriented Software Development, AOSD 2004, Workshop Report.*, pages 93–102 (97–106), March 2004.

[93] Sun Microsystems. The Enterprise JavaBeans Specification Version 2.1, August 2002. At http://java.sun.com/products/ejb.

[94] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton. N degrees of separation:multi–dimensional separation of concerns. In *1999 International Conference on Software Engineering*, pages 107–119. ACM, 1999.

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

[96] Michiaki Tatsubori, Shigeru Chiba, Marc-Olivier Killijian, and Kozo Itano. Openjava: A class-based macro system for java. In Walter Cazzola, Robert J. Stroud, and Francesco Tisato, editors, *Lecture Notes in Computer Science 1826, Reflection and Software Engineering*, pages 117–133. Springer-Verlag, 2000.

[97] AspectJ Team. The AspectJ Programming Guide. At http://dev.eclipse.-org/viewcvs/indextech.cgi/~checkout~/aspectj-home/doc/progguide/index.html http://eclipse.org/aspectj, 2003.

[98] Bedir Tekinerdogan, Paul Clements, Ana Moreira, and Jo ao Araújo. Early aspects 2004: Aspect-oriented requirements engineering and architecture design. In *Workshop at International Conference on Aspect-Oriented Software Development, AOSD 2004*, March 2004.

[99] Walter Tichy. Should Computer Scientists Experiment Mode? *IEEE Computer*, 31(5):32–40, May 1998.

[100] Guilherme Travassos, Dmytro Gurov, and Edgar Amaral. Introduction to experimental software engineering (in portuguese). Technical Report ES-590/02, Computer Systems Engineering Program, COPPE/UFRJ, April 2002.

[101] Euricelia Viana and Paulo Borba. Integrating Java with Relational Dadabases (in portuguese). In *III Brazilian Symposium on Programming Languages*, pages 77–91, Porto Alegre, Brazil, May 1999.

[102] 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.

[103] Seth White and Mark Hapner. JDBC 2.1 API. Version 1.1. Sun Microsystems, October 1999.

[104] C. Wohlin, P. Runeson, M. Höst, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, 2000.

[105] Marvin Zelkowitz and Dolores Wallace. Experimental Models for Validating Technology. *IEEE Computer*, 31(5):23–31, May 1998.

[106] 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.

Tese de Doutorado apresentada por **Sérgio Castelo Branco Soares** a Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título **"An Aspect-Oriented Implementation Method"**, orientada pelo **Prof. Paulo Paulo Henrique Monteiro Borba** e aprovada pela Banca Examinadora formada pelos professores:

Prof. Silvio Romero de Lemos Meira
Departamento de Informação e Sistemas - CIn / UFPE

Prof. Augusto César Alves Sampaio
Departamento de Ciência da Computação – CIn / UFPE

Prof. Jaelson Freire Brelaz de Castro
Departamento de Ciência da Computação - CIn / UFPE

Prof. Guilherme Horta Travassos
COPPE – Programa de Engenharia e Sistemas / UFRJ

Prof. Paulo César Masiero
Departamento de Ciência da Computação e Estatística / USP

Visto e permitida a impressão.
Recife, 4 de outubro de 2004.

Prof. JAELSON FREIRE BRELAZ DE CASTRO
Coordenador da Pós-Graduação em Ciência da Computação do
Centro de Informática da Universidade Federal de Pernambuco.