CHAPTER 3

LAWS

As far as the laws of mathematics refer to reality, they are not certain;
and as far as they are certain, they do not refer to reality.
—ALBERT EINSTEIN

Sometimes, modifications required by refactorings are difficult to understand as they might perform global changes in an application. We use laws of programming [16] 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 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. Note that this is our second law. The first law, *Add Empty Aspect*, is very simple and can be found in Appendix A.

Law 2. Make Aspect Privileged

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

provided

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

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

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
class C {
  fs
  ms
  T m(ps) {
    body';
    body
  }
}
paspect A {
  pcs
  as
}
```

=

```
ts
class C {
  fs
  ms
  T m(ps) {
    body
  }
}
paspect A {
  pcs
  as
  before(context) :
        execution(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 assume that we 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.

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.

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

Table 3.1. 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

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.

Law 4. Add Before-Call

```
ts
class C {
  fs
  ms
  T n(ps') {
    body;
    exp.m(αps)
  }
}
paspect A {
  pcs
  as
}
```

=

```
ts
class C {
  fs
  ms
  T n(ps') {
    exp.m(αps)
  }
}
paspect A {
  pcs
  as
  before(context) :
        withincode(σ(C.n())) &&
        call(σ(O.m()) &&
        bind(context) {
    body[cthis/this]
  }
}
```

provided

(→) *body* does not declare or use local variables; *body* does not call `super`;

(←) *body* does not call `return`;

(↔) O is the type of *exp*;

We use the `withincode` operator of AspectJ to restrict the calls to the captured method occurring only inside the originating method (n). We also denote α preceding a list of parameters to represent the list its values. 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.

Next, we start exploring the `after` advices. The first case is when a piece of code executes after another, independently of how the first one finishes execution. The Java construction to do that is a `try--finally` block. The `try` block executes and then `finally` block executes, even if the first part raises an exception. This structure maps to the construction of a simple `after` advice in AspectJ shown in Law 5.

Law 5. Add After-Execution

```
ts
class C {
  fs
  ms
  T m(ps) {
    try {
      body
    } finally {
      body'
    }
  }
}
paspect A {
  pcs
  as
}
```

=

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

provided

(→) *body*′ does not declare or use local variables; *body*′ does not call `super`;

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

The preconditions to applying this transformation are the same as Law 3 preconditions. In fact, all the laws used to introduce advices have a similar behaviour because they all make a piece of code execute near a join point on the base code. Again, notice that this transformation applied from right to left is almost the same applied by the AspectJ compiler when weaving an `after` advice.

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. The following law 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.

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

($\leftarrow$) *exp*1 and *exp*2 bind all parameters in *ps*.

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. This law 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. For instance, the next law is responsible for removing a `target` parameter of an advice provided that the parameter is not used in the advice body.

Law 15. Remove Target Parameter

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

provided

(→) *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*. 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 the type of the unchecked exception used by AspectJ, it wraps the softened exception. There is a similar version of this law, which uses the pointcut designator call.

Law 18. Soften Exception

```
ts
class C {
  fs
  ms
  T m(ps) throws E, exs {
    body
  }
}
paspect A {
  pcs
  as
}
```

=

```
ts
class C {
  fs
  ms
  T m(ps) throws exs {
    body
  }
}
paspect A {
  declare soft : E :
        execution(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

```
ts
class C {
  fs;  T field
  ms
}
paspect A {
  pcs
  as
}
```

=

```
ts
class C {
  fs
  ms
}
paspect A {
  T C.field
  pcs
  as
}
```

provided

(→) 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.

The following law has the purpose of moving the implementation of a single method into an aspect using an inter-type declaration. According to the AspectJ semantics, visibility modifiers of inter-type declarations are related to the aspect and not to the affected class. Hence, it is possible to declare a private field as a class member and as an inter-type declaration at the same time and using the same name. As a consequence, transforming a member method that uses this field, into an inter-type declaration, imply that the method now uses the aspect inter-typed field. This leads to a change in behaviour. A precondition is necessary to avoid this problem.

Law 23. Move Method to Aspect

```
ts
class C {
  fs
  ms
  T m(ps) throws es {
    body
  }
}
paspect A {
  pcs
  as
}
```

=

```
ts
class C {
  fs
  ms
}
paspect A {
  T C.m(ps) throws es {
    body
  }
  pcs
  as
}
```

provided

($\leftrightarrow$) A does not introduce any field to C with the same name of a C field used in *body*

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 [21] where we can represent most 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.