Using Programming Laws to Modularize Concurrency in a Replicated Database Application

Leonardo Cole[1*] Paulo Borba[1†]

[1]Informatics Center
Federal University of Pernambuco
P.O. Box 7851 - 50.732-970 Recife, PE, Brazil

email: lcn@cin.ufpe.br, phmb@cin.ufpe.br

Abstract

As the use of AspectJ grows, it seems appropriate to define refactorings for that language. Most of the already defined aspect-oriented refactorings do not have a formal basis to ensure it preserves behaviour. This paper shows the application of programming laws and derived refactorings to separate a crosscutting concern from a system's core logic using AspectJ. We used programming laws to increase the confidence that the refactorings preserve behaviour.

1. Introduction

Refactoring [4, 10] has been quite useful for restructuring object-oriented applications. It can also bring similar benefits to aspect-oriented applications [3], increasing legibility, extensibility and maintainability. Moreover, refactoring might be a useful technique for introducing aspects to an existing object-oriented application.

In order to explore the benefits of refactoring, aspect-oriented developers are identifying common transformations for aspect-oriented programs [9, 5, 7], mostly in AspectJ[8], a general purpose aspect-oriented extension to Java. However, they lack a formal support for assuring that the transformations preserve behaviour and are indeed refactorings.

In this paper, we use aspect-oriented programming laws [6] to modularize a crosscutting concern in a commercial application with some confidence that the transformation preserves behaviour. Laws are much simpler than most refactorings because they involve only local changes, and each one focus on a single AspectJ construct. It is easier to show that this kind of transformation preserves behaviour. Therefore, the laws are useful to derive behaviour preserving transformations (refactorings) for this language. Also, the laws establish how to introduce or remove AspectJ constructs, and how to restructure AspectJ applications. We apply AspectJ programming laws similar to the laws defined for ROOL [1], an object-oriented language. We also show that AspectJ is useful as a solution to modularize concurrency in a real application and the process to introduce aspects to an existing object-oriented application.

This paper is organized as follows. Section 2 gives an overview of the aspect-oriented programming laws. Next, Section 3 overviews the application used, the Mobile Server. Following, Section 4 describes the sequence of refactorings applied to restructure the application. Section 5 then compares the original and the refactored versions, and conclude.

[*]Supported by CAPES.
[†]Partially supported by CNPq, grant 521994/96-9.

2. Laws and Refactorings

Sometimes, modifications required by refactorings are difficult to understand as they might perform global code changes. We use programming laws [2] to increase the confidence that an AspectJ transformation preserves behaviour. We rely on the simplicity of the laws, which involve only local changes and deal with one AspectJ construct each.

In this section we describe a simple law, showing its intent, structure, and preconditions. The laws establish the equivalence of AspectJ programs given that some restrictions are respected. Therefore, the structure of each law consists of three parts: left-side, right-side and preconditions. The first two are templates of the equivalent programs. The third part indicates conditions that must hold to ensure the equivalence is valid. For example, the following law is useful to extract code from the beginning of a method into an aspect. If the extracted code is spread through several methods, we would apply the law several times to isolate this code. Afterwards, we would use another law to merge the resulting advices, increasing reuse.

Law . Add Before-Execution

```
tds
class C {
  fds
  mds
  T m(pds) {
     body';
     body
  }
}
paspect A {
  pcds
  ads
}
```

=

```
tds
class C {
  fds
  mds
  T m(pds) {
     body
  }
}
paspect A {
  pcds
  ads
  before(C c, pds) :
     execution(T C.m(Γ(pds)))
     && this(c) && args(pds){
   body'[c/this]
  }
}
```

provided

(→) *body*′ does not declare or use local variables; *body*′ does not call `return` and `super`; *body*′ only access *C* members through `this`; variable *c* is not used in *body*′

The laws basically represent two transformations, one applying the law from left to right and another one in the opposite direction. Each law have preconditions to ensure that the program is valid after the transformation. When applied from left to right, this law moves part of a method's body into an advice that is triggered before method execution.

Inside advices, we can access variables in the context of the captured join point. The law always expose the maximum context available, in this case, the executing object (`this`(*c*)) and the method parameters (`args`(*pds*)). We denote the set of class and aspect declarations by *tds*, and set of field declarations and method declarations by *fds* and *mds*, respectively. We omit visibility modifiers, `throws` clauses and inheritance constructs for simplicity. However, there are similar laws that include the variations of visibility modifiers, exceptions and inheritance constructs [2].

As we move *body*′ to the aspect, its visible context changes. Hence, it is necessary to constrain the context dependencies in order to guarantee that the law relates valid AspectJ programs. Therefore, we impose conditions on accessing private members, local variables and calls to `super`. While the last two are forbidden, access to private members is allowed if done through `this`. This is necessary to enable the mapping of accesses to the object referenced by `this`, to the object exposed as the executing object on the advice (c). The mapping is denoted by the expression $body'[c/\texttt{this}]$, where we substitute all occurrences of `this` for the variable c in $body'$.

However, there are other implications that must be considered. Changes to the method execution flow (calls to `return`) are generally not allowed because the advice cannot implement it, or it would increase complexity. This precondition is necessary to ensure that the law preserves behaviour. Other laws are similarly defined in terms of transformations and preconditions, and establish properties of other constructs besides before. Table 2 shows a summary of the laws.

Table 1: Summary of laws

Law	Name	Law	Name
1	Add empty aspect	16	Remove argument parameter
2	Make aspect privileged	17	Add catch softened exception
3	Add before-execution	18	Soften exception
4	Add before-call	19	Remove exception from throws clause
5	Add after-execution	20	Remove exception handling
6	Add after-call	21	Move exception handling to aspect
7	Add after-execution returning successfully	22	Move field to aspect
8	Add after-call returning successfully	23	Move method to aspect
9	Add after-execution throwing exceptions	24	Move implements declaration to aspect
10	Add after-call throwing exceptions	25	Move extends declaration to aspect
11	Add around-execution	26	Extract named pointcut
12	Add around-call	27	Use named pointcut
13	Merge advices	28	Move field introduction up to interface
14	Remove `this` parameter	29	Move method introduction up to interface
15	Remove `target` parameter	30	Remove method implementation

3. The Mobile Server

The Mobile Server is a commercial application that provides replication and synchronization of data that might be used off-line in different platforms (including mobile devices). It keeps information regarding changes made by users on each platform, solves conflicts with modifications made elsewhere and then propagates the resulting changes to all replicas.

In this system, one important part is the *Concurrency Manager*, which is responsible for coordination of data repository (a database with useful information) accesses. Thus its services are used by several modules, decreasing code legibility and making maintenance and extension harder. Figure 1 shows the components of the Mobile Server. The ones that access the repository need concurrency control.

A copy of the database is available in each platform allowing users to access the system off-line. As a result, the system needs to provide synchronization mechanisms between the central database and its local copies. There are two processes that carry out this responsibility. The first one is the *Input Processor*, which analyzes changes made on each local database and incorporates these changes in the centralized database. The second one is the *Output Processor*, which analyzes the database to collect changes that will be applied to the local databases. Those two processes respectively consume and produce files that are used by the *Synchronization*

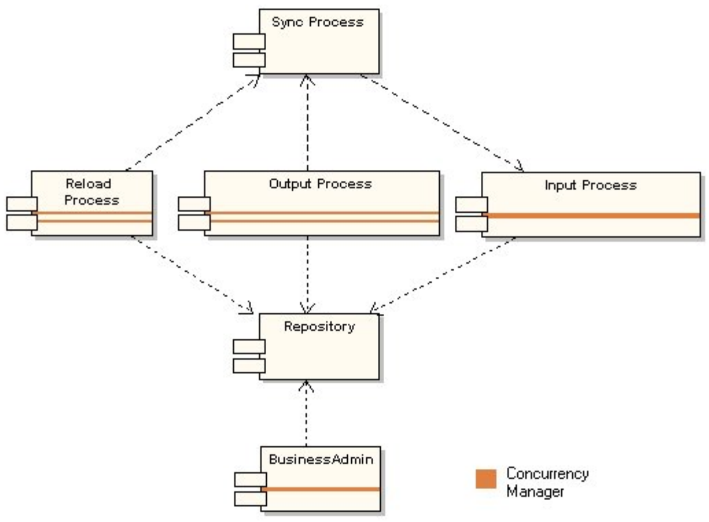

Figure 1: Mobile Server Before Refactoring.

process, which is responsible to receive local changes from the device and send global changes from the system. The *Business Admin* process configures the database tables. The *Reload* process is used only in case a local database is lost (new device or device crash). In this case, it sends the complete database copy to the device.

4. Restructuring The Mobile Server

This case study focus on separating the code related to the *Concurrency Manager* (CM) from all the other parts of the system, using aspects to provide the separation that could not be adequately achieved with object-oriented techniques. Once the crosscutting concern is identified, our strategy consists in two steps. First, we use object-oriented refactorings [4] for restructuring the code to enable the laws application (satisfy preconditions). Second, we apply a sequence of laws (refactoring). We start by removing the concurrency control from the *Output* and *Reload* processes. The following method `process` is part of both processes.

```
void process() {
  CM.beginExecution(this.id,this.tableNames,this.user);
  CM.sort(this.tableNames);
  for (int i=0; i<this.tableNames.length;i++) {
    CM.getNextLock(this.id,this.user);
    // different logic for each process;
    CM.releaseTable(this.id,this.getTableNames()[i],this.user);
  }
  CM.endExecution(user,user);
}
```

At first, we use object-oriented refactorings. We apply *Extract Method* to eliminate the use of local variables, and *Encapsulate Field* to ensure that the fields (`id`, `tableNames` and `user`) are used by its accessor methods. At this point we apply *Extract Method* once again, to provide the required join points to be used by the advices. It was necessary to extract the `processTable` method, which contains the logic applied to a single table. Now this new method is the only difference between the processes. So we applied *Pull Up Method* on two methods (both called `process`), isolating the concurrency control on an abstract super class. The resulting code is shown next.

```
public abstract class APThread {
  void process() {
    CM.beginExecution(this.getID(),this.getTableNames(),this.getUser());
    CM.sort(this.getTableNames());
    for (int i=0; i<this.getTableNames().length;i++) {
      CM.getNextLock(this.getID(),this.getUser());
      this.processTable(this.getTableNames()[i]);
      CM.releaseTable(this.getID(),this.getTableNames()[i],
                      this.getUser());
    }
    CM.endExecution(this.getID(),this.getUser());
  }
}
```

The process starts indicating to the CM the tables that will be required (`beginExecution`), the manager provides the order in which the process can use the tables (`sort`), finally the process waits for its turn to use each table (`getNextLock`). For each used table, the process notifies the manager to release it (`releaseTable`) and, after processing, the manager is notified to release all the remaining tables (`endExecution`). All those calls to CM are tangled with the process business code. Our goal is to modularize that code.

Once the system is restructured, we can start applying the laws. We use Laws 1 and 2 to create an aspect and make it privileged. Then we apply Law 11 to create a new advice around the `process` method execution, moving the calls to the CM (`beginExecution` and `endExecution`) to the aspect. Next we apply Law 12 to introduce a new advice around a call to the extracted method `processTable`, moving the remaining calls to the manager.

```
public abstract class APThread {
  public void process() {
    for (int i=0; i<getTableNames().length;i++) {
      processTable(this.getTableNames()[i]);
    }
  }
}
```

The above code shows the resulting method without the concurrency control. The following code shows the aspect that is responsible to making the call to CM when necessary. We applied Law 15 to the second advice in order to remove the `target` parameter since this parameter was not used. We can now move the methods that are used only by the concurrency control code using Law 23. The only method moved was `getID`, which returns a constant of the CM class. Then we can start restructuring the aspect. To that matter, we use the *Extract Pointcut* refactoring that creates named pointcuts from the advice expressions and makes the advices refer to these pointcuts. This basically finish refactoring the *Output* and *Reload* processes.

```
privileged aspect CMAspect {
  void around(APThread c): execution(void APThread.process()) &&
                       this(c){
    CM.beginExecution(c.getID(),c.getTableNames(),c.getUser());
    CM.sort(c.getTableNames());
    proceed(c);
    CM.endExecution(c.getID(),c.getUser());
  }
  void around(APThread c, String table):
                       call(void APThread.processTable(String)) &&
                       this(c) && target(APThread) && args(table){
    CM.getNextLock(c.getID(),c.getUser());
    proceed(c, table);
    CM.releaseTable(c.getID(),table,c.getUser());
  }
}
```

The next process analyzed was the *Business Admin* process. This process is responsible for configuring the database managing the replicated tables. We started preparing the code to be refactored as we did before. In this case, we used *Replace Temp with Query* and *Extract Method* to eliminate local variables.

As this process uses only one table for each operation, it does not have a loop similar to the one showed in the previous process. Thus, we need only one advice (Law 11) that is responsible to make all the necessary calls to CM. The rest of the refactoring was exactly the same showed to the output and reload process.

The last affected module is the Input process which is responsible for processing the information received from the devices, solving conflicts and propagating this information to the centralized database. We used object-oriented refactorings to remove local variables and to provide the necessary join points to be used by the aspects. It was also necessary to change the way the CM was accessed. It was originally accessed through a field. However, it can be directly accessed (through static methods), without the need for a field.

```
public class IPThread {
  public void process(..) {
    CM.beginExec(..);
    CM.sort(..);
    try {
      \\(for loop similar to other cases)
    } finally {
      CM.endExec(..);
    }
  }
}
```

The prepared code ended with a structure slightly different from the other processes shown above. The notification of the end of execution appears inside a `finally` clause. This happens because the processing exceptions were not handled inside the method affected by the concurrency control. Hence, we cannot use an around advice as we did before, we must use before and after advices. The first notifies the beginning of the process and the second notifies its end. So, we use Law 3 to introduce the before advice and Law 5 to introduce the after advice. The remainder of the refactoring is identical to the refactoring of the *Output* and *Reload* processes.

Now that we have refactored out all the concurrency control code, there is still one last issue: exception handling. Therefore, we use the *Extract Exception Handling* refactoring, which moves exception handling code to an aspect. This refactoring is achieved from the sequence of laws shown in Figure 2. We used this refactoring on the exceptions related to the concurrency control. The resulting system is showed in Figure 3

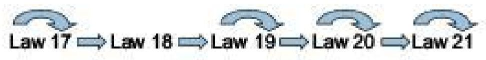

Figure 2: Extract Exception Handling Refactoring.

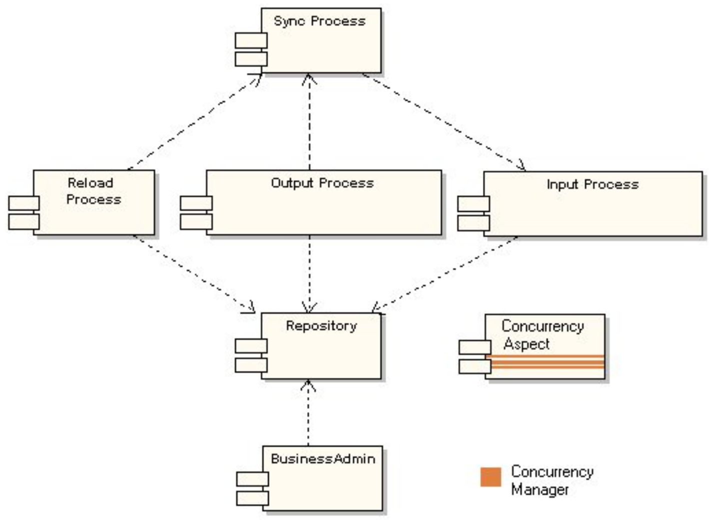

Figure 3: Mobile Server After Refactoring.

5. Evaluation and Conclusion

Using the laws, we have a good start point to ensure the behaviour preserving property of the entire refactoring. Moreover, laws are simpler than most refactorings because they involve localized changes, and each one focus on one AspectJ construct. However, the laws still do not guarantee correctness. To ensure correctness it is necessary to define a formal semantics and prove that the two sides of a transformation are semantically equivalent. Also, the preconditions indicate code properties to ensure that the transformation preserves behaviour. Consequently, the laws helped to identify object-oriented refactorings applied to prepare the code.

Thus, we need another way to verify if the behaviour was preserved. Therefore, we built a test suite to exercise the concurrent actions on the *Repository*. A good test suit still does not guarantee correctnes. However, tests are considered a good practice to verify if a refactoring preserves behaviour [4]. Our test suite exercises the *Repository*, creating situations where concurrency problems would arise. The execution of the test suite did not reveal any error. The system state was always coherent during the test execution and the operations were performed as expected.

We also monitored the system performance during the execution of the test suite before and after the restructuring. The refactored version showed a decrease in performance. However, the performance bottleneck is the access to the database. Thus, this difference was not relevant.

There were also benefits yielded by the use of AOP. We showed that the business code separated from the crosscutting concerns is cleaner and more legible, increasing systems maintainability. Moreover, the aspects increased the systems modularity since the scattered code is now localized inside aspects. The new implementation also reduced the lines of code, due to the fact that the aspects have advices controlling several different join points. The code on the advices was repeated in every captured point. This reduction would be more visible in cases where advices capture more join points. Unfortunately, we could not use the same advice for several join points in the Mobile Server example.

We refactored a commercial application to extract the concurrency crosscutting concern with AspectJ. We used laws of programming and refactorings derived from them [2] to support the restructuring process. The laws provide some confidence that the refactorings applied indeed preserve behaviour. Besides, we showed that AspectJ is useful to modularize a crosscutting concern in a real application, and also the process to migrate from an object-oriented application an aspect-oriented one.

References

[1] P. H. M. Borba, A. C. A. Sampaio, A. L. C. Cavalcanti, and M. L. Cornelio. Algebraic reasoning for object-oriented programming. *Science of Computer Programming*, January 2004.

[2] L. Cole and P. H. M. Borba. Deriving refactorings for AspectJ. In *4th International Conference on Aspect-Oriented Software Development*. (submited).

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

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

[5] S. Hanenberg, C. Oberschulte, and R. Unland. Refactoring of aspect-oriented software. In *4th Annual International Conference on Object-Oriented and Internet-based Technologies,Concepts, and Applications for a Networked World (Net.ObjectDays)*, pages 19–35, Erfurt, Germany, Sept. 2003.

[6] C. A. R. Hoare, I. J. Hayes, H. Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sorensen, J. M. Spivey, and B. A. Sufrin. Laws of programming. *Commun. ACM*, 30(8):672–686, 1987.

[7] M. Iwamoto and J. Zhao. Refactoring aspect-oriented programs. In F. Akkawi, O. Aldawud, G. Booch, S. Clarke, J. Gray, B. Harrison, M. Kandé, D. Stein, P. Tarr, and A. Zakaria, editors, *The 4th AOSD Modeling With UML Workshop*, 2003.

[8] 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.

[9] R. Laddad. Aspect-Oriented Refactoring Series. TheServerSide.com, Dec. 2003.

[10] W. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, Urbana-Champaign, IL, USA, 1992.