Implementing Modular and Reusable Aspect-Oriented Concurrency Control with AspectJ

Sérgio Soares[12*], **Paulo Borba**[13]

[1] SPG — Software Productivity Group

[2]Computing Systems Department — Pernambuco State University
Rua Benfica, 455, Madalena – 50720-001 — Recife, PE, Brazil

[3]Informatics Center — Federal University of Pernambuco
Caixa Postal 7851, 50740-540 — Recife, PE, Brazil

sergio@dsc.upe.br, phmb@cin.ufpe.br

Abstract. *The advent of information systems based on the World Wide Web increased the impact of concurrent programs. Such increase demands the definition of methods for obtaining safe and efficient implementations of concurrent programs, since the complexity of implementation and tests in concurrent environments is higher than in sequential environments. This paper presents a simple framework that helps to modularize and, therefore, reuse concurrency control concern using AspectJ, an aspect-oriented programming language. This framework was derived from the definition of a concurrency control implementation method that guarantees system correctness without redundant concurrency control, increasing performance and guaranteeing safety. It defines abstract aspects that can be extended to implement concurrency control in several applications. The achieved modularization makes the concurrency control easy to evolve and decreases the complexity of other parts of the software, such as business and data management, by decoupling concurrency control code from them.*

1. Introduction

The advent of information systems based on the World Wide Web increased the impact of concurrent programs. In fact, concurrent environments increase the complexity of the implementation and test activities, since subtle implementation errors lead to incorrect program executions that may be difficult to detect. This scene indicates that we need more adequate ways to implement concurrent programs, such as using a method definition [15, 12] to support concurrency control, avoiding controls based programmer's intuition.

In this paper we use AspectJ [7], a general purpose aspect-oriented [3] extension to Java [6], to implement a simple aspect framework derived from previous work [15, 12] on concurrency control on object-oriented languages. Such abstract aspects were derived from the aspect-oriented concurrency control method definition [13]. In fact, concurrency control is a good example of crosscutting concern that is hard to modularize using object-oriented programming and design patterns [11, 5, 1].

The concurrency control method definition was made with a simple but real and non trivial web-based information system, a health complaint system, which was originally implemented in Java and restructured to use AspectJ. The Health Watcher, the information system we used, was 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

*Work done when the author was at the Informatics Center, Federal University of Pernambuco.

food shops, health care institutions can promptly investigate the complaints and take the required actions. The system has a web-based user interface for registering complaints and performing several other associated operations.

This paper is structured in five sections. An overview about AspectJ, the aspect-oriented language we use, is presented in Section 2.. After that, in Section 3., we present the aspects framework for aspect-oriented concurrency control. Finally, Section 4. presents related work and Section 5. the conclusions.

2. AspectJ overview

AspectJ [7] is a general purpose aspect-oriented extension to Java. The aspect-oriented constructs support the separate definition of crosscutting concerns — concerns that affect several units of a system. This separation of concerns allows better modularity, avoiding tangled code and code spread over several units. Therefore, system maintainability is also increased.

Programming with AspectJ uses both objects and aspects to separate concerns. Concerns that are well modeled as objects are separated that way; concerns that crosscut the objects are separated in 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 [7] language is an *aspect*. Each aspect defines a functionality that crosscuts others, called crosscutting concerns, in a system. An aspect can declare attributes and methods, and may extend another aspect by defining concrete behavior for some abstract declarations. An aspect may also affect the static structure of Java programs by using AspectJ's static crosscutting mechanism. This mechanism allows one to introduce new methods and fields to an existing class, convert checked exceptions into unchecked, and change the class hierarchy by making an existing class extend another one.

Aspects affect the dynamic structure of a program by changing the way a program executes. An aspect can intercept certain points, called *join points*, of the program execution flow and add behavior *before*, *after*, or *around* the join point. Examples of join points are method calls, method executions, constructor executions, field references (get and set), exception handling, static initializations, and combinations of these using the logical `!`, `&&`, and `||` operators[sdfsd1]. Usually, an aspect declares *pointcuts* that select sets of join points and context values at those join points. The aspect also declares *advice* in order to specify the piece of code that should be executed when a pointcut is reached during execution. The advice declaration indicates whether the code should execute before, after, or around the join points specified by the pointcut.

3. A framework for aspect-oriented concurrency control

In this section, we present a simple aspect framework we derived when restructuring an object-oriented software. The restructuring relied on removing the concurrency control code that was tangled and spread over software units and implementing aspects to modularize this concurrency control. These steps are actually defined as guidelines to restructuring the object-oriented concurrency control, and, in fact, by ignoring the steps to remove the concurrency control from the object-oriented software, the restructuring guidelines are simplified to implementation guidelines to be used during aspect-oriented software development [13].

The following presents general concurrency control techniques that can be applied in object-oriented Java applications:

- `synchronized` Java modify [9, 6] — synchronized methods of an object cannot be concurrently executed;

- `synchronized` Java block [9, 6] — 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 [9, 6] — used to implement monitors behavior;
- use of the Concurrency Manager design pattern [14] — 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 [15] — avoids undesirable update of objects copies that might lead the system to an inconsistent state.

Since `wait`, `notify`, and `notifyAll` methods were not used by the concurrency control implementation method that derived the framework, we implemented aspects to the other four types of concurrency control as follows.

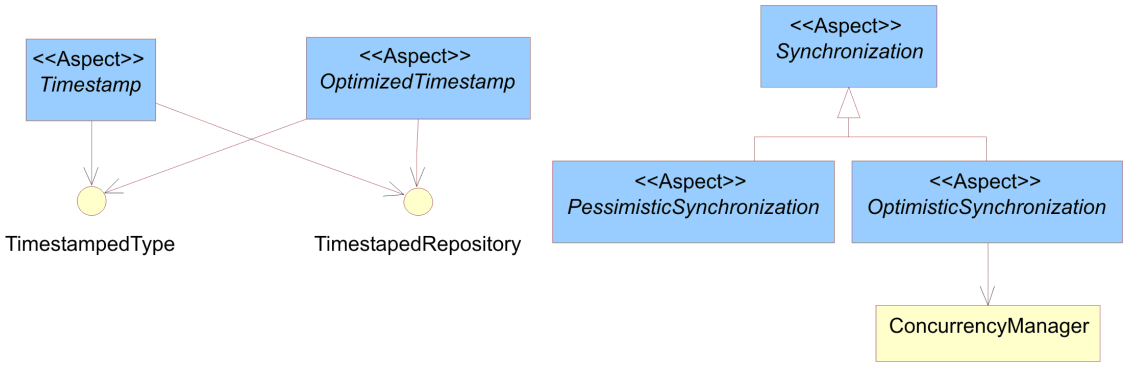

Figure 1. Concurrency Control Framework.

Figure 1 depicts the Concurrency Control Framework we present in the following sections 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 used in other architectures. It is only necessary to identify which methods should be synchronized and which approach to use. Besides providing reusable code and behavior, the aspects also guide the implementation of the concrete ones that will implement the abstract aspects.

3.1. Implementing reusable synchronization aspects

The first abstract aspect of the framework identifies join points that cannot execute concurrently, and therefore, should be synchronized, without specifying a specific concurrency control technique.

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

Besides identifying the synchronization join points, the `synchronizationPoints` pointcut also receives the object to be synchronized by the chosen synchronization technique, which is the `syncObj` parameter of the above pointcut.

A pessimistic alternative for synchronization aspects

The following abstract aspect extends the `Synchronization` aspect and defines a synchronization approach similar to the `synchronized` modifier or the `synchronized` 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 joint point, which should be a 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 whose lock should be used by the synchronization policy. This join point 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 the `this` designator), whose lock will be used by the synchronization policy, which is the semantics of the `synchronized` method modifier [9, 6].

An optimistic alternative for synchronization aspects

As an optimistic alternative to the previous approach, another abstract aspect implements the use of the Concurrency Manager design pattern. This design pattern provides an alternative to method synchronization with the aim of increasing system performance. It uses knowledge about the semantics of the methods in order to block only conflicting execution flows, allowing the non-conflicting ones to execute concurrently.

```
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 `synchronizationPoints` 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 two advices to provide the concurrency manager behavior.

```
  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 possible conflicting execution finishes, which is notified by the `after` advice. The `getKey` method is abstract and should be implemented along with the `synchronizationPoints` pointcut defining the semantics of the synchronization.

3.2. Implementing a reusable timestamp aspect

The last concurrency control reusable aspect is responsible for implementing the Timestamp technique. In fact, 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 update their copies. In addition, a timestamp field should be added to the classes of the objects that cannot be concurrently updated; an object can be updated only if there is not a newer copy of it stored in the database. The data management 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 on this technique can be found elsewhere [15, 12].

The abstract aspect for the timestamp technique defines two auxiliary interfaces to identify classes that will be affected by the aspects. Those are classes from the objects being updated (basic classes), and classes that manage data access (data collection classes).

```
abstract aspect Timestamp {
  interface TimestampedRepository {
    void updateTimestamp(TimestampedType object);
    long searchTimestamp(TimestampedType object);
  }
  interface TimestampedType { long getTimestamp(); }
```

The aspect also declares a constant to be used in concurrency exceptions and uses the inter-type declaration mechanism to add a `timestamp` field and methods responsible for managing the field (the set method is not shown but is similar to the get) in the subtypes of `Timestamped` interface.

```
  private static final String EXCEPTION_MESSAGE = ...;
  private long TimestampedType.timestamp;
  public long TimestampedType.getTimestamp() { return timestamp; }
```

A pointcut is defined to identify affected `search` methods in data collections, 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 `this` designator to expose the data collection, which is the currently executing object.

```
  pointcut managedSearchMethods(TimestampedRepository rep):
    execution(TimestampedType search(..)) && this(rep);
  after(TimestampedRepository rep) returning(TimestampedType obj):
      managedSearchMethods(rep) {
    long timestamp = rep.searchTimestamp(obj);
    obj.setTimestamp(timestamp);
  }
```

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

```
    pointcut managedInsertMethods(TimestampedRepository rep,
                                  TimestampedType obj):
      execution(void insert(TimestampedType)) && this(rep) && args(obj);
    after(TimestampedRepository rep, TimestampedType obj) returning:
        managedInsertMethods(rep, obj) {
      rep.updateTimestamp(obj);
    }
```

The `managedInsertMethods` pointcut and its 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 and an advice to update the timestamp information if the object is successfully updated.

```
    pointcut managedUpdateMethods(TimestampedRepository rep,
                                  TimestampedType obj):
      execution(void update(TimestampedType)) && this(rep) && args(obj);
    void around (TimestampedRepository rep, TimestampedType obj):
        managedUpdateMethods(rep, obj) {
      synchronized(rep) {
        long timestamp = rep.searchTimestamp(obj);
        if (obj.getTimestamp() == timestamp) {
          obj.setTimestamp(timestamp + 1);
          proceed(rep, obj);
          rep.updateTimestamp(obj);
        } else {
          Exception ex = new ConcurrencyControlException(EXCEPTION_MESSAGE);
          throw new SoftException(ex);
} } } }
```

Note the use of the `synchronized` block in the advice. This is needed in order to guarantee serialization, for each data collection (repository), during timestamp checking [15, 12].

If the object timestamp is different from the timestamp stored in the data collection a concurrency control exception is raised. 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 done by other aspects defined elsewhere [17].

4. Related Work

A related work implements transactions independently developed in the context of the OPTIMA framework for controlling concurrency and failures with transactions [8]. This implementation deals mostly with transactions for implementing concurrency concerns. 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 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 [4], 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/her 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.

"Concurrent Object Programming" [2] deals with separation of concurrency concerns using design patterns, pattern languages, and object-oriented framework. It covers several issues related to concurrent programming, including how to generate, control, and guarantee interaction between concurrent objects. On the other hand, our work's scope is limited to controlling concurrency in a specific software architecture, although the derived aspects can be reused by different software architectures.

This related 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 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. The concurrency control concerns we implemented do not have such problems, since there is no specific concurrency control in classes that are specialized in the specific software architecture used. The classes that are specialized in our work are basic classes, and the concurrency control applied to them is based in its fields' type. The control is applied if a single object can be concurrently accessed, or if it is possible that two copies of a same object be concurrently updated in the system, but 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 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.

"Concurrent Programming in Java" [9] proposes models for implementing concurrent Java programs, design patterns to guarantee a safety execution in concurrent environments, and some

rules to insert and to remove method synchronization. The approach in this work suggests 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 software maintainability. On the other hand, our approach allows implementing concurrency control separately (untangled), also separating the reasoning about concurrency control. Elsewhere [16, 13] we define a progressive approach where concurrency control is delayed until functional requirements validation, in order to reduce the impact caused by requirements changes during development. Another differential of our work is that it is based on a specific software architecture, which facilitates the definition of precise guidelines, also allowing a high automatization level.

Other related work is COOL [10], a domain specific language that express coordination of threads. COOL is a part of D [10], a domain specific language framework that uses an aspect-oriented approach in order to achieve separation of the distribution concerns from the basic software. The use of COOL to control concurrency introduces a new language the programmer must know. On the other hand, our work presents an aspect-oriented approach that uses a general-purpose language, AspectJ, and 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 [12, 15] 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.

5. Conclusions

Concurrent environments have a great complexity inserted by their non-determinism that can turn the system to an inconsistent state in abnormal interference. In this paper, we show that 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 easier to change concurrency control policies, since concurrency control code is not tangled with other concerns code. Therefore, besides making programs more modular, this separation also decreases their complexity, since there is no concurrency control to reason about when implementing business, data management and user interface.

We defined abstract aspects that constitute a simple aspect framework that can be extended to implement concurrency control in other applications. Those reusable aspects are quite simple and concise if compared to the object-oriented solution. These aspects were successfully used in a simple but real and non trivial web-based information system, the Health Watcher system. We are currently working into a code generation tool in order to support a implementation method that considers data management, distribution besides concurrency control aspects [13], and will guide the definition of the application specific aspects that will use the defined framework.

Moreover, the framework also validates the use of AspectJ with reuse purposes. Although simple, the abstract aspects provide all the functionality concerned with concurrency control, which is actually the goal of the aspects definition. Programmers that will use the framework have to define which classes and methods should be synchronized and to select the synchronization policy to be used. Regarding the timestamp technique, the programmers should provide more than just identifying points to be affected. Some programming effort is needed, but the abstract methods of the aspects in the framework help to guide such task.

6. Acknowledgments

We would like to thank the anonymous reviewers and the members of the SPG group, especially to Rohit Gheyi, Tiago Massoni, and Vander Alves that made important comments to improve this paper. We also thank CNPq and Capes, Brazilian research funding agencies, for partially supporting this work.

References

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

[2] 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.

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

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

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

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

[7] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. Getting Started with AspectJ. *CACM*, 44(10):59–65, October 2001.

[8] Jörg Kienzle and Rachid Guerraoui. AOP: Does it Make Sense? The Case of Concurrency and Failures. In *ECOOP'02*, LNCS 2374, pages 37–61, June 2002. Springer–Verlag.

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

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

[11] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern–Oriented Software Architecture, Vol. 2: Patterns for Concurrent and Networked Objects*. Wiley, 2000.

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

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

[14] Sérgio Soares and Paulo Borba. Concurrency Manager. In *First Latin American Conference on Pattern Languages of Programming*, pages 221–231, Rio de Janeiro, October 2001.

[15] 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.

[16] Sérgio Soares and Paulo Borba. PIP: Progressive Implementation Pattern. In *Proceedings of the 1st Workshop on Software Development Patterns (SDPP'02)*, November 2002.

[17] Sérgio Soares, Eduardo Laureano, and Paulo Borba. Implementing Distribution and Persistence Aspects with AspectJ. In *Proceedings of the 17th Conference on Object-oriented programming, systems, languages, and applications*, pages 174–190, November 2002. ACM Press.