Progressive implementation with aspect–oriented programming

Sérgio Soares* and Paulo Borba †
Informatics Center
Federal University of Pernambuco

Abstract

The object–oriented paradigm has some limitations, such as tangled code and code spread over several units, making harder software maintainability. Some of these limitations maybe compensated by design patterns and implementation methods guiding the software structure. On the other hand, extensions of the object–oriented paradigm, such as aspect–oriented programming, try to solve the object–oriented limitations. These techniques allow higher software modularity making easer software reuse and maintainability in practical situation where the object–oriented paradigm, design patterns, and implementation methods do not offer an adequate support. The aim of the work presented here is to analyze if different concerns (non–functional requirements) can be implemented and tested separately, in a progressive way, providing quality software and development productivity. A general–purpose aspect–oriented extension to Java, called AspectJ, is being used to define the aspect–oriented progressive approach implementing its crosscutting concerns (persistence, distribution and concurrency control). This separation of concerns allows better modularity, by avoiding tangled code and code spread over several units, and therefore increasing software maintainability.

Keywords: Aspect–oriented programming, separation of concerns, frameworks, patterns, software engineering, Java.

1 Introduction

The need for developing quality software increased the use of object–orientation looking for greater reuse and better maintainability, increasing the development productivity and supporting requirements changes. However, the object–oriented paradigm has some limitations [14, 15], such as tangled code and code spread over several units, making harder software maintainability. Some of these limitations maybe compensated by design patterns [4, 8] and implementation methods [3, 10].

On the other hand, extensions of the object–oriented paradigm, such as aspect–oriented programming [6], composition filters [1], subject–oriented programming [15], and adaptive programming [9], new programming techniques, are trying to solve the object–oriented limitations. These techniques allow easer software reuse and maintainability in practical situation where the object–oriented paradigm, design patterns, and implementation methods do not offer an adequate support [18]. However, there are some limitations of these techniques that demand tool support [18] to keep high the maintainability level.

The aim of this work is to analyze if different non–functional concerns can be implemented and tested separately, possibly by different teams, in a progressive way, increasing software quality and development productivity. If this is possible, we could support a progressive approach

*PhD. Student. Email: scbs@cin.ufpe.br
†Advisor. Email: phmb@cin.ufpe.br

for object–oriented implementation, where persistence, distribution, and concurrency control are not initially considered in the implementation activities, but are gradually introduced, preserving the application's functional requirements. This approach allows early functional requirements validation, as the functional requirements can be fully implemented without the effort to implement the non–functional ones, which will be implemented only after the validation. In fact, this progressive implementation was partially validated with design patterns, however, without full separation of concerns, which is our aim in using AOP.

A general–purpose aspect–oriented extension to Java, called AspectJ [11] is being used to define these crosscutting concerns (persistence, distribution and concurrency control) separated from the system core (functional requirements). This separation of concerns allows better modularity, by avoiding tangled code and code spread over several units. The system maintainability is also increased, since it is just needed to implement a new aspect and then weaving (process that composes aspects with the original system to create the final system) it with the system to add a new concern, or a new implementation of a concern.

This paper is structured in five sections. Section 2 shows a software architecture used by the system, in which crosscutting concerns are implemented, and an overview about AspectJ. Distribution aspects are showed in Section 3 describing a particular aspect of the Ph.D. work. Related works are discussed in Section 4, and finally, Section 5 presents our conclusions.

2 Background

This section explains the software architecture of the system used in the implementation, and presenting an overview of AspectJ, the Aspect–Oriented language used to define the aspects.

2.1 Software architecture

The progressive implementation approach is tailored to a specific software architecture that implements a layer architecture using design patterns [8, 4]. This software architecture aims to separate data management, business rules, communication (distribution), and presentation (user interface) concerns. Such structure prevents tangled code, for example, business code interlacing with data access code, and design patterns allow us to reach greater reuse and extensibility levels. However, this goal is not reached at all; there are some tangled code, for example, code regarding exception handling of each concern, code regarding concurrency control, and code that defines which interface implementation for data management will be used. In addition, the architecture does not prevent some spread code, for example, code regarding which classes has to be serialized in distribution environment is scattered in the system, as well as the exception handling and the concurrency control code.

AspectJ was chosen to implement the crosscutting concerns in order to avoid some tangled code that still remains when using the layer architecture, as well as spread code, and to restructure the system to become easier maintain and evolve it, without invasive changes. The aspects definition sometimes eliminates the need for design patterns usage, since it modularizes in a higher level the crosscutting concerns.

Figure 1 presents an UML [2] diagram of the specific software architecture. Accesses to the system are made through a unique entry point, the system facade [8]. In this software architecture the system facade also implements the Singleton [8] design pattern to guarantee that there is just a single instance of this class. This is the class whose instance should be distributed over the user interfaces. The facade is composed of business collections, target of facade methods delegation. The persistence mechanism interface is responsible to abstract which persistence mechanism is in use. Classes implementing this interface should handle databases connections, transaction management and sharing of databases communication channels among concurrent users. Persistent data collections are used to map flat data into business objects, and vice versa.

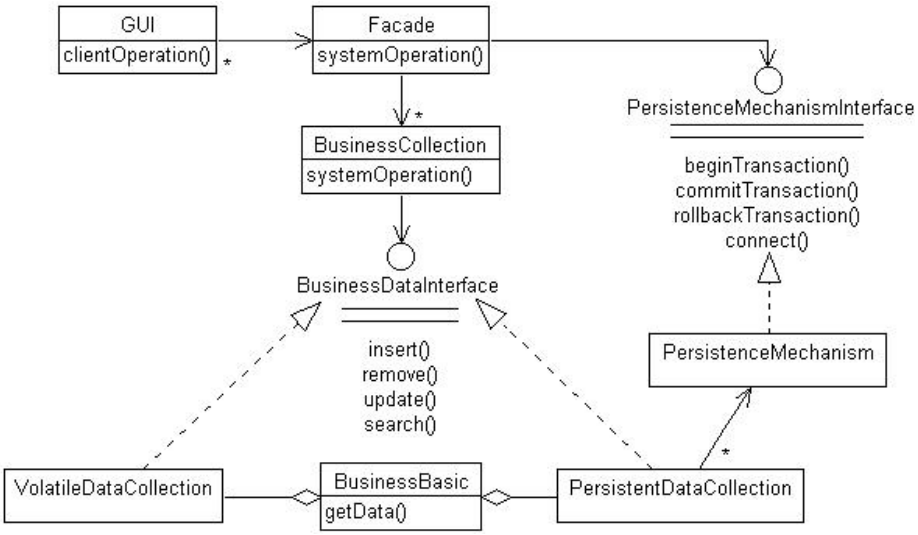

Figure 1: Software architecture's class diagram.

Those collections are used by business collections through business–data interfaces. These interfaces allow multiple implementations where each data collection access different mechanisms, even though volatile implementation.

2.2 AspectJ overview

AspectJ [11] is a general–purpose aspect–oriented extension to Java. The aspect–oriented paradigm makes possible to define crosscutting concerns separated from each other. This separation of concerns allows better modularity, by avoiding tangled code and code spread over several units. The system maintainability is also increased, since it is just needed to implement a new aspect and then weaving (process that composes aspects with the original system to create the final system) it to add a new concern, or a new implementation of a concern in the system.

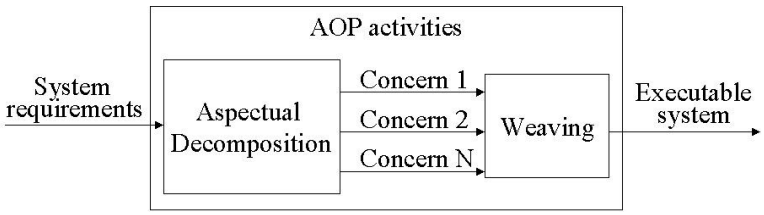

Figure 2: AOP development phases.

Figure 2 illustrates the aspectual decomposition, which identifies the crosscutting concerns of a system, and the weaving, which composes the identified concerns with the system to obtain the final version with the required functions.

Features

The main construct of the AspectJ language is an *aspect*. Each *aspect* defines a functionality that crosscuts others, called concerns, in a system.

An *aspect* can have attributes, methods, and a hierarchy of aspects, by defining specialized aspects. The *AspectJ introduction* mechanism can affect the static structure of the programs

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

Figure 3: AspectJ definition example.

by introducing new methods and fields to an existing a class, convert checked exceptions into unchecked exceptions, and changing the class hierarchy, by extending an existing class or interface with another or implementing an interface in an existing class.

Furthermore, an aspect can also affect the dynamic structure of a program changing the way a program executes, by intercepting points of the program execution flow, called *join points*, and adding behavior *before*, *after*, or *around* (instead of) the *join point*. Examples of *join points* are method calls, method executions, instantiations, constructor executions, field references (get and set), exception handling, static initializations, others, and combinations of these by using the !, && or || operators.

Usually, an aspect defines a *pointcut* that selects some *join points* and values at those *join points*. Then an *advice* defines the code that is executed when a *pointcut* is reached. The *advice* is who defines if the code executes *before*, *after*, or *around* the *pointcut*.

Figure 3 shows an aspect definition example, from the AspectJ Programming Guide, where the aspect `FaultHandler` adds one field onto `Server` class (line `3`), defines two methods (lines `5` to `7` and `9` to `11`). The aspect also defines one *pointcut* (line `13`), and two *advice* (lines `15` to `17` and `19` to `22`). The introduced field flags the server's objects if the execution of any `public` method (identified by the *pointcut* definition) throws a `FaultException`, which is defined by the *after advice*. The *before advice* checks the server's flag before calling any `public` method (identified by the *pointcut* definition) avoiding methods call to disabled servers.

3 Distribution Aspects

This section overviews guidelines for implementing distribution code using AspectJ. These guidelines were derived from an experience on implementing distribution aspects in a real information system, a health complaint system. The distribution aspects implement basic remote access to system services using RMI (Remote Method Invocation) [12] and exception handling. In fact the aspects definition follows a pattern that can be used to implement distribution using another technology, like CORBA [13]. The aspects are divided in server–side, client–side, and exception handling aspects. Figure 4 shows how the aspects and the system are structured.

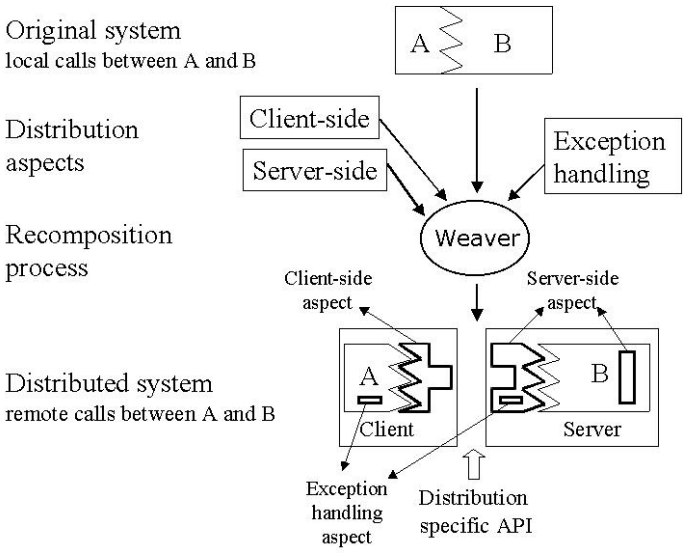

Figure 4: Distribution aspects' structure.

When the distribution aspects are woven with the system code that uses the software architecture presented in Section 2.1, it affects the system facade (server–side aspect) and the user interface classes (client–side aspect), making the communication between them remote, by distributing the facade instance. In fact, the server–side aspect might crosscuts others classes that are return types or arguments type of the facade methods.

The exception handling aspect affects both facade and the user interface to provide the respective handling regarding new exceptions introduced by the aspects.

3.1 Server–side distribution aspect

The server–side distribution aspect is responsible to distribute the facade instance allowing remote calls to its methods. Therefore, to distribute an object using RMI the aspect should modify its class and the classes whose objects will be received as parameters and returned by the methods of the facade class.

The RMI API demands that the interface methods of the class whose object will be remote must declare that it can throws a specific RMI exception. Therefore, this exception should added in the `throws` clause of the facade methods. However, the actual version of AspectJ does not allow it. As a consequence of this problem a remote interface should defined as an aspect auxiliary interface, which extends the `java.rmi.Remote` interface and has all facade methods signatures with the specific RMI exception in its `throws` clause. This is a problem regarding software evolution, because every time a new method is added in the facade, the remote interface should be modified to add this new method signature with the API–specific exception in its `throws` clause. However, a tool can mostly automate the aspect definition, increasing the aspects productivity and reuse. Another bad consequence of this limitation is discussed in the client–side aspect definition.

A *change request* was submitted in the AspectJ website, requesting a new introduction construction, which would add an exception in a method throws clause.

After that the server–side aspect modifies the facade class to implement the auxiliary interface previously defined using the introduction mechanism. Additionally, the aspect exports the facade object making it available to accept incoming calls from clients. The server–side aspects also modifies the classes whose objects are arguments or return values of the facade methods. They just have to use the Java Object Serialization mechanism, by implementing

the `java.io.Serializable` interface

3.2 Client–side distribution aspect

The simplest way to define the client–side aspect would be just changing the client's facade attribute initialization to the get a reference to the remote instance. However, because of the "adding exception in a throws clause" problem the remote instance is not a subtype of the facade class, which is the attribute's type of the client classes. In fact, the remote instance and the facade class have the same super type, the remote interface.

A solution is writing an advice for each facade method, redirecting the facade local executions to its remote instance. However, this decreases the aspect maintainability since for every new facade method an advice should be wrote to redirect the local call to the remoter instance. As soon as the "adding exception problem" is solved a simplest solution can be used, changing the client's facade attribute initialization to the get a reference to the remote instance.

3.3 Exception handling

Another aspect to is exception handling. The AspectJ police to handle with exceptions introduced by the aspects definition is encapsulating them in to an unchecked exception, called *soft* exception. Therefore, this unchecked exception should be handled in the user interface classes. Note that exception handling is a natural crosscutting concern, usually implemented with spread code.

The exception handling aspect crosscuts both, user interfaces and facade classes. The aspect should encapsulate new exceptions in a `SoftException` at the server–side, and handling them at the client–side, providing the necessary behavior, usually showing messages to the user interface.

4 Related works

A related work [5] uses design pattern, pattern languages and object–oriented frameworks to provide separation of concerns regarding concurrency and distribution programming. Our approach implements, or will implement, those concerns using AspectJ, without invasive code changing, with is necessary to implement the design patterns or using frameworks. Those invasive changes avoids the obliviousness [7] provided by the aspects definition regarding the abstraction that the system core programmers might have about the aspects definition.

Another related work [17] defines a tool to support aspect-oriented distributed programming. It defines a language to specify the classes whose objects will be distributed and the hosts that these objects must execute. The tool includes the feature of deploying the classes code at those hosts. Besides our approach does not consider just distribution concerns, but also concurrency control and persistence, we will also implement a tool support for implement those aspects, including distribution.

An approach that could be used to implement distribution aspect is Composition Filters (CF) [1]. This approach can filter messages received and invoked by objects associating actions to them. For example, using CF we could filter messages between the components to have their communication distributed, and associate the action of redirecting the communication to the remote instance. However, AspectJ is a general approach for separation of concerns. For example, CF does not offer static crosscutting and some kinds of dynamic crosscutting that AspectJ does. As our approach also considers persistence and concurrency control aspects, which might not be implemented using composition filters, we use AspectJ.

5 Conclusion

This position paper presented the Ph.D. project that aims in analyzing if different concerns (non–functional requirements) can be implemented and tested separately, in a progressive way. These concerns will be implemented using AspectJ, a general–purpose aspect–oriented extension to Java. It also presented an overview of guidelines for implementing distribution aspects with AspectJ and RMI, which were derived from an experience on restructuring a real information system. In fact, those guidelines were originally defined for both restructuring and implementing distribution aspects. Those guidelines are also a pattern to implement distribution aspects using different APIs. Guidelines for both restructuring and implementing persistence aspects with AspectJ are already defined and can be found elsewhere [16], including the original definition of the distribution guidelines. At this moment, the guidelines for concurrency control are being defined, as well as a tool to completely or partially automatize the system refactoring and the aspects generation.

The guidelines are based on an AspectJ framework and on related patterns, which increases the productivity of projects that use the guidelines. The framework and the patterns were identified when restructuring the system with AspectJ. Although the guidelines are tailored to a specific architecture and related patterns, it has been widely used to implement a wide range of real web–based information systems. The use of a specific architecture allows the definition of precise and more useful guidelines, giving better support to AspectJ programmers.

This experience validates AspectJ and indicates that this language is useful to separately implement the distribution and persistence concern considered [16]. However, a few drawbacks in the language ware identified and reported. This was the motivation for suggesting here some minor modifications to AspectJ but that could significantly improve the implementations that follow the guidelines.

This project will also analyze the aspect–oriented progressive implementation approach through a case study in a real software company. Another analyzes, based in experiments, will show what is the impact of using an AOP technique and what is the impact of using a progressive approach to implement systems.

6 Acknowledgements

CAPES and CNPq, the Brazilian research agencies, supported this work. CAPES supports the first author and the second is supported in part by CNPq, grant 521994/96–9.

References

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

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

[3] 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.

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

[5] 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.

[6] Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect–oriented programming. *Communications of the ACM*, 44(10):29–32, October 2001.

[7] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Workshop on Advanced Separation of Concerns, OOPSLA'00*, 2000.

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

[9] 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.

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

[11] 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.

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

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

[14] 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.

[15] 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.

[16] Sérgio Soares, Eduardo Laureano, and Paulo Borba. Implementing distribution and persistence aspects with AspectJ. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'02)*, Seattle, WA, USA, 4th–8th November 2002. ACM Press. To appear.

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

[18] Detlef Vollmann. Visibility of Join–Points in AOP and Implementation Languages. In *Second Workshop on Aspect-Oriented Software Development*, Bonn, Germany, February 2002.