Concurrency Control Modularization with Aspect-Oriented Programming

Sérgio Soares[†] Paulo Borba[⋆] Ricardo Lima[†]
[†]Pernambuco State University
Computing Systems Department
Rua Benfica, 455, CEP 50720-001
Recife-PE, Brazil
{sergio,ricardo}@dsc.upe.br

[⋆]Federal University of Pernambuco
Informatics Center
Av. Professor Luis Freire sn, CEP 50740-540
Recife-PE, Brazil
phmb@cin.ufpe.br

Abstract

Concurrent programs are essential in the development of web based information systems. The wide dissemination of these systems increased the need for methods to create correct and efficient concurrent programs, which are usually difficult to implement and test. This paper presents guidelines to improve the concurrent control structure of object-oriented software using aspect-oriented programming through AspectJ based on an existent object-oriented concurrency control implementation method. We have defined a simple aspect framework that can be extended to implement concurrency control in applications complying with a software architecture presented in this paper. The framework comprises a set of reusable aspects useful for modularizing concurrency control.

1. Introduction

The correctness and efficiency of web based information systems relies greatly on the quality of concurrent programs. In turn, concurrent environments increase the complexity of implementation and test activities, since subtle implementation error leads to incorrect program execution. This scenario indicates the need for using more rigorous methods to implement concurrent programs. In this context a concurrency control method (CCM) should drive the implementation of concurrent control through a set of programming rules clearly stated as a set of techniques. The adoption of a CCM has the advantage of avoiding the inclusion of concurrent control structures (CCS) based only on the programmer intuition. This intuitive programming strategy may have a negative impact in the efficiency of systems and is usually error prone. Additionally, it may introduce new race conditions in the code, yielding invalid executions in concurrent environments. On the other hand, CCMs contribute to improve the system correctness, documentation, and standardizes the control to be applied, favoring system safety, extensibility, and maintainability.

In this paper we use AspectJ [3], a general purpose aspect-oriented [1] extension to Java, to define aspects used to implement the CCMs presented elsewhere [7]. These aspects are tailored to a software architecture, useful to several kinds of software as stated in Section 2. The proposed aspects complement the CCM for object-oriented software by modularizing concurrency control, which was not possible to achieve without an aspect-oriented approach.

The main contributions of this paper are summarized below:

- *guidelines to restructure an object-oriented to an aspect-oriented concurrency control;*
- *guidelines to implement aspect-oriented concurrency control, derived from the previous;*
- *a framework of reusable aspects that can be used in several applications;*
- *examples of application-specific aspects that can be used as aspect implementation pattern for different applications that use the framework.*

This paper is structured in five sections. In Section 2 we present a specific software architecture and the kinds of class that implement this architecture using design patterns [2]. An overview about AspectJ, the aspect-oriented language we use, is presented in Section 3. After that, in Section 4, we define guidelines for aspect-oriented concurrency control, and finally, Section 5 presents the conclusions and future work.

2. Software Architecture

The guidelines presented in this paper are tailored to the software architecture [8] presented in this section. Several systems have been implemented using this architecture. One of them was adopted to derive the aspects proposed in this work as well as in several experiments.

Figure 1 depicts an UML class diagram for a piece of the Health Watcher System. The diagram includes examples of each kind of class and interface of the object-oriented software architecture. UML stereotypes are employed to identify such classes and interfaces. The Health Watcher 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 against companies (such as restaurants and food shops) and institutions, so that health care institutions can promptly investigate the complaints and take the required actions. The system has a web-based user interface for registering complaints and performing several other associated operations.

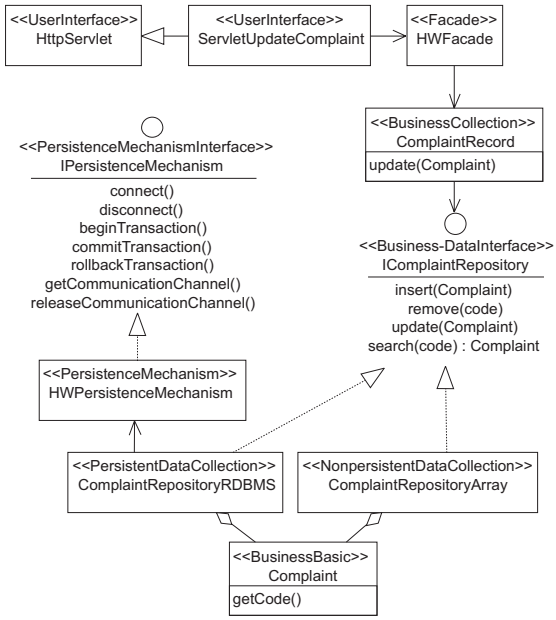

Figure 1. Software architecture class diagram.

The object-oriented version of the software architecture was primarily conceived to implement a four layer architecture. It separates user interface, communication, business rules, and data management concerns. In this architecture the concurrency control code was spread over different modules of the software, being tangled with user interface, communication, business rules, and data management concerns. By using AspectJ we could derive a fifth layer, orthogonal to the others, responsible for implementing the concurrency control approach.

3. AspectJ overview

AspectJ [3] is a general purpose aspect-oriented extension to Java. The aspect-oriented constructs support the separate definition of units of a program which affect (crosscut) other concerns. Such units are called crosscutting concerns. These concerns often cannot be cleanly decomposed from the rest of the system in both the design and implementation, and result in either scattering or tangling code, or both. Thus, this separation of concerns allows better modularity, avoiding tangled code and code spread over several units. Consequently, the system maintainability is also increased. Programming with AspectJ explores both objects and aspects concepts to separate concerns. Object-oriented programming can be used when the concern are well modeled as objects. If it is not the case, concerns that crosscut the objects are separated using units called aspects, and those are composed with the objects of a system by a process called weaving. By weaving AspectJ aspects with standard Java code, we obtain a new AspectJ application.

The main construct of the AspectJ [3] language is called aspect. Each aspect defines a functionality that crosscuts others (crosscutting concerns) in a system. An aspect can declare attributes and methods, and can extend another aspect by defining concrete behavior for some abstract declarations. An aspect can affect both static and dynamic structure of Java programs. The static structure might be changed by introducing new methods and fields to an existing class, as well as converting checked exceptions into unchecked exceptions, and changing the class hierarchy. The dynamic structure is changed by intercepting specific points, called join points, of the program execution flow and adding behavior before, after, or around the join point.

4. From object-oriented to aspect-oriented concurrency control

In this section, we present steps to restructure object-oriented software by removing the concurrency control (CC) code tangled and spread over software units and implementing aspects to modularize this CC. The following list is a summary of the restructuring guidelines presented through this section.

- *Identify and document where and what concurrency control is applied in the object-oriented system (Section 4.1);*
- *Remove the applied concurrency control (Section 4.2);*
- *Implement the removed concurrency control as aspects using the aspect framework (Section 4.3).*

The steps are guidelines to evolve object-oriented systems with aspect-oriented CC. An important step recommends identifying and removing CC code. If this recommendation is ignored, the guidelines are simplified into implementation rules to aspect-oriented software programming.

- *Implement the system with Java without any concurrency control;*
- *Apply the Concurrency Control Implementation Method [7] to identify where and what concurrency control should be applied in the system to guarantee safety and performance;*
- *Implement the concurrency control as stated by the method with aspects using the aspect framework (Section 4.3).*

The following sections present some details of each guideline, the complete details can be found elsewhere [5].

4.1. Identifying existing CC code

This subsection provides guidelines to identify CC code in a system. The CC implementation method introduced in [7] precisely defines which kind of CC is required and what modules of the software need to be controlled. This implementation method forms the basis for the guidelines proposed here. It is worth notice that the method itself is not a contribution of this paper. In fact, based on our experience with the method, we propose the modularization of the CC implementation through aspect-oriented programming.

The types of CC are listed below:

- `synchronized` *Java modify [4] — synchronized methods of an object cannot be concurrently executed;*
- `synchronized` *Java block [4] — 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 [4] — used to implement monitors behavior;*
- *use of the Concurrency Manager design pattern [6] — 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 update of objects copies that might lead to an inconsistent state.*

In the method definition [7] we identified which CCs are most frequently applied in the classes of the software architecture described in Section 3, which are summarized by Table 1.

4.2. Removing the CC code

After identifying where the CC code is located, we must remove it. We should also document which CCs are implemented in the software in order to implement them back later with aspects. The documentation might use the summary presented in Table 1 to indicate where and which CC might be used in the system being restructured, as defined by the CC implementation method.

Class type	Concurrency control
User interface	No concurrency control
Communication	No concurrency control
Facade	Transactions if in persistence environment. However, transactions are already implemented by persistence aspects
Business collection	Concurrency Manager or `synchronized` modifier
Data collection	`synchronized` modifier and additional SQL commands of the persistent collections
Basic	Timestamp field and related methods

Table 1. Most common concurrency control.

4.3. Implementing CC aspects

At this point we should define aspects to implement, in a modular way, the CC we removed as described in the former section.

The framework we defined supports four type of CC: `synchronized` *modify,* `synchronized` *block, use of Concurrency Manager design pattern, and use of timestamp technique.*

CC aspects framework

In order to allow aspect reuse we defined abstract aspects that constitute a simple aspect framework. Therefore, this framework (abstract aspects) should be extended for implementing CC in applications that comply with the layer architecture presented in Section 2.

The one of the abstract aspects of the framework is responsible for identifying join points that cannot execute concurrently, and therefore, should be synchronized.

```
1: abstract aspect Synchronization {
2:   abstract pointcut syncP(Object obj);
3: }
```

The `syncP` *pointcut (line* `2`*) identifies synchronization join points and receives the object to be used by the chosen synchronizations technique. The following abstract aspect extends the* `Synchronization` *aspect and defines a synchronization approach similar to the Java* `synchronized` *modifier or block.*

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

Its `around` *advice uses the inherited pointcut (*`syncP`*) to synchronize any joint point (method 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 to be used by the synchronization policy. The abstract pointcut* `syncP` *should be defined in a concrete aspect of an application that uses this framework. 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 will be used by the synchronization policy, which is the semantics of the* `synchronized` *method modifier.*

As an alternative to this pessimistic approach, a third abstract aspect implements the use of the Concurrency Manager design pattern [6]. Another CC reusable aspect is responsible for implementing the Timestamp technique. The complete aspects definitions can be found at http://sergio.dsc.upe.br/compsac07 and more details at [5].

Using the concurrency control framework

In order to implement the concurrency control in a specific software we should define concrete aspects for the abstract ones in the framework to identify where and which kind of concurrency control must be used. If this step is being performed as part of a restructuring process, in the "Removing the concurrency control code" step we had documented where and which kind of concurrency control was used in the software. Otherwise, we

should apply the concurrency control implementation method [7] analysis to identify such places and which controls to be used.

In the Health Watcher application we used the framework to implement synchronization of some methods, the Concurrency Manager design pattern to avoid undesirable interferences, and the Timestamp technique to avoid object inconsistencies. In fact, we also developed an alternative timestamp implementation that optimizes database accesses. The complete aspects definitions can be found at http://sergio.dsc.upe.br/compsac07 *and more details at [5].*

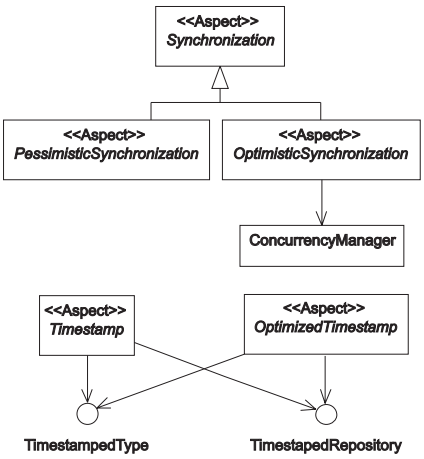

Figure 2. Concurrency Control Framework.

4.4. Concurrency Control Framework

Figure 2 depicts the Concurrency Control Framework in a UML class diagram. Note the use of the stereotype "Aspect" to identify aspects. All the aspects in the diagram are abstract.

The synchronization aspects can be easily reused in other applications that do not comply with the architecture we use, since they are very simple and not application or architecture-dependant. To use them it is only necessary to identify which methods should be synchronized and which approach to use (optimistic or pessimistic). The timestamp aspects could also be used for other architectures with some modifications, but they are harder to reuse since they use information about architecture's data management organization.

5. Conclusion

In this paper we presented guidelines to improve the correctness and performance of concurrent systems through aspect-oriented programming (AOP). The idea is separating concurrency control (CC) from other concerns, such as business rules, communication (distribution), data management, and user interface. This way the CC code will not be tangled with other concerns code. This strategy reduces the system complexity by improving its modularity. Thus, programmers need not to think about CC policies when implementing business, distribution, data management, and user interface.

Some aspects defined here are abstract and constitute a simple aspect framework that can be used to implement concurrency control in other applications, complying with the software architecture adopted in this work. In fact, some of them (synchronization aspects) can be reused for applications with different architectures. Those aspects are quite simple and concise if compared with the object-oriented solution. The others are application specific but different implementations might follow the same aspect pattern, which allows automatic aspects generation. We are currently working in a code generation tool to support a more general implementation method that considers, besides concurrency control, data management and distribution aspects.

We defined guidelines to restructure an object-oriented to an aspect-oriented concurrency control. Despite being originally defined to restructuring existing object-oriented software, the guidelines can also be used during aspect-oriented software development, when the system is implemented from scratch with aspects. The guidelines are also tailored to a specific software architecture, which allows more precise definitions and provides better support to programmers. The software architecture adopted in this paper has been used to implement several kinds and families of software.

Finally, as a future work, experiments will be performed to evaluate the proposed guidelines. The experiments will evaluate metrics like, reusability, and modularization level, identifying the benefits and drawbacks of the defined aspect framework, and the proposed guidelines.

References

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

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

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

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

[5] *S. Soares.* An Aspect-Oriented Implementation Method*. PhD thesis, Informatics Center, Federal University of Pernambuco — CIn-UFPE — Brazil, October 2004.*

[6] *S. Soares and P. 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.*

[7] *S. Soares and P. 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.*

[8] *S. Soares, E. Laureano, and P. 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, Seattle, WA, USA, November 2002. ACM Press. ACM SIGPLAN Notices 37(11).*