Aspect-Oriented Programming Reloaded

Henrique Rebêlo and Gary T. Leavens

CS-TR-17-03
May 2017

Keywords: Aspect-oriented programming, modularity, AspectJ, AspectJML
2017 CR Categories: D.1.5 [*Programming Techniques*] Object-Oriented Programming — languages, tools, AspectJML language; D.2.1 [*Software Engineering*] Requirements/Specifications — languages, methodologies; D.2.3 [*Software Engineering*] Coding Tools and Techniques — object-oriented programming, aspect-oriented programming, AspectJ language, AspectJML language; D.3.3 [*Programming Languages*] Language Constructs and Features — classes and objects, control structures, modules, packages, procedures, functions, and subroutines, AspectJ language, AspectJML language;

Submitted for publication

Copyright © 2017, Henrique Rebêlo and Gary T. Leavens

Dept. of Computer Science, University of Central Florida
4000 Central Florida Blvd.
Orlando, Florida 32816, USA

Aspect-Oriented Programming Reloaded

Henrique Rebêlo
Universidade Federal de Pernambuco
Recife, PE, Brazil
hemr@cin.ufpe.br

Gary T. Leavens
University of Central Florida
Orlando, FL, USA
leavens@cs.ucf.edu

ABSTRACT

Many programs have crosscutting concerns for which neither procedural nor object-oriented programming adequately modularize, which has led to the idea of aspect-oriented programming (AOP). However, AOP has not found favor with the programming languages community due to a belief that AOP breaks classical modularity and modular reasoning. We propose a new AOP programming model that enables both crosscutting modularity and modular reasoning. This model is implemented by AspectJML, a general-purpose aspect-oriented extension to Java. It supports modular crosscutting concerns using key object-oriented mechanisms, such as hierarchical structure, and allows reasoning that scales to ever-larger programs.

CCS CONCEPTS

•**Cross-computing tools and techniques** → **reliability, validation;** •**Software creation and management** → *software verification and validation;* •**General programming languages** → *language features;*

KEYWORDS

Aspect-oriented programming, modularity, AspectJ, AspectJML

ACM Reference format:
Henrique Rebêlo and Gary T. Leavens. 2017. Aspect-Oriented Programming Reloaded. In *Proceedings of Submitted, Earth, May 2017,* 9 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Two decades ago aspect-oriented programming (AOP) emerged, as a new programming model to deal with crosscutting modularity problems [18]. For such crosscutting problems, neither procedural nor object-oriented programming techniques are sufficient to clearly modularize the design decisions that a program must implement. These conventional programming techniques force the implementation of certain design decisions to be scattered throughout the code, thus resulting in tangled code that is excessively difficult to develop and maintain. Such design decisions are what AOP calls *crosscutting concerns*, since they cut across the system's basic functionality and force programmers to work around the standard modularization enforcement mechanisms by scattering code throughout a program [16].

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
Submitted, Earth
© 2017 Copyright held by the owner/author(s). 978-x-xxxx-xxxx-x/YY/MM. . . $15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

AOP became popular because it makes possible for programmers to enable the modular crosscutting structure for concerns such as distribution and persistence [39], error handling [7], some design patterns [9, 12], design by contract [17, 36] etc. To date, AspectJ [17] still is the most influential AOP language.

However, AspectJ's constructs [17], such as pointcuts and advice, aimed supporting crosscutting modularity, have been considered by some to have a negative impact on modularity [1, 3, 29, 33, 37, 40, 41]. The difficulties center around *obliviousness* [8], which means that crosscutting concerns are not visible in the code that they may affect. However, obliviousness compromises classical notions of modular reasoning [30], since by definition the code being advised contains no trace of the aspects that advise it. Our approach is to abandon obliviousness and create a new programming model, called the hierarchical crosscutting model, which enables classical modularity and modular reasoning, and yet which is able to do most of what AOP languages can do.

We explain the hierarchical crosscutting model through a prototype extension to Java called AspectJML. With AspectJML, programmers feel like they are programming in plain Java, but AspectJML also allows them to modularize crosscutting concerns without having any meta-level shift [16] or without having any extended syntax shift for advanced modularity [5].

In the rest of the paper we show how object-oriented hierarchies can be used to cope with crosscutting modularity. We also present some preliminary evaluation of AspectJML and point out directions for future work.

2 A PLETHORA OF ISSUES IN AOP

In this section, we revisit the well-known AOP programming model along with its main modularity issues, as demonstrated by AspectJ-like languages.

2.1 The AOP Programming Model

AOP mainly focuses on the modularization of crosscutting concerns. To enable this modularization, AOP provides both obliviousness and quantification [8]:

$$\text{AOP} = \text{quantification} + \text{obliviousness} \quad (1)$$

Obliviousness means says that certain parts of a program, called the "base code," have no knowledge of the aspects that advise them. Quantification means that advice can be applied to multiple sites in the base code with a small amount of text [8, p. 2]. Thus AOP may be seen as a specialized form of implicit invocation [10], where procedures are called without explicitly referencing them (obliviousness), and where such implicit invocation can affect several parts of a program (quantification [45]). A special form of implicit invocation is implicit invocation with implicit announcement of events (IIIA) [33, 40, 46].

```
interface Shape {
  void moveBy(int dx, int dy);
}
class Point implements Shape {
  int x, y;

  void setX(int x) {this.x = x;}
  void setY(int y) {this.y = y;}

  void moveBy(int dx, int dy) {
    x += dx; y += dy;
  }
  //... other methods
}
class Line implements Shape {
  Point p1, p2;

  void setP1(Point p1) {this.p1 = p1;}
  void setP2(Point p2) {this.p2 = p2;}

  void moveBy(int dx, int dy) {
    p1.x += dx; p1.y += dy;
    p2.x += dx; p2.y += dy;
  }
  //... other methods
}

aspect UpdateSignaling {
  pointcut change():
    execution(void Point.set*(*))
  || execution(void Line.set*(*))
  || execution(void Shape+.moveBy(int, int));

  after(): change() {
    Display.update();
  }
}
```

Figure 1: The classical AspectJ implementation for the shape classes with display update signalling [17].

The most influential AOP language, AspectJ [17], is an instance of IIIA. As an example, consider the classical AspectJ implementation for the shape classes in Figure 1. It illustrates the crosscutting concern update signalling, in which whenever a shape changes, a call to
lstinline!Display.update! must be made. Through implicit invocation, AspectJ enables the modularization of this update signalling crosscutting concern; otherwise scattered in plain Java. Basically, AspectJ works with advice and pointcuts to enable modular crosscutting. For example, the AspectJ **after** advice, declared within the UpdateSignalling aspect, is executed without an explicit reference to the advice from the shape-like classes code. The places where the advice should be executed are called *join points*, which are well-defined points in system's execution [17]. Such join points are selected by AspectJ pointcuts, such as change in Figure 1. It collects all the places, from the classical system structure, where advice should be applied to enable modular crosscutting behavior.

Such oblivious quantified statements in AspectJ, via IIIA mechanism, resulted in an intense debate about whether AOP constructs aimed to support crosscutting modularity actually break class modularity [1, 3, 29, 33, 37, 40, 41]. Next, we discuss the main AOP issues in supporting the current Formula 1.

2.2 Non-orthogonality and asymmetry

According to some authors, AspectJ-like languages present an unnecessary lack of orthogonality and symmetry [13, 34]. The asymmetry in AspectJ-like languages are related to aspect modules, advice and pointcuts [13, 34]. For instance, since AspectJ makes explicit the distinction between advised code (classes) and advising code (aspects), as can be seen in Figure 1. Besides this asymmetric separation, classes cannot advise aspects in the same way that they advise classes. Advice can only be declared in aspects, whereas pointcuts can be declared in both classes and aspects.[1] Moreover, although both advice and methods support procedural abstraction [24], unlike methods, advice declarations are anonymous. This results in a pointcut asymmetry since one cannot identify advice by name in pointcuts. To advise advice one needs to use the *adviceexecution* pointcut, which selects all advice executions in a running program. The are other important semantic implications, like the non-overriding effect in advice since advice is anonymous [34].

2.3 Non-modular reasoning

Modular reasoning means to conclude properties of a module (like a class or aspect) just by considering its interface, specification and implementation, and the interfaces and specifications, but not the implementations, of modules referenced by it [22, 38]. However, reasoning about aspect-oriented programs that use pointcuts and advice, as found in AspectJ, often seems difficult, due to its implicit invocation feature [30, 38, 40]. As a consequence, the task of determining what advice applies to a particular join point (method of interest) must be done in several non-modular steps.

For example, suppose one wants to reason about the behavior of the method moveBy in Point (see Figure 1). Since the method's behavior may be changed by advice, one needs to first determine what advice may apply to that method. In our example, there is just have one **after** advice in aspect UpdateSignalling, but due to obliviousness, the program text does not reveal if there may be other advice. This this determination of what advice applies must consider all aspects in the program — it is a whole program analysis.

Some important argumentations. In an interview in 2006, Kiczales said that AspectJ was intellectually controversial at the time it came in and still is [16]. He went further when he said that AspectJ has a different modularity enforcement mechanism than traditional languages. Today we can understand some of these issues because AspectJ has its roots on non-classical modularity. Ostermann et al. [30] describes that the classical understanding of modularity is highly related to classical logic. Hence, the common notion of modularity, especially the idea of information hiding (discussed below), is deeply related to classical logic. As such, some modularity mechanisms, like AOP and object-oriented inheritance, leave the world of classical logic, and correspond to a non-classical logic.

2.4 Pointcut fragility

Another well-known AOP problem is pointcut fragility [11]. The fragile pointcut problem of AspectJ refers to the dependence of

[1] The work by Rajan and Sullivan's [34] reports that we can only declare pointcuts within aspects. But in practice, AspectJ does allow pointcuts in classes, thus making the nonorthogonality even more evident.

pointcut descriptions on names in advised code. Such pointcut descriptions are vulnerable to changes in the base code (such as renaming setX to changeX or modifying its parameter declarations), which may cause the pointcut to no longer refer to the same set of join points. This problem stem from the way that AspectJ declares pointcuts.

The dependence of pointcut descriptions on program names in the base code also imposes extra ordering constraints on development. In particular, developers must agree on naming conventions that advice can rely on before pointcut descriptions that use such naming conventions can be written.

2.5 Information *not* hiding

Information hiding [32] (also known as black-box abstraction) is a widely accepted principle in software development. It advocates that a module should expose its functionality but hide its implementation behind an interface. This supports modular reasoning and independent evolution/maintenance of the hidden parts of a module. If programmers have carefully chosen to hide those parts "most likely" to change [32], most changes, in the hidden implementation details, do not affect the module's clients. Information hiding and its benefits apply not only to code but also to other artifacts, such as documentation and specifications [21, 35].

Unfortunately, interfaces supported by current AspectJ-like pointcut mechanisms fall short with respect to information hiding. For instance, the execution join points captured by the pointcut change (see Figure 1) is coupled to the base code. Thus, the resulting design does not actually perform much better in terms of absorbing changes [11, 31]. Besides fragility, due to coupling, the aspect UpdateSignalling cannot be reused by another system (with different set of join points) in its full generality. So, this reduces the power of aspects to support pluggability of crosscutting concerns.

2.6 Extended Syntax for Modularity

Providing mechanisms for modularization is one of the primary concerns of programming languages in general. In this respect, Chiba et al. [5] asked the question "Do we really need to extend syntax for advanced modularity?" Since the trend for the last twenty years is based on traditional AOP mechanisms like pointcuts and advice declarations (as in Figure 1), the question is whether such new syntax is really needed to achieve modularity?

Pointcuts and advice, since they automate the Observer pattern, might be seen not as new syntax, but as syntactic sugar. However, when new kinds of crosscutting concerns arise, these tend to prompt the invention of new kinds of syntactic extensions [5]. This concern is also related to programming languages symmetry, because the addition of new features, along with new supporting syntax, is always fraught with the potential to cause unexpected interactions [13, 26, 34]. In AspectJML, by contrast, we try to follow Rajan and Sullivan [34] in reusing existing language mechanisms, such as methods.

3 PRELIMINARY SOLUTIONS

There have been several previous attempts to address the issues described in the previous section, either by restricting quantification and obliviousness, or by providing an unified model for both classes and aspects. In this section, we briefly discuss these prior works.

3.1 Restricting Quantification and Obliviousness

Formula 1, which describes AOP as a combination of quantification and obliviousness, is the source of the programming language community's unease about AOP's support for modularity. That is, due to obliviousness, the shape classes (in Figure 1) do not mention the aspect UpdateSignalling, which precludes modular reasoning.

On the other hand, while obliviousness has been the subject of the main critics about AOP, the quantification property has advantages [45]. Quantification is what allows one to selectively apply implicit invocation to modularize crosscutting concerns [8]. However, when combined with obliviousness, there are still modularity issues. What if the programmer of the shapes classes, in Figure 1, add a method say setAlready?. Since the name of this method starts with set, the pointcut change in the aspect UpdateSignalling will match. What is really questionable is whether or not such an aspect interference is desired. If it is not, then the behavior of the setAlready method may be modified by the aspect, producing incorrect behavior. There is a growing consensus in the aspect-oriented community that an interface between the base code and aspect code is necessary for alleviating such problems [1, 3, 29, 33, 37, 40, 41].

For instance, Bodden et al. [3] define *join point interfaces*, which are type-based contracts between aspects and advised code. Similarly, Rebêlo et al. [37] provide *design rule interfaces* between aspects and classes. That is, they support the notion of implicit announcement, but restrict its scope to that of the implementation of an explicit interface. The *Join Point Types* (JPT) work [40], also enables implicit announcement via interfaces, but also supports explicit announcement where explicit one fall short (i.e., quantification failure [42]).

3.2 Aspect-aware interfaces

Kiczales and Mezini [19] argue that programmers can reason modularly about aspect-oriented programs by using *aspect-aware interfaces* (AAIs), which are maps that send each program point to a list of advice that applies at that point. AAIs are created using global system knowledge (i.e., a whole program analysis). Kiczales and Mezini say that since crosscutting concerns are inherently global, such AAIs are a sensible starting point for reasoning about programs. However, it is clear that a development methodology that uses AAIs postpones reasoning until after the development of the entire program, since the AAIs cannot be computed until the entire program (base code and aspects) are available.

Another way to view AAIs is that they provide a way to discard obliviousness, since the entire point of an AAI is to make where advice applies available. So the AAI proposal corroborates the idea that some notion of aspect interface is important for modular reasoning.

3.3 Fluid Modularity

In another attempt to restore modular reasoning in aspect-oriented programming, Hon and Kiczales' work discusses modularity through a notion of fluidity called Fluid AOP [15]. With fluid AOP, the programmer can temporarily shift a program to an alternative crosscutting module structure to enable or regain classical reasoning [30]. For example, with Fluid AOP, the shape classes (recall Figure 1) programmer, can shift from the AOP version (with aspect modules)

to one without AOP (only classes and crosscutting structure as is). Hence, to regain modular reasoning, one can understand the control-flow effects by looking the modules without AOP, which have no aspect indirection. However, if the program is written with pointcuts and advice, then the AOP side view cannot be computed until the entire program is available. Thus, as with the AAI [19] approach, modular reasoning cannot occur until after the entire program is available for this transformation.

One step beyond Fluid AOP is given by Chiba et al. [5]. This work describes how to use dynamic text to overcome the need for extending syntax to regain modularity. Dynamic text is an automatic technique that edits the source code while editing or browsing in accordance with the user's directions about crosscutting. Like Fluid AOP, their goal is to tackle the limitations provided by AspectJ-like syntax and regain classically modular reasoning. While Fluid AOP still uses AspectJ-like pointcut definitions for specifying crosscutting structures, Chiba et al. demonstrate with concrete examples how to provide advanced modularization to an object-oriented language, such as Java, without modifying the original syntax. While this approach is more symmetric, compared to Fluid AOP, it still suffers the limitation that it regains modularization only through specific IDE support (e.g., an Eclipse plugin).

3.4 Stepping back: OOP!

One direction explored elsewhere is to work with object-oriented programming (OOP) using plain objects, as in Java [4, 34]. This is the direction followed by Eos [34] and @AspectJ (often pronounced as "at AspectJ") [4]. The @AspectJ syntax was conceived as a part of the merge of standard AspectJ with AspectWerkz [4], and uses the metadata annotation facility of Java 5. The main advantage of this syntactic style is that one can compile a program using a plain Java compiler, allowing the modularized code using AspectJ to work better with conventional Java IDEs and other tools that do not understand the traditional AspectJ syntax.

Figure 2 illustrates a simplified @AspectJ version of the update signalling crosscutting concern previously implemented with the traditional AspectJ syntax (see Figure 1). Instead of using the **aspect** keyword, the class is annotated with an **@Aspect** annotation. This tells the compiler to treat the class as an aspect declaration. Similarly, the **@Pointcut** annotation marks the empty method change as a pointcut declaration. The name of the method serves as the pointcut name. Finally, the **@After** annotation marks the method update as an **after returning** advice. The body of the method is the after advice, which is executed after the matched join point's execution returns (with or without throwing an exception). As with @AspectJ, Eos works with similar classes that act as aspects, like those in AspectJ.

Following the more symmetric models of @AspectJ and Eos, we can adapt the Formula 1 to the following one:

$$\text{AOP} = \text{OOP} + \text{quantification} + \text{obliviousness} \quad (2)$$

However, since this formulation still includes obliviousness, this revised notion of AOP will have the same modularity issues discussed previously before; although in a more symmetric model. For example, in @AspectJ an ordinary Java class plays the exact same role as a standard AspectJ aspect.

```
@Aspect
class UpdateSignalling {
 @Pointcut("execution(* Point.set*(*))")
 public void change() {}

 @After("change()")
 public void update(Shape s) {
  Display.update();
 }
}
```

Figure 2: A simplified @AspecJ syntax to modularize the crosscutting update concern illustrated in Figure 1.

```
class Point implements Shape {
  int x, y;
  //@ invariant x >= 0 && y >= 0;

  /*@ requires x + dx >= 0 && y + dy >= 0;
    @ ensures x == \old(x) + dx
    @       && y == \old(y) + dy;
    @*/
  void moveBy(int dx, int dy) {
    x += dx; y += dy;
  }

  //... other methods and their specifications
}
```

Figure 3: Example of JML specifications for the Point class of the shape classes presented in Figure 1.

4 TOWARDS A NEW ASPECT FORMULA

In this section, we revisit the common notion of classical modularity and present a new AOP formula (third one) that is used (in the next section) to avoid the modularity issues exhibited by the previous AOP formulas (Formula 1 and Formula 2).

4.1 Classical modular reasoning

Monotonicity. Monotonic reasoning allows one to prove properties "once and for all," never needs to withdraw any conclusion when more is learned. For example, if we establish a property of a program using a monotonic logic, we do not need to revise that property again [30, 38]. When more components are added, the previously established property does not change. This is the case of the well-known Hoare logic [14]. Specification languages, such as JML [20], that enable one to establish properties of a program that must be respected (e.g., precondition). Hence, monotonicity in reasoning aids one in using the classical notion of modularity and modular reasoning [30, 38].

Figure 3 shows an example of JML specifications for the Point class. JML specifications are written in annotation comments which start with an at-sign (@). In JML, preconditions are introduced by the keyword **requires** and postconditions by **ensures**. The JML notation **\old**(_x+dx) means the pre-state value of _x+dx. The invariants in JML are introduced by the keyword **invariant**. The invariant defined in this example restrict points to the upper right quadrant.

To reason about a call to moveBy, a programmer must determine what specifications to use. In this case, the specifications are the pre- and postconditions of moveBy and the declared invariant. Since there are no specifications in Point's supertype (Shape), the programmer

does not need to include them in reasoning; otherwise, the specifications inherited from such supertypes [22, 38] would also need to be considered.

Expanded modular reasoning. Recall that our notion of modular reasoning means that one can verify a piece of code in a given module, such as a class, using only the module's own specifications, its own implementation, and the interface specifications of modules that it references [22, 30, 38].

For instance, to reason about the method call sp.moveBy(10,10) one uses the specification of the moveBy method from sp's static type, say ScreenPoint. Assuming that ScreenPoint is declared as a direct subclass (and hence subtype) of Point, then the method specification used would be the join of the method specifications for moveBy from Point and ScreenPoint, together with the conjunction of their declared invariants. The join of method specifications [22] means that both pairs of pre- and postconditions must be obeyed by such an overriding method; thus ScreenPoint's method moveBy must obey the specifications given for it in the class Point. Similarly the conjunction of invariants means that ScreenPoint objects must also obey the invariant given in the class Point. Since classes in Java name their supertypes, such reasoning is not technically outside the standard definition of modular reasoning, however we emphasize the use of all ancestor supertype specifications (recursively) by calling this *expanded modular reasoning*.

That is, expanded modular reasoning also applies to indirect ancestor types, not just direct supertypes. For example, since Point implements the interface Shape, if any specifications are provided by Shape, they must also be used in reasoning about ScreenPoint objects. (However, in our example, there are no specifications in type Shape, nor are there any specifications relevant to a call to the moveBy method in type Object, which is the top of the type lattice in Java.) If necessary all specifications in all ancestor types are consulted in expanded modular reasoning [22, 23, 30, 38].

4.2 A new aspect formula

The classical notion of modularity, which is related to classical logic [30], is exactly what we need to refine Formula 2. The elements of this refinement are scoped-quantification [36, 37, 40] and langugage-level obliviousness [42].

Scoped-quantification. *Scoped-quantification* is a form of quantification that is limited to a statically-defined area of a program. In a specification language like JML [20], invariants are a good example of scoped-quantification [36, 37, 40]. Such an invariant must be preserved by every method in its declaring type as well as every method declared in its subtypes, and there is no way to constrain/quantify unrelated types. Hence, the scope of quantification, provided by invariants, is the type itself extending to its subtypes. History constraints [25], also present in JML, are another example of a scoped-quantified declaration.

Language-level obliviousness. *Language-level obliviousness* [42] is what is allowed when advising constructs, such as AspectJ's advice [17], are introduced to a programming language. Recall again invariants or history constraints of a specification language like JML. They exhibit language-level obliviousness, since the methods they constrain have no explicit references to these scope-quantified declarations.

Formulation. With these above definitions, we can now restrict quantification and obliviousness to a form compatible with classical modularity [30]. This results in the following simple equation:

AOP = OOP + scoped-quantification + language-level obliviousness (3)

5 RELOADING ASPECTS

We now use the new AOP formula (Formula 3), presented above, and discuss how it supports crosscutting modularity and avoids the issues described previously in Section 2. We also introduce AspectJML, as a general-purpose aspect-oriented extension to Java, and explain how it embodies our formulation of our new AOP.

5.1 The AspectJML language choice

What. AspectJML itself was described in a previous work [36]. It enables programmers to specify pre- and postconditions in a modular fashion. There are some contracts that present crosscutting structure and to cope with them, AspectJML allows the crosscutting contract specification mechanism, or XCS for short. It is based on the alternative @AspectJ syntax, which is fully compatible with plain Java code. This merge enables crosscutting concern implementation by using constructs based on the metadata facility of Java 5.

Why. Our previous work is deeply related to this one since it faced the same modularity problems we discuss here, such as modular reasoning. With AspectJML, a programmer can reason about crosscutting contracts using the classical notion of modularity [30], i.e., expanded modular reasoning. However, AspectJML was originally descrbed [36] as a domain-specific language (DSL) that dealt with only one kind of crosscutting concern, which is design by contract [27].

From DSL to general-purpose. Now, AspectJML is becoming [2] a general-purpose aspect-oriented language extension to Java. This means that a programmer using AspectJML can deal with any kind of crosscutting concern, thanks to AspectJML's support for Formula 3. That is, in addition to design by contract, programmers can also deal with concerns such as logging, security, tracing, persistence, etc. Moreover, one can also apply the crosscutting contract specification feature [36] to specify these other crosscutting concerns.

As with our previous work, the current AspectJML language has been developed using @AspectJ syntax. Hence, an AspectJML program looks like an @AspectJ program. But they differ in several meaningful ways. For instance, there is no need to use a class as a third-party aspect module, and one can define and implement advice in any class or interface, unlike AspectJ/@AspectJ. We highlight the main differences below while explaining the main AspectJML features.

Compilation strategy. The AspectJML language is implemented with the existing infrastructure of the AspectJML/ajmlc compiler [36]. This compiler uses the standard AspectJ/ajc compiler as a back-end to generate Java bytecode.

5.2 First examples of new AOP

We return to the update signalling crosscutting concern in the shape classes (Figure 1), and use it to provide some reimplementations of that application using AspectJML.

[2]The language is currently in development, but the main ideas of Formula 3 are already available as a prototype implementation.

```
class Point implements Shape {
  @After("execution(void set*(*))" +
     "|| execution(void moveBy(int,int))")
  void update(){
    Display.update();
  }

  //... other methods
}
```

Figure 4: Example of an AspectJML @After advice declared within class Point.

```
interface UpdateSignalling {
  @After("execution(void set*(*))" +
     "|| execution(void moveBy(int,int))")
  default void update(){
    Display.update();
  }
}
class Point implements Shape, UpdateSignalling {...}
class Line implements Shape, UpdateSignalling {...}
```

Figure 5: Example of the full modularization of the crosscutting concern update signalling in AspectJML using a standard OOP hierarchy.

Crosscutting types. The first notion we present is *crosscutting types*. By crosscutting types we mean that any valid type in Java (e.g., class or interface) can declare an advice and providing crosscutting modularity. Figure 4 illustrates a simple example of a crosscutting type. Let us assume that there is a dedicated programmer for the type Point. Such a programmer can enable local crosscutting modularity by declaring an AspectJML **@After** advice, which intercepts set-like methods (named set*), and the method moveBy.

AspectJML has scoped-quantification, thus the programmer does not need to explicitly write a type pattern for advice (which AspectJ uses to identify the type owner of the intercepted join point). Thus, the two execution pointcuts declared in the advice method update, only intercept join points in the type Point and its subtypes; they do not intercept join points in any other unrelated type, such as Line (as they would in an AspectJ pointcut written without a limiting type pattern).

With respect to modular reasoning, a programmer should reason about the quantified statements, provided by the advice method update, similarly to invariants or history constraints in JML. For example, in order to reason about the setX in Point, that advice should be included in the reasoning. In this case, there is a match in which the behavior of the setX is augmented after its execution.

In relation to crosscutting implementation, as observed, AspectJML eliminates anonymous advice in favor of standard named ones as we have in Java. Hence, the same notion of procedure abstraction is used here to localize any crosscutting concern [24]. The main difference is, unlike regular methods, advice methods in AspectJML are annotated with a corresponding pointcut (in AspectJ-like syntax).

Hierarchical crosscutting. It is claimed in the literature [17, 18] that, while useful, hierarchical modularity cannot enable the modularization of crosscutting concerns. Contradicting this common belief, AspectJML enables one to use a crosscutting type as a supertype that crosscuts the entire hierarchy enabling the modular

```
interface UpdateSignalling {
  @Advice("join points should be only exposed" +
          "within the appropriate modules")
  default void update(){
    Display.update();
  }
}
interface UpdateSignallingShapes extends UpdateSignalling {
  @After("execution(void set*(*))" +
     "|| execution(void moveBy(int,int))")
  default void updateForShapes(){
    UpdateSignalling.super.update();
  }
}
class Point implements Shape, UpdateSignallingShapes {...}
class Line implements Shape, UpdateSignallingShapes {...}
```

Figure 6: Example of a reusable implementation of the crosscutting concern update signalling in AspectJML.

implementation of crosscutting concerns. Figure 5 presents the supertype UpdatingSignalling that localizes the crosscutting concern related to update shapes. In order to crosscut each type that exhibits crosscutting structure, an ordinary Java programmer just needs to implement that interface to activate/plug the crosscutting functionality. As mentioned, the reasoning is similar to a quantified statement like an invariant in JML. Scoped-quantification and language-level obliviousness in AspectJML ensures the classical notion of modularity enforcement (expanded modular reasoning) that is present in OOP. That is, through explicit reference to supertypes, the classical notion of modular reasoning is missing in the traditional AOP formulation (see Formula 1), is regained in the new AOP formulation with Formula 3.

As observed, we implement and localize the crosscutting concern with a plain Java interface. A key concept of our approach is, that since Java 1.8, we can implement methods in interfaces through **default** methods. This approach was proposed earlier [28], but only recently became available in Java. Therefore, using plain Java interfaces, we can plug as many crosscutting interfaces we want to cope with several other kinds of crosscutting concerns.

Abstraction and information hiding. The crosscutting concern update signalling implementation shown in Figure 5 does not provide a full reusable solution, due to its coupling between the pointcut specifications, defined within the **@After** advice, with some shape classes join points (e.g., method names). If another part of the program of even another program needs this update signalling functionality, but uses different method names, then one would need to change the interface UpdateSignalling in Figure 5 to accommodate the new names [11, 31]. To avoid this impact of such pointcut fragility in absorbing changes (such as may occur when the Shape classes change or when one adopts the crosscutting concern implementation to some other context) we allow the pointcut specifications to be defined in proper modules. This means that modules like Point or Line must be responsible for creating an interface to expose their implementation details. In the example of the shapes classes, since Point and Line share the same set of join point interceptions, we can expose their join points through another interface (as illustrated in Figure 6). Suppose later, a private method from Point should be exposed. In this case, we do not expose it through the interface

```
interface TransactionManagement{
  @Before("execution(* *(..))")
  void beginTransaction(){
    getPm().beginTransaction();
  }
  @AfterReturning("execution(* *(..))")
  void beginTransaction(){
    getPm().commitTransaction();
  }
  @AfterThrowing("execution(* *(..))")
  void beginTransaction(){
    getPm().rollbackTransaction();
  }
}
class HealthWatcherFacade
  implements IFacade, TransactionManagement {...}
```

Figure 7: An AspectJML crosscutting-based Java interface for HW's [39] transaction concern.

UpdateSignallingShapes; otherwise, we break the notion of information hiding [32]. To allow the private method exposure, we use the same approach to that in interface UpdateSignallingShapes, but declared within the type Point.

5.3 Benefits of our new notion of AOP

AspectJML overcomes the plethora of issues described in in Section 2 by using Formula 3. This formula allows an AOP language, such as AspectJML, to build on a standard OO language, such as plain Java, which results in a more symmetric and orthogonal model. An example of the benefits of this symmetry is that, since both method and advice declarations are named, AspectJML does not need the AspectJ pointcut adviceexecution to intercept other advice. (This pointcut is forbidden in AspectJML.) AspectJML enables programmers to reason about crosscutting modularity in the light of classical logic. This is very similar to reasoning about any invariant in a specification language such as JML.

One key design decision we took was to implement AspectJML based on the standard AspectJ. This results in a more straightforward adoption by any AspectJ programmer. Programming in AspectJML makes any programmer feel like programming with plain Java with annotations. This benefit avoids any semantic complications due to additional constructs and ultimately lack of adoption [42].

6 A FIRST EVALUATION OF ASPECTJML

Our preliminary evaluation to validate our new notion of AOP, with AspectJML, involves a web-based information system called Health Watcher (HW) [39]. The main purpose of the HW system is to allow citizens to register complaints regarding health issues. This system was selected because it was developed using AspectJ [17]. As a result, we refactored its transaction concern to add a crosscutting interface with AspectJML.

Figure 7 illustrates the implementation of the HW's transaction concern using the TransactionManagement crosscutting interface in AspectJML. The type IFacade denotes the remote interface to be implemented by the facade class HealthWatcherFacade. Also, any type that implements the TransactionManagement interface will have its methods considered as transactional methods and will be affected by the three pieces of AspectJML advice that add the transaction control behavior. Since the code quantifies over method executions, the interface TransactionManagement itself is reusable; it does not expose implementation details of any type (such as HealthWatcherFacade).

7 FUTURE WORK

We are currently developing AspectJML to support all the benefits discussed here. Our main focus now is building up and supporting the AOP community with an Eclipse-based tool and a dedicated online IDE. Any beginning programmer in AspectJML could learn our language using our online IDE (currently under development).

One direction, we are working on, is to simplify the use of the current AspectJ-like features with AspectJML. For instance, to prevent infinitely recursive advice application [2, 6, 43], as can happen in with AspectJ. Another direction, we intend to add in AspectJML is the notion of slicing [44]; this will be useful since in current AspectJML we do not support the open classes mechanism, also known as inter-type declarations in AspectJ [17]. Through slicing, programmers have the ability to use multi-dimensional separation of concerns, thus allowing regrouping of crosscutting concerns.

8 SUMMARY

We have revisited the main issues about the state of the art of AOP and modularity. Based on this analysis, we have been able to propose a new aspect-oriented programming formula that enables programmers to write modular crosscutting code and reason about modularity with all the properties of classical modularity. We realized this new (AOP) formula in the AspectJML language, a general-purpose, aspect-oriented extension to Java. Programming with AspectJML feels like programming in Java. It does not require any advanced syntax or any new concept, beyond AspectJ, to implement crosscutting modularity. We discuss and show how to cope with crosscutting structure using object-oriented hierarchies. With Hierarchical crosscutting in AspectJML, programmers can enable the modularization of crosscutting concerns with respect to classical modularity and modular reasoning. We also discussed a first evaluation of AspectJML, consisting of the modularization of a crosscutting transaction concern in a real system using AspectJML.

ACKNOWLEDGMENTS

We would like to thank Gregor Kiczales and the other creators of AOP. Without their work, this project would not be possible. Thanks to our AOP friends: Harold Ossher, Hridesh Rajan, Robert Dyer, Sudipto Ghosh, Paulo Borba, Fernando Castor, Márcio Ribeiro, and Roberta Coelho. They have been available for discussing about these issues since Modularity 2015. The work of Gary Leavens was supported in part by the US National Science foundation under grants CNS1228695 and SHF1518789.

REFERENCES

[1] Jonathan Aldrich. 2005. Open Modules: Modular Reasoning About Advice. In *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP'05)*. Springer-Verlag, Berlin, Heidelberg, 144–168.

[2] Eric Bodden, Florian Forster, and Friedrich Steimann. 2006. Avoiding infinite recursion with stratified aspects. In *In Proceedings of Net.ObjectDays 2006 (GI-Edition)*. Lecture Notes in Informatics, 49–54.

[3] Eric Bodden, Éric Tanter, and Milton Inostroza. 2014. Join Point Interfaces for Safe and Flexible Decoupling of Aspects. *ACM Trans. Softw. Eng. Methodol.* 23, 1, Article 7 (Feb. 2014), 41 pages.

[4] Jonas Boner. 2005. AspectWerks. (2005). http://aspectwerkz.codehaus.org/.

[5] Shigeru Chiba, Michihiro Horie, Kei Kanazawa, Fuminobu Takeyama, and Yuuki Teramoto. 2012. Do We Really Need to Extend Syntax for Advanced Modularity?. In *Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development (AOSD '12)*. ACM, New York, NY, USA, 95–106.

[6] Éric Tanter. 2008. Controlling Aspect Reentrancy. *Journal of Universal Computer Science* 14, 21 (2008), 3498–3516. Best Paper Award of the Brazilian Symposium on Programming Languages (SBLP 2008).

[7] Fernando Castor Filho, Nelio Cacho, Eduardo Figueiredo, Raquel Maranhão, Alessandro Garcia, and Cecília Mary F. Rubira. 2006. Exceptions and aspects: the devil is in the details. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering (SIGSOFT '06/FSE-14)*. ACM, New York, NY, USA, 152–162.

[8] Robert E. Filman and Daniel P. Friedman. 2000. *Aspect-Oriented Programming is Quantification and Obliviousness*. Technical Report.

[9] Alessandro Garcia, Cláudio Sant'Anna, Eduardo Figueiredo, Uirá Kulesza, Carlos Lucena, and Arndt von Staa. 2005. Modularizing design patterns with aspects: a quantitative study. In *Proceedings of the 4th international conference on AOSD (AOSD '05)*. ACM, New York, NY, USA, 3–14.

[10] David Garlan and Curtis Scott. 1993. Adding Implicit Invocation to Traditional Programming Languages. In *Proceedings of the 15th International Conference on Software Engineering (ICSE '93)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 447–455.

[11] Kris Gybels and Johan Brichau. 2003. Arranging Language Features for More Robust Pattern-based Crosscuts. In *Proceedings of the 2Nd International Conference on Aspect-oriented Software Development (AOSD '03)*. ACM, New York, NY, USA, 60–69.

[12] Jan Hannemann and Gregor Kiczales. 2002. Design pattern implementation in Java and aspectJ. *SIGPLAN Not.* 37 (November 2002), 161–173. Issue 11.

[13] William H. Harrison, Harold L. Ossher, and Peri L. Tarr. 2002. *Asymmetrically vs. symmetrically organized paradigms for software composition*. Technical Report. Research Report RC22685, IBM Thomas J. Watson Research.

[14] Charles Antony R. Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (1969), 576–580.

[15] Terry Hon and Gregor Kiczales. 2006. Fluid AOP Join Point Models. In *OOPSLA '06*. ACM, New York, NY, USA, 712–713.

[16] Gregor Kiczales. 2006. Interview with Gregor Kiczales. Software Engineering Radio: Episode 11. (2006). http://www.se-radio.net/2006/04/episode-11-interview-gregor-kiczales/.

[17] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. 2001. An Overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP '01)*. Springer-Verlag, London, UK, UK, 327–353.

[18] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. 1997. Aspect-oriented programming. In *ECOOP'97 Object-Oriented Programming (Lecture Notes in Computer Science)*, Mehmet Aksit and Satoshi Matsuoka (Eds.), Vol. 1241. Springer Berlin / Heidelberg, 220–242.

[19] Gregor Kiczales and Mira Mezini. 2005. Aspect-oriented programming and modular reasoning. In *Proceedings of the 27th international conference on Software engineering (ICSE '05)*. ACM, New York, NY, USA, 49–58.

[20] Gary T. Leavens, Yoonsik Cheon, Curtis Clifton, Clyde Ruby, and David R. Cok. 2005. How the Design of JML Accommodates Both Runtime Assertion Checking and Formal Verification. *Science of Computer Programming* 55, 1-3 (March 2005), 185–208. http://dx.doi.org/10.1016/j.scico.2004.05.015

[21] Gary T. Leavens and Peter Müller. 2007. Information Hiding and Visibility in Interface Specifications. In *International Conference on Software Engineering (ICSE)*. IEEE, 385–395. http://dx.doi.org/10.1109/ICSE.2007.44

[22] Gary T. Leavens and David A. Naumann. 2015. Behavioral Subtyping, Specification Inheritance, and Modular Reasoning. *ACM Trans. Program. Lang. Syst.* 37, 4, Article 13 (Aug. 2015), 88 pages.

[23] Gary T. Leavens and William E. Weihl. 1995. Specification and Verification of Object-Oriented Programs Using Supertype Abstraction. *Acta Informatica* 32, 8 (Nov. 1995), 705–778. https://doi.org/10.1007/BF01178658

[24] Barbara Liskov. 1988. Data Abstraction and Hierarchy. *ACM SIGPLAN Notices* 23, 5 (May 1988), 17–34.

[25] Barbara H. Liskov and Jeannette M. Wing. 1994. A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems* 16, 6 (Nov. 1994), 1811–1841.

[26] Bruce J. MacLennan. 1986. *Principles of Programming Languages: Design, Evaluation, and Implementation (2Nd Ed.)*. Holt, Rinehart & Winston, Austin, TX, USA.

[27] Bertrand Meyer. 1992. Applying "Design by Contract". *Computer* 25, 10 (1992), 40–51.

[28] Markus Mohnen. 2002. Interfaces with Default Implementations in Java. In *Proceedings of the Inaugural Conference on the Principles and Practice of Programming, 2002 and Proceedings of the Second Workshop on Intermediate Representation Engineering for Virtual Machines, 2002 (PPPJ '02/IRE '02)*. National University of Ireland, Maynooth, County Kildare, Ireland, Ireland, 35–40.

[29] Alberto Costa Neto, Arthur Marques, Rohit Gheyi, Paulo Borba, and Fernando Castor. 2009. A Design Rule Language for Aspect-Oriented Programming. In *SBLP '09: Proceedings of the 2009 Brazilian Symposium on Programming Languages*. Brazilian Computer Society, 131–144.

[30] Klaus Ostermann, Paolo Giarrusso, Christian Kstner, and Tillmann Rendel. 2011. Revisiting Information Hiding: Reflections on Classical and Nonclassical Modularity. In *ECOOP 2011 Object-Oriented Programming (Lecture Notes in Computer Science)*, Vol. 6813. Springer Berlin / Heidelberg, 155–178.

[31] Klaus Ostermann, Mira Mezini, and Christoph Bockisch. 2005. Expressive Pointcuts for Increased Modularity. In *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP'05)*. Springer-Verlag, Berlin, Heidelberg, 214–240.

[32] D. L. Parnas. 1972. On the criteria to be used in decomposing systems into modules. *Commun. ACM* 15 (December 1972), 1053–1058. Issue 12.

[33] Hridesh Rajan and Gary T. Leavens. 2008. Ptolemy: A Language with Quantified, Typed Events. In *ECOOP 2008 – Object-Oriented Programming: 22nd European Conference, Paphos, Cyprus (Lecture Notes in Computer Science)*, Jan Vitek (Ed.), Vol. 5142. Springer-Verlag, Berlin, 155–179.

[34] Hridesh Rajan and Kevin J. Sullivan. 2009. Unifying Aspect- and Object-oriented Design. *ACM TOSEM* 19, 1 (Aug. 2009), 3:1–3:41.

[35] Henrique Rebêlo and Gary T. Leavens. 2015. Enforcing Information Hiding in Interface Specifications: A Client-aware Checking Approach. In *Companion Proceedings of the 14th International Conference on Modularity (MODULARITY Companion 2015)*. ACM, New York, NY, USA, 47–51.

[36] Henrique Rebêlo, Gary T. Leavens, Mehdi Bagherzadeh, Hridesh Rajan, Ricardo Lima, Daniel M. Zimmerman, Márcio Cornélio, and Thomas Thüm. 2014. AspectJML: Modular Specification and Runtime Checking for Crosscutting Contracts. In *Proceedings of the 13th International Conference on Modularity (MODULARITY '14)*. ACM, New York, NY, USA, 157–168.

[37] Henrique Rebelo, Gary T. Leavens, Ricardo Massa Ferreira Lima, Paulo Borba, and Márcio Ribeiro. 2013. Modular Aspect-oriented Design Rule Enforcement with XPIDRs. In *Proceedings of the 12th Workshop on Foundations of Aspect-oriented Languages (FOAL '13)*. ACM, New York, NY, USA, 13–18.

[38] José Sánchez and Gary T. Leavens. 2016. Reasoning Tradeoffs in Languages with Enhanced Modularity Features. In *Proceedings of the 15th International Conference on Modularity (MODULARITY 2016)*. ACM, New York, NY, USA, 13–24.

[39] Sergio Soares, Eduardo Laureano, and Paulo Borba. 2002. Implementing distribution and persistence aspects with aspectJ. In *Proceedings of the 17th conference on Object-oriented programming (OOPSLA), systems, languages, and applications*.

[40] Friedrich Steimann, Thomas Pawlitzki, Sven Apel, and Christian Kästner. 2010. Types and modularity for implicit invocation with implicit announcement. *ACM Trans. Softw. Eng. Methodol.* 20, 1 (July 2010), 1:1–1:43.

[41] Kevin Sullivan, William G. Griswold, Hridesh Rajan, Yuanyuan Song, Yuanfang Cai, Macneil Shonle, and Nishit Tewari. 2010. Modular Aspect-oriented Design with XPIs. *ACM Trans. Softw. Eng. Methodol.* 20, 2, Article 5 (Sept. 2010), 42 pages.

[42] Kevin Sullivan, William G. Griswold, Yuanyuan Song, Yuanfang Cai, Macneil Shonle, Nishit Tewari, and Hridesh Rajan. 2005. Information Hiding Interfaces for Aspect-oriented Design. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13)*. ACM, New York, NY, USA, 166–175.

[43] Éric Tanter, Ismael Figueroa, and Nicolas Tabareau. 2014. Execution Levels for Aspect-Oriented Programming: Design, Semantics, Implementations and Applications. *Science of Computer Programming* 80, 1 (2014), 311–342.

[44] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. 1999. N Degrees of Separation: Multi-dimensional Separation of Concerns. In *Proceedings of the 21st International Conference on Software Engineering (ICSE '99)*. ACM, New York, NY, USA, 107–119.

[45] Marco Tulio Valente, Cesar Couto, Jaqueline Faria, and Sérgio Soares. 2010. On the benefits of quantification in AspectJ systems. *Journal of the Brazilian Computer Society* 16, 2 (2010), 133–146.

[46] Jia Xu, Hridesh Rajan, and Kevin Sullivan. 2004. Understanding Aspects via Implicit Invocation. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE '04)*. IEEE Computer Society, Washington, DC, USA, 332–335.