Deriving Refactorings for AspectJ

Leonardo Cole[*]
lcn@cin.ufpe.br

Paulo Borba[†]
phmb@cin.ufpe.br

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

ABSTRACT

In this paper we present aspect-oriented programming laws that are useful for deriving refactorings for AspectJ. The laws help developers to verify if the transformations they define preserve behaviour. We illustrate that by deriving several AspectJ refactorings. We also show that our laws are useful for restructuring two Java applications with the aim of using aspects to modularize common crosscutting concerns.

Categories and Subject Descriptors

D.1 [**Software**]: Programming Techniques—*Aspect-Oriented Programming*; D.3.2 [**Programming Languages**]: Language Classifications—*AspectJ*

General Terms

Languages

Keywords

refactoring, AspectJ, Aspect-Oriented Programming, separation of concerns

1. INTRODUCTION

Refactoring [7, 22, 23] has been quite useful for restructuring object-oriented [20, 2] applications. It can bring similar benefits to aspect-oriented [6] applications. 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 [21, 17, 9, 14], mostly in AspectJ [15], a general purpose aspect-oriented extension to

---

[*]Supported by CAPES.

[†]Partially supported by CNPq.

Java. However, they lack support for assuring that the transformations preserve behaviour and are indeed refactorings.

This paper focus on that problem and introduces AspectJ programming laws that can be used to derive or create behaviour preserving transformations (refactorings) for a subset of this language. Programming laws [13] define equivalence between two programs, given that some conditions are respected. Our set of laws not only establishes how to introduce or remove AspectJ constructs, but also how to restructure AspectJ applications. By applying and composing those laws, one can show that an AspectJ transformation is a refactoring. The laws are suitable for that because they are much simpler than most refactorings. Contrasting with refactorings, they involve only localized program changes, and each one focus on a specific language construct. The laws constitute a basis for defining refactorings with some confidence that they preserve behaviour.

We evaluate our laws by showing how they can be used to derive several refactorings proposed in the literature [17, 9, 14]. This helps to more precisely specify the preconditions and code changes associated with those refactorings, and gives more confidence that they preserve behaviour.

Besides deriving refactorings, we evaluate our laws by restructuring two Java applications. The implementation of crosscutting concerns in those applications are restructured so that they are modularized with AspectJ constructs. This illustrates that the laws might also be useful for transforming Java applications into AspectJ applications.

The remainder of this paper is organized as follows. Section 2 discusses the AspectJ subset used here. Next, Section 3 introduces some of our laws showing their structure, preconditions and intent. Section 4 then discusses the derivation of several refactorings from our laws, focusing on the derivation steps. Section 5 shows the refactoring of two case studies using our laws and the derived refactorings. Then, we discuss related work in Section 6 and conclude in Section 7.

2. ASPECTJ SUBSET

AspectJ [15] is an aspect-oriented [6] extension to Java. Aspect-oriented languages support the modular definition of concerns which are generally spread throughout the system and tangled with core features. This separation of concerns promotes the construction of a modular system, avoiding code tangling and scattering.

In this paper, we consider a subset of AspectJ. This simplifies the definition of transformations and does not compromise our results. First, our language does not have packages, and the use of `this` to access class members is mandatory. Also, the `return` statement can appear at most once inside a method body and has to be the last command. Second, we restrict the aspect-oriented constructs, not considering abstract aspects and supporting only the pointcut designators `call`, `execution`, `args`, `this` and `target`.

Restricting the use of `this` simplifies the preconditions defined for the laws. This can be seen as a global precondition instead of a restriction to the language. Most of the laws dealing with advices require this restriction.

The restrictions applied to the aspect-oriented constructs are limitations to our set of laws. We do not cover transformations involving creation and maintenance of abstract aspects. Also, we only support the mentioned pointcut designators. The language extension to include other pointcut designators would be time demanding but not difficult. Besides, it would not affect the already defined laws. For instance, it would be necessary do define new laws to deal with abstract aspects. This is regarded as future work.

3. LAWS

Sometimes, modifications required by refactorings are difficult to understand as they might perform global changes in an application. We use laws of programming [13] to show that an AspectJ transformation is a refactoring. A refactoring denotes a behaviour preserving transformation that increases code quality. Contrasting with a refactoring, a law is bi-directional and it does not always increase code quality, it is part of a bigger strategy that does. Besides, our laws are much simpler than most refactorings because they involve only localized changes, and each one focus on one specific AspectJ construct.

In this section we describe a subset of our laws, showing their intent, structure, and preconditions. Our laws establish the equivalence of AspectJ programs provided 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 between the programs.

For example, the following law has the purpose of introducing or removing the `privileged`[1] modifier, which indicates that the aspect can access private members of classes. Most of our laws assume that the aspect has access to private members of classes. It enables us to relax conditions in order to transform the code. However, we can always use this law to remove the privilege in situations where the code do not access private members. We denote the set of type declarations (classes and aspects) by *ts*. Also, *pcs* and *as* denote pointcut declarations and advice declarations, respectively.

Law 2. Make Aspect Privileged

```
ts                    ts
aspect A {            paspect A {
  pcs          =        pcs
  as                    as
}                     }
```

---
[1]We abstract the declaration `'privileged aspect'` as `paspect` for simplicity

provided

($\leftarrow$) *as* does not refer to private classes, aspects, methods and fields in *ts*

Our laws basically represents two transformations, one applying the law from left to right and another in the opposite direction. Each law provides preconditions to ensure that the program is valid after the transformation. Another use of the preconditions is to guarantee that the law preserves behaviour. Some laws, when applied from right to left, correspond, roughly, to the transformations applied by the AspectJ compiler to join (weave) the classes and aspects.

We have different preconditions depending on the direction the law is used. This is represented by arrows, where the symbol ($\leftarrow$) indicates this precondition must hold when applying the law from right to left. Similarly, the symbol ($\rightarrow$) indicates that this precondition must hold when applying the law from left to right. Finally, the symbol ($\leftrightarrow$) indicates that the precondition must hold in both directions.

Revisiting Law 2, we see from the preconditions that we can always make an aspect `privileged`, since it only increases the scope of the code inside advices. It is important to note that the captured join points remain the same, as the pointcut expressions are not affected by the `privileged` modifier. Note that private methods can be captured by pointcuts even though the aspect is not privileged.

Eventually, we may realize that our aspect does not need access to private members of classes any more. Thus, we apply this law from right to left, removing the `privileged` modifier. However, it is necessary that the list of advices (*as*) does not refer to private members declared in *ts*.

The next law, when applied from left to right, moves part of a method's body into an advice that is triggered before method execution. Using this law, we can move the beginning of a method's body (*body'*) to an advice that runs before the method execution.

Law 3. Add Before-Execution

```
ts                         ts
class C {                  class C {
  fs                         fs
  ms                         ms
  T m(ps) {                  T m(ps) {
    body';                     body
    body               =     }
  }                        }
}                          paspect A {
paspect A {                  pcs
  pcs                        as
  as                         before(context) :
}                              exec(σ(C.m)) &&
                               bind(context) {
                                 body'[cthis/this]
                               }
                             }
                           }
```

provided

($\rightarrow$) *body'* does not declare or use local variables; *body'* does not call `super`;

($\leftarrow$) *body'* does not call `return`;

We use $\sigma(C.m)$ to denote the signature of method m of class C, including its return type and the list of formal parameters. Moreover, we use *context* to denote the

list of advice parameters, including the executing object (mapped to a parameter named *cthis*) and the method's parameters (*ps*). We use *bind(context)* to denote the expression of pointcut designators that bind the advice parameters (`this`, `target` and `args`). The laws always expose the maximum context available. For example, the previous law can expose the executing object and the formal parameters of the captured method. Considering a method `credit` in an `Account` class, the expanded advice signature looks like the code shown next. In this case, *context* is the parameter list (`Account cthis, float amount`) and *bind(context)* is the expression `this(cthis) && args(amount)`.

```
before(Account cthis, float amount) :
    execution(void Account.credit(float)) &&
    this(cthis) && args(amount)
```

We also denote the set of field declarations and method declarations by *fs* and *ms* respectively. We consider a simplified law where we omit visibility modifiers, throws clauses and inheritance constructs. However, we have similar laws that include the variations of those constructs in order to match different code templates. The `exec` expression denotes the `execution` pointcut designator.

As we move *body'* to an aspect, its visible context changes as well. 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 local variables and calls to `super` and `return`. Local variables can generally be removed using object-oriented programming laws [3]. The language restriction to obligate the use of `this` to access class members is important to enable the mapping of accesses to the object referenced by `this`, to the object exposed as the executing object on the advice (*cthis*). The mapping is denoted by the expression *body'[cthis/`this`]*, where we substitute all occurrences of *this* with the variable *cthis* in *body'*.

Nevertheless, 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.

This is the simplest law to introduce an advice. Our laws consider the `execution` and `call` designators, as well as five types of advices: `before`, `after`, `after returning`, `after throwing` and `around`. Thus, combining the pointcut designators and advices, we have a total of 10 laws for introducing advices[2]. Each of those laws uses different advice constructions, thereby requiring different method templates. The laws are summarized in Table 1. More details about the laws are available online [4].

The following law shows an advice using the pointcut designator `call`. The advices that use the `call` designator are slightly different from the ones using `execution`. The captured join point must appear inside a method's body and before a call to a second method. Moreover, there is a new parameter that can be exposed from the context, the `target` object. Hence, we expose both `this` and `target` objects, and the method's arguments. We use the `withincode` (denoted as `wc`) operator of AspectJ to restrict the calls to the captured method occurring only inside the originating method

---

[2]This number increases considering variations in visibility modifiers, `throws` clauses and inheritance constructs

---

(*n*). We also denote α preceding a list of parameters to represent the list its values.

Law 4. Add Before-Call

$$
\begin{array}{c}
\boxed{\begin{array}{l}
ts \\
\texttt{class } C \ \{ \\
\quad fs \\
\quad ms \\
\quad T \ n(ps') \ \{ \\
\quad\quad body; \\
\quad\quad exp.m(\alpha ps) \\
\quad \} \\
\} \\
\texttt{paspect } A \ \{ \\
\quad pcs \\
\quad as \\
\}
\end{array}}
\end{array}
=
\begin{array}{c}
\boxed{\begin{array}{l}
ts \\
\texttt{class } C \ \{ \\
\quad fs \\
\quad ms \\
\quad T \ n(ps') \ \{ \\
\quad\quad exp.m(\alpha ps) \\
\quad \} \\
\} \\
\texttt{paspect } A \ \{ \\
\quad pcs \\
\quad as \\
\quad \texttt{before}(context) : \\
\quad\quad \texttt{wc}(\sigma(C.n())) \ \&\& \\
\quad\quad \texttt{call}(\sigma(O.m())) \&\& \\
\quad\quad bind(context) \ \{ \\
\quad\quad\quad body[cthis/\texttt{this}] \\
\quad\quad \} \\
\}
\end{array}}
\end{array}
$$

provided

- ($\rightarrow$) *body* does not declare or use local variables; *body* does not call `super`;
- ($\leftarrow$) *body* does not call `return`;
- ($\leftrightarrow$) *O* is the type of *exp*;

Most of the preconditions to apply this law are similar to the preconditions of Law 3. However, some of the preconditions are different. We define a new precondition that must hold in both directions. This precondition states that the type of *exp* is *O*, assuring that the specification of the pointcut is correct. In a similar law considering more than one call to *m* inside *n*, there is an extra condition stating that every occurrence of this call must be preceded by *body*. Another variation in this law considers the existence of code after the call to method *m*. This law exposes an object of type *O*, as the `target`, in addition to the context exposed in Law 3.

Once an advice is in place, we need to simplify its structure, improving legibility. For this purpose we have Laws 13-16 and 26-27, providing a way to merge equal advices, remove some context exposure not used and, finally, create and use named pointcuts from the advice expressions. For instance, Law 13 is responsible for merging advices that execute the same action at different join points. Therefore, it enables us to have an advice capturing several join points. We did not focus on simplifying the resulting expressions, although it would be a valuable contribution. Simplifications would yield new expressions with wild cards for example. Law 13 has just one precondition that must hold from right to left. It ensures that both advice expressions must bind every exposed parameter in *ps*.

There are also other laws that help restructuring the advice in order to improve legibility. The next law (Law 15) is responsible for removing a `target` parameter of an advice provided that the parameter is not used in the advice body.

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

Law 13. Merge Before

```
ts
paspect A {
  pcs
  as
  before(ps) : exp1{
    body
  }
  before(ps) : exp2{
    body
  }
}
```
=
```
ts
paspect A {
  pcs
  as
  before(ps) :
      exp1 || exp2 {
    body
  }
}
```

provided

($\leftarrow$) $exp1$ and $exp2$ bind all parameters in ps.

Law 15. Remove Target Parameter

```
ts
paspect A {
  pcs
  as
  before(T t, ps) :
      target(t) &&
      exp {
    body
  }
}
```
=
```
ts
paspect A {
  pcs
  as
  before(ps) :
      target(T) &&
      exp {
    body
  }
}
```

provided

($\rightarrow$) t is not referenced from $body$

Although we can remove the `target` parameter from the context exposed by the advice, the binding designator (in this case the `target`) cannot be removed. Removing the target designator from the pointcut expression implies a generalization. This may cause the advice to capture more join points than before the transformation. Hence, the law only changes the `target` expression to use the object type instead of the parameter. This law also have similar versions for each kind of advice.

Another useful law is Law 18 which, together with Laws 17 and 19-21, allows the extraction of exception handling code into an aspect. This law is responsible to turn an exception raised by one join point into a soft exception. The other laws deal with `catch` and `throws` clauses to enable the complete extraction of the exception handling.

Law 18 uses the `declare soft` construct to soften the exception. At the same time, it removes the target exception (E) from the throws clause. The other exceptions raised by the method are denoted as exs. We omit the `throws` construct for simplicity. The precondition applied when the law is used from left to right is necessary to guarantee that the softened exception would still be handled. Thus, preventing a change in behaviour. Otherwise, the softened exception would bypass its original handling point. Likewise, the precondition when applying the law from right to left, guarantees that the code compiles and the exception is handled where necessary. The `SoftException` is an unchecked exception used by AspectJ, it wraps the softened exception.

Law 18. Soften Exception

```
ts
class C {
  fs
  ms
  T m(ps) E, exs {
    body
  }
}
paspect A {
  pcs
  as
}
```
=
```
ts
class C {
  fs
  ms
  T m(ps) exs {
    body
  }
}
paspect A {
  declare soft : E :
    exec(σ(C.m));
  pcs
  as
}
```
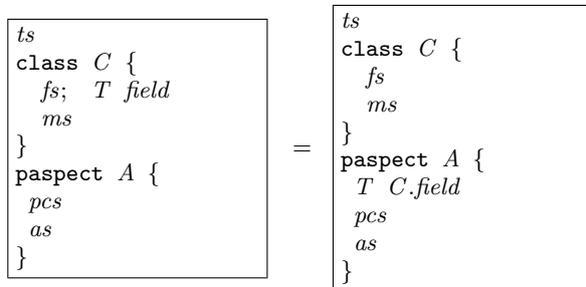
provided

($\rightarrow$) Every handler of E in ts and as have a catch clause for the `SoftException` and a case to handle E when it is the wrapped throwable;

($\leftarrow$) Every handler of `SoftException` containing a case to handle E when it is the wrapped throwable, is able to handle E itself; every call to method m of class C catches or throws E.

Although we do not discuss in detail the other laws related to exception handling, it is important to notice the reason why we have them. First, as we soften an exception as showed in Law 18, the result is that a new unchecked exception is raised instead of the existing one. Hence, every handler of the existing exception ceases to work and the exception would bypass its intended handling point and thus it would not preserve behaviour. It is necessary to copy the existing exception handling code so that the new unchecked exception is handled the same way. In fact, this is a precondition to apply Law 18. To introduce the `catch` for the unchecked exception we have Law 17.

Then, it is necessary to remove the exception from the throws clauses of methods that do not raise it any more. It seems to be an object-oriented refactoring but we provide Law 19 because we need its preconditions to derive aspect-oriented refactorings. Next, it is necessary to remove the `catch` blocks for the softened exception where the `try` body do not raises it anymore. To that intent we have Law 20. Finally, we have Law 21, which is responsible to move the handling of the unchecked exception to an aspect. Those laws are generally related by their preconditions, which imposes a certain order on their application. The composition of the laws as described is a complete refactoring to *Extract Exception Handling* code (see Section 4).

The next law, together with Laws 23-25 and 28-30 deals with inter-type declarations. This law is responsible for moving one field declaration to an aspect. This is necessary in cases where a class field is part of a crosscutting concern and has to be considered inside the aspect. We have to assure that all of its references had already been moved to the aspect before moving the field. This restriction is necessary for non-public fields because the semantics is not the same as simply declaring the field in the class. Visibility modifiers in inter-type declarations are relative to the aspect.

Law 22. Move Field to Aspect

provided

($\rightarrow$) The field *field* of class C does not appear in *ts* and *ms*.

The precondition of only the aspect referencing the moved field is rather strong. Depending on the field visibility, there are other elements which can refer to the field. However, our experience shows that even strong, this precondition is enough and covers all of the analyzed cases included on next two sections.

Sections 4 and 5 shows applications of our laws to derive refactorings and then to restructure two application. For brevity, we omit the direction that each law is used assuming that all laws are applied from left to right.

3.1 Soundness

We argued informally about the correctness of the laws. This is possible due to their simplicity; they involve only local changes and deal with properties of one AspectJ construct at a time. It is also possible to verify that some of the transformations preserve behaviour by checking that, when applied from right to left, the associated laws correspond to the transformations applied by the AspectJ compiler to weave classes and aspects. For instance, applying Laws 3, 4, and 7-10 from right to left is equivalent to the transformation performed by the AspectJ compiler when weaving `before` and `after` advices that capture a single `execution` or `call` join point.

However, soundness of the laws with respect to a formal semantics is a desirable property. It can give better confidence that the transformations preserve behaviour. So we are using the semantics of a toy aspect-oriented language [18] where we can represent part of our laws. We are considering both static and dynamic semantics, and exploring notions of semantic equivalence between aspect-oriented programs. Nevertheless, this work will have limitations, similar to formal approaches for object-oriented languages [3]. Those limitations are related to the use of mechanisms such as reflection, which breaks several refactorings.

4. DERIVING ASPECTJ REFACTORINGS

Several authors consider refactorings for aspect-oriented languages. Some of them [9, 14, 17] show refactorings to transform Java programs into AspectJ programs, yielding results related to ours. However, they focus on describing large and global refactorings. Here we show that some of those refactorings can be derived from our laws. This is important to evaluate the laws, and to show how they are useful. Our intent is not to define new refactorings, but to provide some basis so that refactorings can be defined with some confidence that they preserve behaviour. Also, once a refactoring is in place, a developer uses it directly and does not need to be aware of the laws.

Some of the refactorings in the literature are basic and thus their derivation is not represented as a sequential composition of our laws. Instead, we represent them as a single law chosen from a limited set. For instance, we can use one of the laws related to creating a new advice (Laws 3-12) to accomplish the *Extract Advice* [9] refactoring. Analogously, we use one of the Laws 22-23 to accomplish the *Extract Introduction* [9] refactoring. Those refactorings define different transformations for distinct kinds of advice or inter-type declarations.

As most of the refactorings considered here create a new aspect, we assume that all of them use Law 1 and Law 2 to create a new empty aspect and make it `privileged`. Hence, the subsequent derivations do not show the application of these two laws. We only formalize the *Extract Pointcut* refactoring because it is simple enough to provide a readable law. The other refactorings would be difficult to read and understand, therefore we explain them by means of examples.
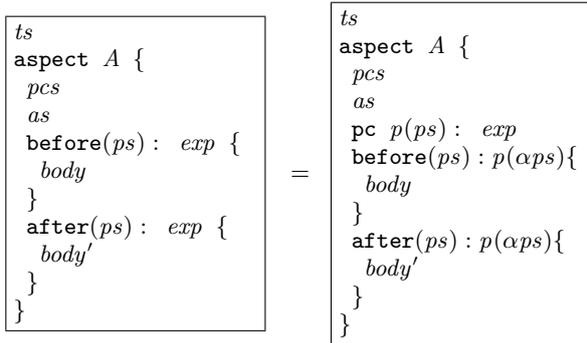
4.1 Extract Pointcut

The *Extract Pointcut* [14] refactoring is the first examined refactoring that does not deal with transformations from Java to AspectJ. This refactoring describes a transformation from AspectJ to AspectJ with the intent of increasing code

quality. Moreover, the formalization of this refactoring as a law, indicates its reverse refactoring as well. This happens because of the equivalence notion that allows the application of the refactoring on both directions. However, we only discuss its application from left to right.

This refactoring transforms an anonymous pointcut (exp) used by a number of advices into a named pointcut (p) and changes all advices that use the same anonymous pointcut to reference the new named one. Although the representation of the refactoring only deals with two advices, this refactoring can be extended to work with any number of advices. For simplicity, we use `pc` to denote the declaration of a named pointcut instead of the `pointcut` declaration of AspectJ.

REFACTORING 1. *Extract Pointcut*

$$
\begin{array}{ll}
\begin{array}{l}
ts \\
\texttt{aspect } A \ \{ \\
\quad pcs \\
\quad as \\
\quad \texttt{before}(ps): \ exp \ \{ \\
\quad\quad body \\
\quad \} \\
\quad \texttt{after}(ps): \ exp \ \{ \\
\quad\quad body' \\
\quad \} \\
\}
\end{array}
& = &
\begin{array}{l}
ts \\
\texttt{aspect } A \ \{ \\
\quad pcs \\
\quad as \\
\quad \texttt{pc } \ p(ps): \ exp \\
\quad \texttt{before}(ps): p(\alpha ps)\{ \\
\quad\quad body \\
\quad \} \\
\quad \texttt{after}(ps): p(\alpha ps)\{ \\
\quad\quad body' \\
\quad \} \\
\}
\end{array}
\end{array}
$$

provided

($\rightarrow$) There is no pointcut named p in pcs;

($\leftarrow$) p is not referenced from ts, as and pcs.

The derivation of this refactoring is simple. We start applying Law 26 which creates a new named pointcut. Then, it is necessary to apply Law 27 on every advice that uses the same expression represented by the new named pointcut. This law is responsible for changing an existing advice to use a named pointcut that declares the same expression used by the advice.

We also derived the preconditions of the refactoring from the preconditions of the two involved laws. However, this is not as easy as ANDing all the preconditions [23]. Sometimes preconditions of some laws can not be satisfied from the beginning of the refactoring. They will be satisfied during the transformation, after applying other laws. In this case, just ANDing all the preconditions would yield a precondition that can never be satisfied.

We can intuitively perceive that the advices after the transformation still captures the same set of join points as before. It is due to the fact that a named pointcut serves only to improve reuse of the pointcut expression. Thus, the advices point to the same expression it used before and captures the same set of join points.

The final result should be exactly the same of the proposed refactoring which implies that this refactoring can be considered to preserve behaviour regarding the equivalence notion provided by our laws and provided that the preconditions are respected. Next we show a summary of the applied laws.

$$\boxed{\text{Law 26} \rightarrow \text{Law 27}} \quad \curvearrowright$$

4.2 Extract Method Calls

This refactoring intends to modularize calls to a method appearing in several other methods. This situation happens quite often. Suppose we use the object-oriented refactoring *Extract Method* [7] to extract code that was duplicated. Now that we restructured the code, we have another problem: there are several repeated calls to the extracted method. This problem may characterize a crosscutting concern.

One solution to the problem, is to use aspect-orientation to modularize the method calls. In AspectJ, the concrete solution would be to create an aspect and define an advice which call the method at proper time [17]. The example shown is the same of the refactoring author. It is part of a bank system that checks for user access on every method of the `Account` class.

```
public class Account {
 float balance;
 void credit(float amount) {
  Access.check(new BankPermission("account"));
  balance = balance + amount;
 }
 void debit(float amount) throws ... {
  Access.check(new BankPermission("account"));
  (verify balance and realizes the debit)
 }
}
```

To show the derivation we start from the code showed above, applying our laws to extract the calls to `Access.check`. We start choosing the proper law concerned with advice execution (Laws 3, 5, 7, 9 and 11), based on where the method call appears. If none of those laws can capture the place where the method call is located, we should apply object-oriented refactorings to make the method call fit the template of one of the mentioned laws. In our example, we chose Law 3 and applied it twice, first for the method `credit` and then for the method `debit`, moving the referred method call to an aspect. The obtained aspect is showed next.

```
paspect PermissionCheckAspect {
  before(Account c, float amount) :
     execution(void Account.credit(float)) &&
     this(c) && args(amount){
    Access.check(new BankPermission("account"));
  }
  before(Account c, float amount) :
     execution(void Account.debit(float)) &&
     this(c) && args(amount){
    Access.check(new BankPermission("account"));
  }
}
```

Next, we must simplify our resulting aspect because it has repeated advices with the same action. Therefore we apply Law 13 that is responsible to merge the similar advices promoting a better reuse and legibility. Then we use Laws 14 (Remove `This` Parameter) and 16 (Remove Argument Parameter) to remove the unused `account` and `amount` parameters. Finally, we use the already discussed *Extract Pointcut* refactoring to transform anonymous pointcuts into named ones. Following, we show the resulting aspect.

```
paspect PermissionCheckAspect {
  pointcut accountPermission():
```

```
    (execution(void Account.credit(float)) &&
        this(Account) && args(float)) ||
    (execution(void Account.debit(float)) &&
        this(Account) && args(float));
  before(): accountPermission() {
    Access.check(new BankPermission("account"));
  }
}
```

Note that the code can be further simplified by reducing the pointcut expression. Although it is not our focus to provide such simplification, it would be a valid contribution. Next, we show a summary of the applied laws.

Law 3→Law 13→Law 14 →Law 16→Extract Pointcut

4.3 Extract Worker Object Creation

A worker object is a class that encapsulates a method. An instance of this class is generally created only to be passed as an argument to a method that performs some operations and eventually call the worker object method. This situation is common when executing methods asynchronously, performing authorization using Java Authentication and Authorization Service (JAAS) API, implementing thread safety in Swing/AWT applications, and so on. Generally this is done by creating anonymous classes on demand, or by creating lots of standard classes.

The *Extract Worker Object Creation* [17] refactoring is intended to modularize the worker object creation and simplify its usage logic. The following example shows an ATM class that use the JASS authorization scheme by passing a worker object to `Subject.doAsPrivileged()`. For simplicity, we omit the `final` modifier on the `getBalance()` and `credit()` parameters.

```
public class ATM {
  ...
  public float getBalance(Account account)
      throws BankingException {
    PrivilegedAction worker
      = new PrivilegedAction() {
        public Object run() {
          BankLiaison bl = ...
          return new Float(bl.getBalance(account));
        }};
    Float balance
      = (Float)Subject.doAsPrivileged(...);
    return balance.floatValue();
  }
  public void credit(Account account, float amount)
      throws BankingException {
    PrivilegedExceptionAction worker
      = new PrivilegedExceptionAction() {
        public Object run() throws Exception {
          BankLiaison bl = ...
          bl.credit(account, amount);
          return null;
        }};
    try {
      Subject.doAsPrivileged(...);
    } catch (PrivilegedActionException ex) {
      Throwable cause = ex.getCause();
      throw new BankingException(ex.getCause());
    }
  }
  ...
}
```

As we see, the code is difficult to understand and the method's core logic is tangled inside the anonymous classes. Following we show the refactored code. The authorization concern is now modularized in an aspect and the ATM class is much simpler.

```
public class ATM {
  ...
  public float getBalance(Account account)
      throws BankingException {
    BankLiaison bl = ...
    return bl.getBalance(account);
  }
  public void credit(Account account, float amount)
      throws BankingException {
    BankLiaison bl = ...
    bl.credit(account, amount);
  }
  ...
}
public aspect AuthorizationRouterAspect {
    pointcut authOperations(ATM atm)
        : execution(public * ATM.*(..)) &&
          this(atm) && within(ATM);
    Object around(final ATM atm)
        throws BankingException
        : authOperations(atm) {
      PrivilegedExceptionAction action
        = new PrivilegedExceptionAction() {
          public Object run() throws Exception {
            return proceed(atm);
          }};
      try {
        return Subject.doAsPrivileged(...);
      } catch (PrivilegedActionException ex) {
        return new BankingException(ex);
      }
    }
}
```

The example before the refactoring uses two distinct worker objects, one that completes execution without raising an exception and one that may raise an exception. The resulting aspect, after the refactoring, has only one advice that uses the second version of the worker object in both cases. Hence, the resulting aspect generalizes the use of the worker object to always be able to raise an exception, including the new worker object and exception handling code in the method that was not prepared to handle this exception before.

We did not derive this refactoring because a complex object-oriented transformation would be necessary. It consists of two steps: add a `try-catch` block for the *Subject.doAsPrivileged* call; and then change the type of the *PrivilegedAction* to *PrivilegedExceptionAction*. Figuring out this transformation would help to uncover the refactoring preconditions. For instance, if the *PrivilegedAction* was assigned to an attribute, it would not be possible to change its type, since some other part of the program could use type casts or tests. This is a complex transformation and its complete precondition would be difficult to find out. Assuming we applied this transformation, we would use Law 11 on the two methods, moving the worker object creation to an aspect. It would enable us to merge the resulting advices using Law 13, and finally apply the *Extract Pointcut* refactoring.

4.4 Other Refactorings

Besides the presented refactorings, we have derived several others with satisfying results. The *Extract Exception Han-*

dling [17] showed some problems. The presentation of the proposed refactoring is specific to cases where the handling code only wraps and re-throws another exception. Thereby, our solution as a composition of laws is more general allowing the extraction of different handling code for the same exception. However, generality implies less legibility on the final program due to the more complex code. Another weakness of the proposed refactoring is that it does not mention the preconditions necessary to apply it. As a composition of our laws, we can derive the preconditions from the preconditions of each law involved. The derivation of this refactoring uses Laws 17-21 as explained in Section 3.

The result of applying the *Extract Interface Implementation* [17] refactoring as a composition of our laws is the same of the proposed refactoring itself. Therefore, the composition of laws illustrates that the refactoring preserves behaviour provided that the preconditions from each law used to represent the refactoring are respected.

The *Extract Concurrency Control* [17] showed a limitation of our laws. As we did not deal with abstract aspects, the resulting code on the proposed refactoring is more reusable, as it uses an abstract aspect which provides an structure easily applied in other cases. However, the refactoring used to generate the abstract portion of the aspects can also be applied to the result we obtained using our laws. We intend to extend our set of laws to include this constructs in the future. Hanenberg and Unland [10] discusses some rules to make better reuse of aspects.

As we see, most of the refactorings are called *Extract*. This is not a rule but an expected coincidence. Since the mechanisms provided by AspectJ generally provide a way to extract some behaviour into an aspect. However, some of the laws used to derive the refactorings do not extract code. In fact, there are laws that insert code, for instance Law 17. In addition, some of the used laws just restructure the aspect to achieve better reuse and legibility.

5. REFACTORING TO ASPECTJ

This section shows two case studies in which we use our laws and the refactorings derived in the previous section to restructure two distinct applications. Both applications were previously restructured to modularize crosscutting concerns using an ad-hoc transformation. We use our laws to justify that the ad-hoc transformation preserves behaviour. This is another way to evaluate the laws. In the first case study, we discuss the concurrency crosscutting concern and in the second we discuss distribution. In both cases we successfully achieve the benefits of aspect-orientation.

5.1 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
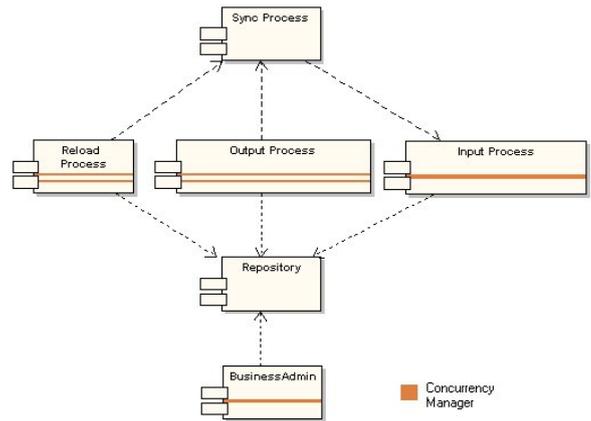
access the repository need concurrency control.

Figure 1: Mobile Server Before Refactoring.

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* 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.

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 [7] 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.

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.beginExec(this.getID(),
```

```
            this.tableNames,this.getUser());
  CM.sort(this.tableNames);
  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.endExec(this.getID(),this.getUser());
 }
}
```

The process starts indicating to the CM the tables that will be required (`beginExec`), 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 (`endExec`). 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 (`beginExec` and `endExec`) 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<this.getTableNames().length;i++){
   this.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.

```
paspect CMAspect {
 void around(APThread c):
      exec(void APThread.process()) &&
      this(c){
   CM.beginExec(c.getID(),
                c.getTableNames(),c.getUser());
   CM.sort(c.getTableNames());
   proceed(c);
   CM.endExec(c.getID(),c.getUser());
 }
 void around(APThread c, String table) :
   withincode(void APThread.process()) &&
   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 can not use an around advice similar to the one used before, we preferred to 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. We used this refactoring on the exceptions related to the concurrency control. The resulting system can be seen as Figure 1 without the CM services been called from the components.

5.2 Health Watcher

The Health Watcher is a real web based system intended to improve the quality of the services provided by health care institutions. By allowing people to register several kinds of health complaints, such as complaints against restaurants and food shops, health care institutions can promptly investigate the complaints and take the required actions. The

system has a web-based user interface for registering complaints and performing several other associated operations.

In order to achieve modularity and extensibility, a layered architecture and associated design patterns [8, 1, 19] were used in the Java implementation of the system. This layer architecture helps to separate data management, business, communication (distribution), and presentation (user interface) concerns. The system also uses the Facade [8] design pattern to provide a single access point to the system business rules.

This structure leads to less tangled code – such as when business code interlaces with distribution code – but does not completely avoid it. This is the case of the code specifying the classes that have to be serializable for allowing the remote communication of its objects. The exception handling code is also scattered throughout the system.

We refactored this application to separate all code related to distribution from the other parts of the system. The system used a common implementation with Java RMI (Remote Method Invocation) to make the system facade remotely available. It is a simple client/server implementation using RMI, where the server is the remote facade and the clients are the servlets that implements the user interface.

This implementation consists of an interface implemented by the facade class (HWFacade). This interface extends from the Remote interface and all of its methods must rise Remote-Exception. Moreover, it is necessary to make all the classes, used as arguments on the facade methods (i.e. Symptom), implement the Serializable interface. The last involved problem is about registering the remote facade on the naming service (server side) and retrieving it (client side).

We start separating the distribution code for the server side. Our first task is to move the serializable implementations to the aspect. In this case, the system only uses the Serializable interface to tag the classes that will be transported through the network. Thus, it is only necessary to move the interface declaration from the classes to the aspect. This is easily achieved with Law 24. We have to apply this law to every class that implements the Serializable interface. If there were transient fields or other behaviour related to serialization, we would use other laws to achieve the modularization.

Next it is necessary to move the remote implementation from the facade to the aspect. We use Law 25 to move the declaration of the Remote interface to the aspect. The following code shows part of the resulting aspect after this steps.

```
paspect ServerSideHW {
  declare parents: IHWFacade extends Remote;
  declare parents: Symptom implements Serializable;
  (implements declarations to other classes)
}
```

The final step on the server side is to move the code which registers the facade on the naming service to the aspect. We use Law 23 to move the main method on the facade class to the aspect, ending the separation of the distribution code on the server side.

```
paspect ServerSideHW {
 public static void HWFacade.main(String[]  args) {
  try {
    HWFacade facade = HWFacade.getInstance();
    UnicastRemoteObject.exportObject(facade);
    Naming.rebind("/HW",facade);
```

```
  } catch (Exception ex) { ... }
 }
}
```

The client side is a bit more complex. All of the client classes (servlets) look for the facade on the naming service to start using it. Another consequence of distribution is the RemoteException that is thrown by all methods in the remote facade, forcing the client classes to handle it. The first part can be achieved using Law 23 to move the facade initialization from the servlets to the aspect.

```
paspect ClientSideHW {
 public void InsertComplaint.initRemoteHW() {
  try {
    Object o = Naming.lookup(..);
    remoteHW = (IHWFacade) o;
  } catch (Exception e) { ... }
 }
}
```

At the end we use the *Extract Exception Handling* refactoring to move all the exception handling related to the RemoteException to the aspect.

We can still improve our result applying the *Extract Interface Implementation* refactoring discussed in Section 4 to move the facade instance and initialization to the servlets superclass, eliminating the repeated code showed on the aspect which introduces the facade initialization on every servlet.

Although we could separate almost all the distribution code, distribution code is still part of the resulting program, since the facade interface still throws RemoteException. This remnant part could not be removed by our *Extract Exception Handling* because we do not have access to the stub implementation. The refactoring would remove the exception from the throws clause on the stub and then remove it on the interface. This was possible when we applied this refactoring to the Mobile Server because we had access to all the implementations of the affected interface.

5.3 Discussion

We showed that the business code separated from the crosscutting concerns is cleaner and more legible, increasing the systems maintainability. Moreover, the aspects increased the systems modularity since the scattered code is now localized inside aspects. The new implementation also reduced the number of 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. For instance, the client code to recover the remote facade was repeated in eighteen servlets.

In terms of affected classes, the first case study was less representative. It only affected four classes and the code extracted generated two aspects, the first responsible for the concurrency control and the second responsible for the exception handling. However, the second case study affected 18 servlets on the client side plus 14 classes on the server side (the facade and serializable classes). In this case, we created three aspects, one responsible for the client side effects, other responsible for the server side effects. The last aspect is responsible for the necessary exception handling.

Another important consequence on the second case study is that it can be generalized, since the distribution implementation of this application is commonly used. Hence, we can generalize the steps used on this case study and derive a refactoring called *Extract Distribution* from the composition of the laws used here.

6. RELATED WORK

The behaviour preserving property of refactoring is not easily proved. Opdyke [22] started showing that preconditions to apply the transformation would help on this task. Afterwards, based on the preconditions of the individual refactorings, Roberts [23] studied the composition of basic refactorings and the derivation of a single precondition to the derived refactoring. However, Opdyke and Roberts do not formally prove that the transformations preserve behaviour. Recently, Kniesel and Koch [16] specialized Roberts concept, deriving the composite precondition based on the weakest precondition of each individual refactoring. We use the concepts of preconditions to define our laws as behaviour preserving transformations. Further, our laws are intended to be composed, generating useful behaviour preserving refactorings.

A previous work [24] used the same case study presented in Section 5.2. Although it presents specific guidelines on how to implement persistence and distribution as aspects by restructuring a pure Java system to an aspect-oriented one, they did not demonstrate that those guidelines are behaviour preserving. In fact, they are not refactorings, they introduce new behaviour. On the other hand our approach uses laws of programming in order to define refactorings, which are behaviour preserving. We restructure the system, therefore the system is supposed to be already distributed.

Several authors discuss refactoring with AspectJ. Some of them [9, 14] address the problem of applying general object-oriented refactorings when using AspectJ. The problem arises from the fact that object-oriented refactoring usually change the structure of join points of the program and thus change how the aspects affect classes.

Hanenberg, Oberschulte and Unland [9] propose some preconditions to apply an object-oriented refactoring in the presence of aspects. Those conditions guarantee a mapping of join points during refactoring, therefore preserving behaviour. They also propose modifications to refactorings such as *Extract Class* [7] in order to make them aspect-aware and therefore respect the preconditions. The second part of Hanenberg, Oberschulte and Unland's research regards refactorings to AspectJ. In fact, they propose some new refactorings from Java to AspectJ. However, they only discuss the refactoring as a whole and the conditions to apply the refactoring. Our approach discusses those kinds of refactorings as basic laws of programming in order to simplify their understanding and proof. We also derived the proposed refactorings using our laws, showing that they preserve behaviour.

Analogously, Iwamoto and Zhao [14] proposes modifications to existing refactorings in order to make them aspect-aware. However, it is a superficial discussion. They only show some examples and give some guidelines on how to avoid the aspect effects on the object-oriented refactorings. They also show examples of refactorings from Java to AspectJ. Although, there is no argumentation about necessary conditions to apply the refactorings to ensure that they preserve behaviour. We used the suggested refactorings and derived them as a composition of our laws. Hence, we were able to state in which conditions we can apply the refactorings as well.

Another related work [11] discusses a tool implemented to support the task of refactoring an aspect-aspect system. Their approach consists in developing a tool to be integrated with the Eclipse IDE. It is designed to involve the developer in a dialog to build the refactoring based on the concern description. The dialog is used to help on the necessary design and implementation decisions during the refactoring process. They have two approaches to achieve that. The first uses a concern graph to describe and implement the refactoring. The second, chooses a target design pattern from the GoF [8] and restructure it using aspects according to a previous work [12].

There is a related work [17] that discusses aspect-oriented refactorings showing problems when applying object-oriented refactorings in the presence of aspects. It proposes several complex and interesting refactorings and shows clear and easy to understand examples. We derived most of the proposed refactorings as discussed in Section 4.

A recent work [25] reports the refactoring of a middleware system to modularize features such as client-side invocation, portable interceptors, and dynamic types. As a future work, we intend to systematize the refactoring applied using our laws. Thus, providing some confidence that the refactored middleware preserves behaviour.

Finally, another recent work [21] proposes a catalog of aspect-oriented refactorings. The refactorings are grouped by categories and described similarly to the way Fowler [7] describes object-oriented ones. It would be interesting to derive the proposed refactorings using our laws. However, this is regarded as a future work.

7. CONCLUSIONS

We propose the use of programming laws for helping developers to deal with the problem of defining behaviour preserving transformations for AspectJ. Those transformations helps to better modularize Java programs by using AspectJ constructs. Moreover, they are useful to restructure AspectJ programs. We derive large and global refactorings from laws that are simple and localized. Our approach gives confidence that a transformation preserves behaviour because we intuitively show that each law preserves behaviour, and thus a composition of those laws also preserves behaviour. One aspect which became evident when defining the laws presented here is that, associated with most of them, there are very subtle preconditions which require much attention. Uncovering preconditions has certainly been one of the difficult tasks of our research.

The derivation of refactorings proposed in the literature showed limitations of the set of laws we use. As mentioned in Section 4, the proposed *Extract Concurrency Control* [17] refactoring results in a more reusable code than we achieve with our laws. It makes use of abstract aspects providing a base to be reused in other applications. Although we do not deal with abstract aspects, as well as get and set pointcuts, we see no further difficulties on defining laws establishing properties of those constructs. Those laws would allow us to achieve a reusable solution with abstract aspects. They would also allow the derivation of refactorings not discussed here, for instance, the *Extract Lazy Initialization* [17]. Additionally, the derivation of the *Extract Worker Object Creation* [17] refactoring showed the necessity for creating new object-oriented transformations.

At last, we used our laws and the derived refactorings to transform two commercial applications separating a crosscutting concern with aspects. On the first case study, we successfully separated concurrency control from the core lo-

gic of the system. On the second case we considered to isolate distribution with aspects and once again we succeeded, despite the remnant exception on the remote interface. Even though our set of laws is not complete in the sense it does not represent every feature of AspectJ, it is representative enough to derive several complex refactorings and to completely restructure common implementations of concurrency and distribution concerns. As a future work we intend to extend this set of laws to include more AspectJ constructs and to show this set is relatively complete. One way to proof the relative completeness is to show that this set of laws is sufficient to reduce an arbitrary program to a normal form [3]. Moreover, we also intend to implement our laws, providing tool assistance and automation. Our approach will extend JaTS [5], a Java transformation language, allowing it to represent AspectJ transformations. Another limitation of our laws is related to their soundness. Although soundness, with respect to a formal semantics, is a required property, it is beyond the scope of this article. However, we intend to work on this in the future. For now, we rely on the simplicity of the laws, which involve only local changes and deal with one AspectJ construct each.

8. ACKNOWLEDGMENTS

9. REFERENCES

[1] V. Alves and P. Borba. Distributed Adapters Pattern: A Design Pattern for Object–Oriented Distributed Applications. In *1st Latin American Conference on Pattern Languages Programming - SugarLoafPLoP*, Rio de Janeiro, Brazil, October 2001.

[2] G. Booch. *Object–Oriented Analysis and Design with Applications*. Benjamin/Cummings, second edition, 1994.

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

[4] L. Cole. Deriving refactorings for AspectJ. Master's thesis, Informatics Center, Federal University of Pernambuco, Recife-PE, Brazil, February 2005. Available at http://www.cin.ufpe.br/spg/GenteAreaThesis.

[5] M. d'Amorim, C. Nogueira, G. Santos, A. Souza, and P. Borba. Integrating Code Generation and Refactoring. In *Workshop on Generative Programming, ECOOP02*, Malaga, Spain, June 2002. Springer Verlag.

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

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

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

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

[10] S. Hanenberg and R. Unland. Using and reusing aspects in AspectJ. In *Workshop on Advanced Separation of Concerns in Object-Oriented Systems , OOPSLA'2001*, Oct. 2001.

[11] J. Hannemann, T. Fritz, and G. C. Murphy. Refactoring to aspects: an interactive approach. In *Proceedings of the 2003 OOPSLA Workshop on Eclipse Technology eXchange*, Anaheim, California, USA, Oct. 2003.

[12] J. Hannemann and G. Kiczales. Design pattern implamentation in java and AspectJ. In *OOPSLA'2002*, pages 161–173, Seattle, Washington, USA, Nov. 2002.

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

[14] M. Iwamoto and J. Zhao. Refactoring aspect-oriented programs. In *The 4th AOSD Modeling With UML Workshop*, 2003.

[15] 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.

[16] G. Kniesel and H. Koch. Static composition of refactorings. In R. Lämmel, editor, *Science of Computer Programming*, Special issue on "Program Transformation". Elsevier Science, 2004.

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

[18] R. Lämmel. A Semantic Approach to Method-Call Interception. In G. Kiczales, editor, *1st International Conference on Aspect-Oriented Software Development (AOSD 2002)*, pages 41–55, Twente, The Netherlands, Apr. 2002. ACM Press.

[19] T. Massoni, V. Alves, S. Soares, and P. Borba. PDC: Persistent Data Collections pattern. In *1st Latin American Conference on Pattern Languages Programming - SugarLoafPLoP*, Rio de Janeiro, Brazil, October 2001.

[20] B. Meyer. *Object–Oriented Software Construction*. Prentice–Hall, second edition, 1997.

[21] M. Monteiro and J. Fernandes. Towards a Catalog of Aspect-Oriented Refactorings. In *4th International Conference on Aspect-Oriented Software Development (AOSD 2005)*, Chicago, USA, Mar. 2005. ACM Press.

[22] W. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, IL, USA, 1992.

[23] D. Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois at Urbana-Champaign, IL, USA, 1999.

[24] S. Soares, E. Laureano, and P. Borba. Implementing distribution and persistence aspects with AspectJ. In *Proceedings of the OOPSLA '02 conference on Object Oriented Programming Systems Languages and Applications*, pages 174 – 190. ACM Press, November 2002.

[25] C. Zhang and H.-A. Jacobsen. Resolving feature convolution in middleware systems. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2004)*, pages 13–26, Vancouver, British Columbia, Canada, October 24–28, 2004.