# Modular Programming with Aspectual Collaborations

## Thesis Proposal

Johan Ovlinger

## Abstract

The holy grail of software engineering is to tackle the three pronged problem of quickly building large, complex systems. We are once again able to choose only two out of three. This thesis investigates a system combining aspects—tackling complexity—with modules—tackling building large systems quickly—in an attempt to provide a solution for all three prongs.

We propose a programming construct called "Aspectual Collaborations", consisting of modules tailored to capture multi-object behavioral interactions, both at the explicit and implicit levels. Static and dynamic properties of the construct are explored and a sample implementation for Java provided. Finally, the construct is evaluated against the claim of supporting all three prongs of the problem.

## 1 Problem Statement

The fundamental problem in software development is that it is too hard *to write large, complex software quickly*. The various directions of research in computer science indicate differing beliefs about which approaches will ultimately lead to a solution. This thesis explores the intersection of the *Object-Oriented Programming* (OOP) and *Aspect-Oriented Programming* (AOP) styles of programming.

The key idea behind module systems is that by allowing the language to enforce encapsulation of implementation details behind an interface, clients of the implemented behavior are forced to remain oblivious to the details of its implementation—and hence immune to changes to the implementation as long as the interface is unchanged. Programs constructed from modules can thus be analyzed in these smaller pieces (the modules), rather than as monolithic structures. If a module is reused, time spent understanding it is amortized over all uses. Modules can be implemented separately, resulting in streamlined parallel development, and reused in other programs, resulting in an overall reduction in programming tasks, without affecting other modules of the program. Thus, modules allow larger programs to be constructed, and by enabling parallel development and reuse, also signi£cantly speed up this construction.

However, not all behaviors £t neatly into a module. Common examples are error handling and context-sensitive behavior (where behavior varies depending on where it was invoked). In general, such behaviors rely on, and modify, occurrences in other parts of the program. We may £nd that code in one module affects an invariant maintained by another module: for example, we may wish to mirror a directory layout in memory, without having to re-write either the i/o module or its client. The speci£cation is compact, but the implementation touches all disk operations.

The AOP manifesto can be summarized as "languages have modules, but programmers have concerns." The key insight/opinion is that a single kind of module boundary – be it along class, procedure, or group of classes – will not be suf£cient to capture all the kinds of concerns the programmer wishes to express. In order to cleanly modularize his concerns, the programmer needs to draw module boundaries in different directions under different scenarios.

Unfortunately, this also implies that the interaction points between modules become less regular: if we wish to modularize a behavior that is *implicitly*[1] invoked whenever a variable is modi£ed, we cannot use the same interface that we would had we wanted to modularize a set of classes exposing a subset of their members. Infact, the needs of ¤exible concern interaction seem to contradict the strict regimentation of module interfaces. Thus, many languages offer a trade-off: either a module system catering to different dimensions of concerns but with simple interactions between the modules, as in Hyper/J, or capturing complex interactions between concerns but offering conversely simpler modularity constructs, as in AspectJ.

This thesis investigates what happens if we want the best of both worlds: modules and aspects, and gives one data point of how we needed to de£ne the concepts to allow them to work well together. We design a language along these guidelines, provide an implementation using Java as a base OO language, and formally analyze a small model of the language using a [much] smaller language as the OO core. We aim to show that with suitable design, a module language interacts very well with aspect oriented features.

### 1.1 Terminology

Because we want to use the same terms to discuss both modular and aspectual features, we will use slightly different de£nitions than introduced in [19] and normally used in literature. Our de£nitions maintain the same ¤avor as the previous de£nitions, differing mainly in how terms invented for AOP can be retro£tted to older concepts like modules. For example, we refer to the interaction of two concerns as an aspect, while normally an aspect is considered to be an advice concern which is tightly coupled with some interaction speci£cation over some [more loosely] coupled base concern. The rest of this section describes terminology in greater depth. Sec-

[1] Why implicit invocation is interesting is covered in the next section.

tion 4.5 describes in more detail why we feel that a terminology change is justi£ed.

A **concern** is a somewhat nebulous de£nition, including all program entities that are involved with some part of the program. Examples of concerns are tracing, logging, class *is-a* and *has-a* graphs, or maintaining some invariant. The language provides **modules**, which may contain several concerns. We would like to have a concern contained in exactly one module, but concern boundaries may require more ¤exibility than the language provides. The mismatch between modules and concerns may force the concern to **crosscuts** several modules. We say that a language supports the **Separation Of Concerns** (SOC) when it has features that enhance the ¤exibility of module boundaries, or provides different kinds of modules to capture different concerns. Some examples of SOC include: the ability to package small multi-class behaviors into a module, where the £nal classes will be constructed from several modules; the speci£cation of how to resolve real-time constraints; and the succinct speci£cation of a traversal path through an object-graph.

As a consequence of the many different and overlapping concern boundaries, the interactions between them can be very complex, with program artifacts participating in several concerns simultaneously. It quickly becomes burdensome to require all interactions between concerns to be *explicit* in the code. From a practical standpoint, explicit concern interaction hampers reuse of concerns by hard-coding assumptions about the referenced concern. The key contribution of replacing explicit interactions between concerns with *implicit* ones is that the concerns can now be *oblivious* to each other, which is a step towards allowing them to be analyzed and understood in separation. On an aesthetic level, the interactions between concerns can be seen as a separate concern, which we should be able to specify separately. We refer to the *external* speci£cation of interaction between concerns as an **aspect**, and a programming language as **Aspect Oriented** (AO) if it supports both separation of concerns and the separate speci£cation of the aspects of their interactions.[2]

The points where concerns can interact—the points that aspects talk about—are called **join points**. The join point model of a language describes which kinds of information and control is provided to the programmer. Our model includes declarations in the program text—allowing class de£nitions to be enriched with additional methods or £elds—and regions in the execution of the program—allowing executions of a method to be monitored, and allowing one join-point to determine whether it is within the dynamic extent of another. Other possible join points include dynamic properties of the program, such as real-time constraints, or remote invocation considerations such as object discovery. AspectJ uses point-cut designators to declare sets of join points. Such sets only capture single join points; there is no way to create a set of join-point tuples. For example, the subject-observer pattern reasons about pairs of join points, rather than individual points. If we wish to advise several instances of the subject-observer pattern, we need sets of join-point tuples, rather than join-points. In order to maintain type safety, we require that each tuple in a join-point-tuple set have the same type. This makes it attractive to view the set as a **join-point table**, with an unknown number of rows, but where each cell in a column has the same type.[3]

A join point can have **advice** associated to it by an aspect. Examples of advice are to introduce program text or to intercede in the execution of the program point. Other possible advice also includes declaring static errors—allowing the advice to specify interaction pattern that should be ¤agged as static errors if they occur—and manipulating the type relationships of a collaboration.

The goal of separating concerns is to enhance both program maintainability, by increasing the clarity of code, and also to increase productivity, by separating out functional blocks into units that can be independently re-used. Using external aspects to control the interactions between concerns ampli£es both of these bene£ts by removing explicit inter-concern references. This improves reusability by removing hard-coded dependencies and maintainability by localizing interaction to the aspect.

Merely separating concerns' interaction into an aspect does not by itself promote reusable and maintainable code. Reuse and maintenance is hampered by several factors: Allowing concerns to interact at arbitrary points contradicts the **encapsulation** principles that allow modules to vary their internals as long as an external interface is maintained. Without encapsulation, aspects are tempted to encode assumptions about the base concern into the aspect—including names, data structures, and call-graphs. Any change to a concern can potentially invalidate the assumptions made by the aspect, increasing the maintenance costs of aspect oriented code. An example is hard-coding class- and member-names from the base application into the concern, which of course tightly couples the concern to be used in contexts that bind exactly those names in a similar manner. Likewise, encapsulation contradicts one of the fundamental assumptions of AOP: that the advising concern has complete understanding of the implementation of the base concern – not only *which* join points are involved in a behavior, but also *how* and *why*. When all that is visible via the interface are exports, how they are implemented, and how they call each other, also becomes opaque to advice. The obliviousness of concerns to aspects implies that there is no evidence of the aspect in the concern; we are reduced to searching all aspects to identify which may be invalidated by a given modi£cation to a concern. If any join point can have advice, and that advice can be from any aspect, the programmer must simultaneously understand all concerns and aspects that make up a program.

These defects can be circumvented by adding a ¤exible module system to protect and encapsulate concerns behind well de£ned interfaces.

## 1.2 Solution

We propose a ¤exible module language—*Aspectual Collaborations* (henceforth ACs)—tailored to interact well with aspect-oriented features. We add a strong encapsulation boundary to a group of cooperating classes—the collaboration—allowing the classes and their interactions to be analyzed as a unit. The encapsulation boundary allows the collaboration to specify explicit imports and exports: methods, £elds, and join points can be exported. [4] The strong

[2]Note that we are being intentionally vague about the [a]symmetry of this relationship. While our solution uses concerns mediated by a mutually external aspect, the most popular AOP language, AspectJ, organizes code asymmetrically into one base concern, which is modi£ed by external aspects that combine interaction speci£cations and the advising concern.

[3]The use of existential types allows seemingly dissimilar join points be viewed as having the same type.

[4]Support for join points in the interface is currently under development, and not yet implemented.

encapsulation boundary allows each collaboration to be compiled separately.

Collaborations can be composed, point-wise merging class de£nitions, allowing explicit imports and exports between composed collaborations, and advice to be attached to join points. In the terminology of Section 1.1, collaborations have two kinds of concerns (and hence also aspects): structural and behavioral.

### *1.2.1 Structural Concern*

The structural concern is captured by our module language. A number of **participants** make up a **collaboration**. Participants are like Java classes in that they declare members—methods or £elds—and form has-a and is-a graphs. Unlike Java classes, participants are also able to declare members as **expected**, so that member must be **provided** before the program is complete. Collaborations are composed by **attaching** one or more constituent collaborations to an output—or host—collaboration. The attachment process resolves differences in has-a and is-a graphs between the interacting collaborations, with the output collaboration providing the graphs that the attached constituent collaborations must be made to conform to. Additionally, attachment links provided members to expected members, and constructs the composed collaboration's interface by controlling which members and participants are exported or hidden.

A collaboration's structural concern consists of participant and member de£nitions. These are also the join points of the collaboration. Attachment is the structural aspect, whose advice speci£es how a collaboration's declarations should be added to the output collaboration, and how its explicit member imports and exports should be linked to other collaborations' members.

### *1.2.2 Behavioral Concern*

The behavioral concern deals with how one collaboration can intercede in the execution of code in another collaboration, with neither collaboration aware of the other. The join points of the behavioral concern are method executions, with optional conditions on when the join point is active. The advice we offer for method executions is to reify them as £rst-class objects and then allowing advising methods to use the API on these objects to control how the execution proceeds. The advising methods are able to inspect and affect the arguments to the advised method execution, whether it is executed at all or several times, and similarly control the result of the execution.

Since structural aspects modify the typing relationships of structural concerns, and since structural concerns type behavioral concerns, we can only reason about behavioral aspects in the context of structural ones. Once structural aspects have united types for the two concerns, we are able to determine whether the behavioral concerns and advice are well typed in the new typing environment. We are currently in the process of formalizing the constraints on structural aspects necessary to guarantee that well typed behavioral aspects and concerns remain well typed when interpreted in the context of structural aspects.

Behavioral and Structural concerns may interact at the same point: a behavioral concern may have a method's execution as a join point, while the structural concern has references to that method as a join point. Such situations are handled just like concerns: the behavioral concern is interpreted in the context of the structural. The details of such interactions can be complicated. For example, an expected method can be a behavior join point, and thus its execution can be advised. When a method is provided to the expected method, it is clear that all executions of the expected method should be advised, but it is less clear whether executions of the provided method (which is now the same as the expected) should be affected by the same advice.

One extreme case of interactions between structural and behavioral concerns are Join Point Tables. Each column in the table contains behavioral join points with the same type, while each row is a set of related join points, sharing a common set of types. In order to make all cells in a column have the same type while retaining the ability to write powerful advice, we existentially quantify the types that differ from row to row. These types can then be used during attachment, allowing advising collaborations to be instantiated for the types of each row: thus, join-point tables carry both structural and behavioral types. Join-point tables can manually speci£ed, or generated manually by matching against the participant graph. Tables can be exported and expected from collaborations, but like join points have no representation at runtime.

### *1.2.3 Behavioral / Structural vs Dynamic / Static*

This section hopes to clarify the de£nitions of closely related terms that we are *not* rede£ning. We have used the terms "Structural" and "Behavioral" to classify kinds of concerns and aspects, while the terms "Static" and "Dynamic" are in common use in current aspect oriented discourse. The terms seem to refer to similar concepts, but there is actually no overlap in their de£nitions.

**Static / Dynamic** refers to the distinction of when an aspect is applied: before runtime, or at runtime. Runtime applied aspects are often assumed to be under programmatic control, but this is not the only possibility: for example a quality-of-service aspect can be externally selected based on network congestion and applied dynamically.

**Structural / Behavioral** refers to what the concern talks about. Our implementation is completely static, yet has both structural and behavioral concerns. A more dynamic language might well have structural concerns that were applied dynamically.

Join points can exist both at runtime and/or compile time, depending on the language. A dynamic AOP language might allow join points to be passed as £rst-class values and advised at runtime, or perhaps only allow activation or deactivation of advice at runtime, without treating join points as £rst class values. Our implementation is static: we can not pass nor advise any kind of join point at runtime. The API for behavioral advice *does* instantiate method execution join points as Java objects, but only as representations of the join point – the instantiated join point cannot be advised, nor will it effect other instances of the same join point. However, the join point instance *is* a £rst-class object, which allows a great deal of ¤exibility.

## 1.3 Thesis Contribution

Aspectual Collaborations combine powerful modularity constructs with aspectual features, supporting external composition, separate checking and compilation, and strong encapsulation, while also capturing both explicit and implicit interactions between modules.

The focus of the thesis will be a thorough investigation of the semantics of Aspectual Collaborations. We show that not only is

modular encapsulation compatible with aspectual programming, but that encapsulation is in fact a powerful tool for structuring aspects. The thesis provides one data point describing how to balance the power of aspects and the control of modules.

Adding encapsulation interfaces to code comes at a price: a module must a-priori specify which join points it will expose to outside advice, while languages without encapsulation—such as AspectJ—can advise any point in the base code while the base code remains completely oblivious that any interaction will take place at all. However, the gains from our approach outweigh the costs:

**Modular reasoning.** The biggest gain attributed to modularity is that the program can be reasoned about in smaller units. We show that ACs allow modular reasoning to be applied to aspectual programs—both the base code and the advice that is applied to it.

**Separate analysis.** The interfaces that allow modular reasoning are precise enough to allow ACs to be analyzed and compiled separately from their uses. We develop a set of rules that identify correct composition speci£cations. We believe that these rules will maintain speci£c static guarantees, such as type safety and non-invasiveness (as per [10]), which we state informally but do not prove. We will however formalize and prove these properties on a smaller language that avoids the complex semantics of Java.

**External composition of aspects.** By expressing composition externally to the constituent ACs, we allow the programmer to see the high-level architecture of the program without needing to inspect the code. The integration of aspects and modules in Aspectual Collaborations extends this understanding to aspectual behavior. Without external composition of modules, we would need to comprehend the entirety of the program to understand what will happen when some method is invoked.

**Generic, type-safe aspectual behavior.** We develop an API that allows the advice to manipulate the join point (including executing it multiple times, not at all, and storing the join point in a variable for later invocation). Our API is carefully crafted to allow the manipulating code to be compiled separately from the join point being manipulated, while remaining type safe. This includes the guarantee that there will be no typecast failures in the generated adapter code necessary for the API to function.[5]

**Join-point tables.** An important feature is to decouple the ability to advice a method execution from the ability to call the method directly: this allows invariant-insensitive methods to be advised without endangering the invariant. Thus, we need to have join points represented directly in the interface of modules, rather than exposing the methods they represent. All other AOP systems we know of focus on manipulating sets of join points—pointcuts, in normal nomenclature. We have discovered that in order to achieve type-safe reuse of generic aspectual behavior, all related advice must be attached in the same context, necessitating use of sets join-point tuples, rather than unstructured sets of join points. Thus our interfaces contain what we call Join-Point Tables (JPTs), and account for them in our precise but smaller model of ACs.

JPTs additionally allow us to directly express how state should be shared between advice to join points in a set. Thus, we are able to clearly differentiate between state that is local to one join point, to a join-point tuple, to a table, or global for the whole application.

[5]We mention this as AspectJ's whole-program compiled advice does not have this property. See Section 3.1.

**Terminology.** Our partial rede£nition of "aspect" and "concern" allows us to use the same terminology to discuss module systems, multi-dimensional separation of concerns, and aspect-oriented programming. This new terminology allows us to precisely state our position in the continua of these concepts.

# 2 Collaborations

This section illustrates the modular features of collaborations: the ability to import and export methods and variables, and how collaborations are composed. We start with an example that highlights structural concerns, moving on to behavioral concerns for the following examples. The rest of the report will discuss ACs as they are currently designed. In parallel with this report, we are investigating the precise semantics of how join points can be added to the interface of a collaboration. This will likely signi£cantly enhance the match speci£cations of Section 2.2.2.

The Aspectual Collaboration language is purposefully kept minimal (for example, eschewing "before" and "after" advice for behavioral join points for the more general "around"). A small, orthogonal language allows us to keep the language implementation—and its semantics—small and easily understood. This allows the user to predict what is going to happen when a language feature is used in a non-standard way, and more importantly to understand why something failed to work when it goes wrong. The intent was to layer a prettier language ontop of this base, but we have found that the minimal language is surprisingly easy to program in.

## 2.1 Aspectual Collaborations Outlined

An Aspectual Collaboration consists of a **participant graph**, where the nodes are Java-class like entities called participants, and the edges are *is-a* and *has-a* relations. The set of participants is closed; it can be extended only by recompiling the whole collaboration.

Each participant has zero or more members. A member is a £eld or a method, obeying the normal semantics of Java. In addition to the member modi£ers offered by Java classes, participants can have three additional orthogonal member modi£ers:

- **expected** members are deferred members that, like **abstract** members, must be provided before the collaboration can be executed. These, however, differ from **abstract** members in that both methods and £elds can be expected, and, more importantly, in that an expected member does not inhibit class instantiation, and is provided to the participant directly, rather than by overriding it.
- **aspectual** methods are able to intercede in – advise – invocations of other methods. When an aspectual method is attached to advise a host method, all invocations of the host method is intercepted, rei£ed as a method call object, and passed to the aspectual operation as an argument. The aspectual method can then control and modify the details of the call, until returning control to the original caller. An aspectual method may be constrained to work with only a more limited set of signatures, thereby gaining access to the arguments to- and returned value of- the method execution.
- **exported** members are visible outside the collaboration. When creating a composed collaboration, any members that should remain visible must be explicitly re-exported. Non-exported members become hidden behind the encapsulation interface of the AC. Note that we will want to export all ex-

**Listing 1. A simple host package**

```
1 package variables;
2 class Vars {
3   String foo;
4   Baz bar;
5 }
6 class Baz {
7   Vars var;
8 }
```

pected members that have not been provided, as otherwise they will not be visible to subsequent attachment speci£cations, dooming the collaboration to never be executable.

These annotations are applied at the member—rather than participant—level for reasons of practicality rather than fundamental design. If we wanted to use imported participant usefully, we would anyway need to specify signatures of members the importing module needs to reference on the imported participants. Furthermore, we would also need to also import participants mentioned transitively by the signatures of the imported members. We would then need to apply some programming pattern like Jiazzi's mixin and open class patterns in order to be able to compose participants referencing imported behavior and the imported behavior into one exported participant. We avoid these obstacles by instead specifying imports and exports at the member level.[6]

Unlike other module systems, we choose to derive the signature of an AC from in-line annotations in its declaration, rather than an external signature applied to the declaration. The signature of an AC consists of the exported and expected members of the collaboration, along with the participants on which these members are de£ned. However, the names expected and exported are confusingly similar, so we often refer to these as the *required* and *provided* interfaces, respectively.

A collaboration is either compiled directly from source code, or else composed from already compiled collaborations. We call the resulting collaboration the output collaboration, and the already compiled collaborations from which it is composed, the constituent collaborations. The composition is akin to static linking, by which we mean that a change to a constituent collaboration will *not* be re¤ected in the output collaboration until it has be re-composed. This is merely an implementation choice, and nothing in the language design would hinder a more dynamic approach.

## 2.2 Collaborations and Structural Concerns

Listing 1 is a package consisting of two classes. The only difference between a trivial collaboration and a package is that a collaboration is a unit of compilation and is thus closed to subsequent addition of participants, while a single class can be compiled to augment an existing package. Since we will not be adding classes to the package, it can also be claimed to be a collaboration with two participants.

A slightly less trivial collaboration is adviceSetGetAttribute in Listing 2. Its HasAttribute participant de£nes a pair of `set` and `get`

[6]Collaboration-private participants are easily achieved by omitting them from the type mapping phase of collaboration attachment. This is only a possibility for participants that don't appear in the exported signature of the collaboration.

**Listing 2. De£ning generic setters and getters**

```
1  collaboration adviceSetGetAttribute;
2  participant HasAttribute {
3    expected AttributeType aName;
4    public void set(AttributeType aName) {{
5      this.aName = aName;
6    }}
7    public AttributeType get() {{
8      return aName;
9    }}
10 }
11 limited participant AttributeType;
```

methods [7] for an expected attribute `aName`, which will only be provided later. If adviceGetSetAttribute is composed with another collaboration, and `aName` is mapped to a concrete £eld, the effect will be to introduce a setter and a getter for the expected £eld. Composition is speci£ed by attachment clauses.

The **limited** keyword (line 2.11) allows for an external class (such as String), for which all modi£cations are prohibited, to play the role of AttributeType. Limited participants could hypothetically also be mapped to primitive types such as `int`: this would require a small amount of automatic modi£cation to the compiled participants, as they are represented as Java `.class` £les containing bytecode that treats primitive and Object types differently, and would additionally restrict how such a participant could be used.

### 2.2.1 Attachment

A composite collaboration is created by combining several constituent collaborations, and specifying (a) the pointwise mapping of constituent participants to output participants, and (b) how the members that thus end up on the same participant should be provided and exported. The composition speci£cation controls the structural concern of our application. Syntactically, this is expressed in an *attachment* speci£cation, and we say that the details of how *one* constituent is mapped to the output collaboration is its attachment. [8]

In greater detail, a composite collaboration is composed from one or more constituent collaborations, of which one is denoted the *base* collaboration.[9] The base collaboration is special in that it will play the role of skeleton for the output collaboration, with all of its members exported by default, rather than unexported, as is the default for the other constituent collaborations. This asymmetry is a compromise that allows us conveniently to model adding advice to a base program while retaining a core module language that maintains encapsulation principles. Once built—either by copying from the base, or by compiling the in-line speci£ed participants—the output collaboration is decorated by attaching the remaining constituents collaborations as per the attachment speci£cations.

[7]The double braces around the method bodies should be read as if they were a single brace. Their sole purpose is to allow our implementation to avoid parsing full Java, and are included in our presentation to indicate that all examples are verbatim runnable code.

[8]Currently, the attachment speci£cation also manages behavioral concerns, but that is set to change in the next version of our system.

[9]The base collaboration can also be speci£ed in-line, and future work includes efforts to derive a suitable base collaboration automatically from the constituents and attachment speci£cations.

**Listing 3. Attaching setters and getters for `foo`**

```
1  collaboration adviceSetGetFoo;
2  extends variables ;
3  attach adviceSetGetAttribute {
4    Vars += HasAttribute {
5      provide aName with foo;
6      export set as set_foo ;
7      export get as get_foo ;
8    }
9    String += AttributeType ;
10 }
```

The adviceSetGetFoo collaboration (Listing 3), for example, is created by designating variables as the base collaboration (line 3.2), thus creating the output collaboration from its contents. The remaining constituent collaboration (adviceSetGetAttribute) is attached (line 3.3) to introduce and subsequently export a setter (line 3.6) and getter (line 3.7) for the `foo` field (line 3.5) of Vars (line 3.4).[10]

The structural aspect expressed in the attach clause (lines 3.3–10) controls the interaction of the two structural concerns: the base from variables, and the constituent collaboration adviceGetSetAttribute. Line 3.4 maps HasAttribute to Vars, which has the effect of decorating the latter with three members provided by the former (the methods `get` and `set`, and the variable `aName`). Participants are mapped with the **+=** operator, which also redirects any references of the inserted type to the destination type (HasAttribute and Vars, respectively). Inspection of Listing 1 shows us that `Vars.foo`, which is the concrete field provided to `aName`, is of type String. In order to satisfy the structural aspect's static consistency requirements with respect to the host application, we need to map AttributeType to String (line 3.4). Luckily, AttributeType has been marked **limited** (line 2.11), which means that it cannot be decorated at all, but is allowed to be mapped to a non-collaboration type such as String.[11]

Once decorated, the members now on participant Vars can be linked, allowing structural advice to affect explicit member references between concerns. Line 3.5 provides the variable `foo` to the expected variable `aName`. Lines 3.6 and 3.7 export the `set` and `get` methods to more accurate names. Since `aName` was not exported, it is not visible on the resulting participant.

### 2.2.2 Multiple Attachment

A collaboration can be attached several times. For example, we may want getters and setters for several variables. Listing 4 illustrates how we can attach adviceSetGetAttribute twice to provide getters and setters for both the variables `foo` and `bar` (on participants Vars and Baz, respectively).

While multiple attachments can be specified in this way, duplicating the attachment clause for every variable is unsatisfactory. One improvement would be to abstract an attach clause over formal variables (making an attachment template), and to instantiate it for a set of variable-binding tuples. Even better would be to generate these tuples automatically by matching against the class graph and solving constraints. This corresponds to using a *point-cut designator* to generate a set of join points (a point cut). This section describes how such a mechanism works, how our needs differ from those provided by point-cut designators, and how it can be used to attach collaboration to several parts of a program.

A template attachment clause needs one or more textual holes to be filled in with appropriate values to generate a complete attach clause, which is then evaluated to attach the constituent- to output-collaborations. The mapping of template holes to text is called a MT. By applying the attachment template to a set of MTs, we can generate a set of attach clauses from a template. [12]

The important difference between combining Match Templates with an attachment template and attaching advice to a set of Join Points (JPs) is that the former allows a whole collaboration to be simultaneously attached, maintaining its internal structure and shared references, while advising a JP merely talks about one join point in the concern. A MT allows us to write advice that can observe and intercede in the interplay between several methods and variables.

Writing out lists of MTs manually is clearly an improvement over writing out whole attachments, but only by a constant factor. A better option is to generate MTs by evaluating a match clause against the output collaboration's structure. Listing 5 shows an example: the attach clause (lines 5.7–14) looks very similar to Listing 3, but with some identifiers replaced by variable references (within < ... >). The identifiers thus replaced control the name of the variable for which we are generating the getter and setter, and the variable's type. Each application of this abstraction generates an attachment indistinguishable from one written explicitly. Indeed, for the variable bindings

$$(\mathsf{HasFields} \mapsto \mathsf{Vars}, \quad \texttt{fieldName} \mapsto \texttt{foo}, \quad \mathsf{FieldType} \mapsto \mathsf{String})$$

the attachment is equivalent to that in Listing 3, and

$$(\mathsf{HasFields} \mapsto \mathsf{Vars}, \quad \texttt{fieldName} \mapsto \texttt{foo}, \quad \mathsf{FieldType} \mapsto \mathsf{String})$$
$$(\mathsf{HasFields} \mapsto \mathsf{Vars}, \quad \texttt{fieldName} \mapsto \texttt{bar}, \quad \mathsf{FieldType} \mapsto \mathsf{Baz})$$

corresponds to Listing 4. The match specification (lines 5.3–6) will generate the latter.

To generate a set of Match Templates from a **match** clause, it is interpreted as a subgraph constraint against the output collaboration. A tuple of variable bindings is generated so that the graph in the match clause matches the output collaboration—where *matches* is defined as a subgraph relationship. The set of all distinct match tuples is generated.

We have already seen a template attachment to attach getters and setters for all instance variables in the output collaboration, but to use it we need to generate tuples binding the name of the variable, the class it is defined in, and its type. lines 5.3–6 declare a match clause that produces such a tuple from every [visible] instance variable in every participant of the output collaboration. The keyword **role** is used in place of **participant** to highlight that this is not a declaration of participants, but rather a pattern match against them. The result is that addAllSettersGetters will have getters and setters for each of its variables `foo`, `bar`, and `var` (recall Listing 1 for the definitions of the variables).

[10] In the cases that there are many fields and only some of the fields should be provided with getters and setters a more expressive matching is available.

[11] By redirecting references from AttributeType to String, we achieve type parameterization as a degenerate case of attachment [1, 2, 33].

[12] MTs are precursors to Join-Point Tables. MTs generate textual results that are tightly bound to attachment templates, which is not the case for JPTs.

**Listing 4. Attaching two getter methods.**

```
1  collaboration adviceGetTwo;
2  extends variables ;
3  attach adviceSetGetAttribute {
4    Vars += HasAttribute {
5      provide aName with foo;
6      export get as get_foo ;
7      export set as set_foo ;
8    }
9    String += AttributeType ;
10 }
11 attach adviceSetGetAttribute {
12   Vars += HasAttribute {
13     provide aName with bar;
14     export get as get_bar ;
15     export set as set_bar ;
16   }
17   Baz += AttributeType ;
18 }
```

**Listing 5. De£ning all setters and getters.**

```
1  collaboration addAllSettersGetters ;
2  extends variables ;
3  match {
4    role <HasFields> {
5      <FieldType> <£eldName>;
6    }
7  } attach adviceSetGetAttribute {
8    <HasFields> += HasAttribute {
9      provide aName with <£eldName>;
10     export set as set_<£eldName>;
11     export get as get_<£eldName>;
12   }
13   <FieldType> += AttributeType;
14 }
```

The particular matching clause in Listing 5 contains only variable bindings, but in general a matching clause will contain hardwired names as well, signi£cantly constraining the number of generated matches. More complicated matches can match chains of getters, every pair of getters and setters (illustrated in Section 2.3), or in general any constraint expressible by a partially labeled subgraph of the static program.

In the current state of the language, the Match Templates generated by match clauses must be immediately used to attach a collaboration. We are currently working out a more advanced semantics which would allow information similar to that in MTs to be named and exported from Aspectual Collaborations. The MTs will then be subject to manipulation such as concatenation, £ltering, and narrowing. Sections 3.2 and 5.1.1 describe our progress towards this goal.

## 2.3 Collaborations and Behavioral Concerns

So far, we have seen examples of combining collaborations, linking expected and provided members, and controlling the interface of a collaboration through exports, which constitute the *structural* aspects of Aspectual Collaboration's module language. This section will focus on the other half, the *behavioral* aspects, where we are able to intercept method calls without either invoker or invokee being aware.

Listing 6 implements a simple collaboration which maintains some state. Two aspectual methods (lines 6.4 and 8) keep count of how many times `inc` is called between `reset`s. We'll use this collaboration to introduce the concepts of aspectual methods and sharing of state between attachments of a collaboration.

The novelty of Listing 6 are the **aspectual** methods `reset` and `inc`, and—more precisely—how they are invoked. Aspectual methods are seldom invoked directly, but instead intercede when the execution of the program reaches a Join Point that the aspectual method advises. In our prototype system, we focus on method execution as the only kind of behavioral join point. Other possible points include variable access and object instantiation. It is also possible to attach conditions to Join Points: a common example is AspectJ's *c¤ow*, which states that a Join Point only £res if the program execution is within the dynamic extent of a certain method.

Unlike **exported** and **expected** members which can be wired together to form explicit references between collaborations—allowing information and control to cross collaboration boundaries—**aspectual** methods and behavioral join points can be wired up to form *implicit* references between collaborations. The difference is that in the explicit model, the *calling* collaboration controls the interaction, while in the implicit model, the *called* collaboration is in control. In the terminology of Filman and Friedman [11], aspectual methods allow the host collaboration to be oblivious to invocation of such aspectual behavior. We prefer to think of aspectual methods as *intercessionary*, as they have the ability to intercept and advise the invocation of advised methods.

***Method Interception Through Late Binding*** Instead of aspectual methods, it could seem that late binding and overriding would be able to simulate method interception. Indeed, the ability to invoke the overridden method from the overriding one is very similar to how we want to be able to control the execution of the intercepted method call in the advice of a join-point, and this is incidentally the technique we foresee our reimplementation with Jiazzi as a back end (Section 5.1.4) to use.

There we run into some dif£culties if we try to use this technique directly. (a) We would be unable to expose a method to be advised without exposing it to direct invocation. The visibility that allows us to override the method also allows us to call it directly, if only from within the overriding method. (b) Our advice becomes tightly bound to the exact signature of the method we are advising, losing the genericity offered by the API of rei£ed method invocations. As a consequence, it becomes problematic to allow the same advice to advise a set of anonymous join points: all must now have exactly the same signature. (c) Late binding works only if the runtime type of the object is a subclass of the class with the overriding method. Since the base code, whose method is being intercepted, is oblivious to advice, and hence the name of the subclass, we must use a programming pattern that allows late binding of class names. This is rather contradictory to the "oblivious" nature of the base code.

### *2.3.1 Aspectual Methods*

Aspectual methods are like normal methods, but have a stylized signature that starts with the keyword **aspectual**:

**aspectual** RetVal methname(MethM arg),

where both RetVal and MethM are user chosen participant names that are either unde£ned or de£ned locally to the collaboration. The

**Listing 6. Count incs between resets.**

```
1  collaboration counter;
2  participant Counted {
3    int count;
4    aspectual ResetR reset(ResetM e) {{
5      count = 0;
6      return e.invoke();
7    }}
8    aspectual IncR inc(IncM e) {{
9      count++;
10     return e.invoke();
11   }}
12 }
```

user thus names the types that reify the intercepted method call and its return value.We'll assume these names for the following discussion.

Reifying the intercepted method call as MethM allows us to treat it as £rst class value: storing it in a variable and passing it as an argument to other methods to invoke, in addition to controlling when (or if) to proceed with the advised method call.Similar arguments hold for the bene£t of reifying the returned value as RetVal. To maintain type safety, each aspectual method in the collaboration needs distinct types for its rei£ed method call and return value.[13] Otherwise, an aspectual method could attempt to return a RetVal object obtained from proceeding with some other advised method execution that generated some other return type.

The rei£cation participants corresponding to the user chosen names RetVal and MethM, are automatically generated by `acc`. The two generated participants' default API consists of only one expected method on MethM:

**expected** RetVal invoke().

The default RetVal participant has an empty API.[14] By having rei£cation participants be completely ignorant of the signature of the method call and return value they represent, an aspectual method can advise methods of any signature (including raised exceptions or void returns). Encapsulation guarantees that this holds even when the aspectual collaboration has been attached to a host.

In addition to specifying exports and provides, the attach clause of a collaboration also sets up which methods should be advised by which aspectual methods. When a method with aspectual advice is invoked, the system intercepts the method invocation. The intercepted method invocation is rei£ed as a MethM object, and passed as the sole argument to the aspectual method. The aspectual method can invoke the host method at any time, multiple times or not at all (returning either a default return value, or perhaps a return value from a previous invocation). Both method thunks and return value objects are plain Java objects, and can be stored in data structures, passed as arguments, or even persisted to the £le system. The aspectual method is guaranteed eventually to return a rei£ed return-value object (by its signature), which is unpacked to reveal the real return value of the intercepted method call. This real value is then returned to the caller, from which point the program continues oblivious to the advice.

[13] Actually, not only does every aspectual method need to have distinct types, but every *attachment* of an aspectual method needs to have distinct types. Our implementation simply maps RetVal and MethM to fresh participants for each attachment.

[14] Optionally, the MethM's API can be extended to provide access to [some] arguments of the call. This of course reduces the general applicability of the aspectual method, as it can now only wrap methods whose argument-list contains these types. Similarly, we can also gain access to details concerning result of invoking the wrapped method, including its return value or whether it returned normally or threw an exception.

**Listing 7. Counting all the getters and setters.**

```
1  collaboration usecounter;
2  extends addAllSettersGetters;
3  match {
4    role <Part> {
5      <FType> get_<name>();
6      void set_<name>(...,<FType>,...);
7    }
8  } attach counter {
9    <Part> += Counted {
10     export count as count_<name>;
11     around get_<name> do inc;
12     around set_<name> do reset;
13   }
14 }
```

### *2.3.2 A Counter Example*

Listing 6 presented an AC for maintaining statistics between calls to two methods. To illustrate its use and attaching aspectual methods, we present usecounter in Listing 7, which extends the collaboration varsgns. By the **extends** clause, we know usecounter will contain the structure imported from addAllSettersGetters: two participants, three variables, and a getter and setter for each variable.

The **match** clause of usecounter (lines 7.3–7) illustrates a clause that constrains some of the graph structure. The variable `<Part>` will be matched against all participants that have at least one pair of getter and setter methods. We identify as such a pair any two methods whose names start with pre£xes `get_` and `set_` have the same name suf£x, and talk about the same type. The constraint that the methods have the same name suf£x is expressed by the variable `<name>` occurring in both lines 7.5 and 6. The setter method may have arguments of other types as well, but the return type of the getter method must occur at least once in the arguments to the setter, which must return void. Each match against the output collaboration will bind the variables `<Part>`, `<FType>`, and `<name>`, to which the attach clause (lines 7.8–14) is applied. The three generated tuples are:

(Part ↦ Vars, `name` ↦ `foo`, FType ↦ String)
(Part ↦ Vars, `name` ↦ `bar`, FType ↦ Baz)
(Part ↦ Baz, `name` ↦ `var`, FType ↦ Vars)

Listing 7 attaches collaboration counter three times. It is speci£ed in line 7.11 that when a get method is invoked, `inc` should advise the invocation. Likewise, line 7.11 attaches `reset` as advice to setters. Each pair of getter and setter has a distinct `count` variable, which are exported as `Vars.count_foo`, `Vars.count_bar`, and `Baz.count_vars`, respectively. This makes the calculated statistics accessible to the rest of the program.

The result of the collaboration is that each time a getter (`Vars.get_foo`, `Vars.get_bar`, or `Baz.get_baz`) is called on the resulting collaboration, aspectual method `inc` (line 6.8) intercedes, incrementing the `count` variable corresponding to the invoked get-

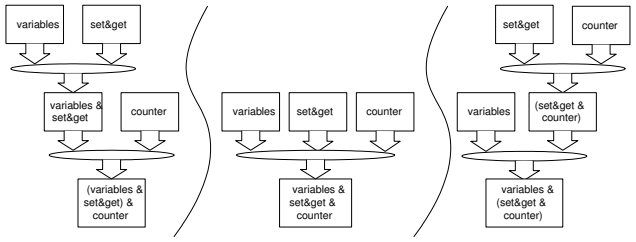


**Figure 1. Different ways to compose collaborations.**

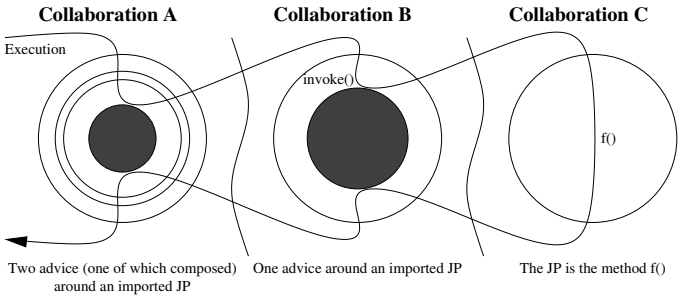


**Figure 2. Advice as skins of an Onion.**

ter. Likewise, each setter causes its corresponding `count` to be reset to zero, via `reset` (line 6.4).

Had we used wild-card syntax to generate a join-point set of all methods that matched `get_*`, and likewise for `set_*`, we would have been hard pressed to £gure out which join point in the former corresponded to a join point in the latter. Match Tuples makes this correspondence explicit, and is key to our ability to advise the interplay between several members.

## 2.4 Composition and Advice Interaction

Figure 1 graphically shows a number of ways to compose three collaborations. From left to right, we can (a) attach collaborations incrementally to the base (variables), as shown earlier in this section, (b) attach all collaborations in one monolithic attachment, or we can (c) combine two collaborations £rst, and then attach that composite to the base.

These organizational options follow from our underlying module system. However, things become interesting when we consider how to order advice attached to a join point. Previous approaches have left the order unspeci£ed, or speci£ed as a hard-wired relation between particular sets of advice (AspectJ aspects, in particular).

Exploring the options open to us, we could do something similar, expressing the ordering of advice at the granularity of their containing concerns. Instead, we opt to allow ordering to be expressed at the granularity of individual advice.

We have three scenarios, not coincidentally similar to Figure 1: (a) attach advice to a join point, incrementally, in different attachments, (b) attach a number of advice to a join point in the same attachment, or (c) compose advice £rst, and then attach the composed advice to a join point.

We can intuitively order these cases by viewing advice attached to a join point like the skins of an onion with the join point at the center, as in Figure 2. Each layer represents wraps its inner layer, deciding if and how program control should proceed inward. Later attachments of advice are added to the outside of the onion, creating an easy-to-understand model of how the advice interacts. This interacts very well with the external nature of our aspects: if we need to have a careful ordering of advice to a set of join points, we merely need to import the advice and join points into the same attachment, and there specify the ordering. This approach has been successfully used by the QuO [28] project to order advice for managing quality-of-service for a video stream from an unmanned aerial vehicle. Alternately, behavioral advice can be composed £rst, and then [perhaps later] attached to a join point.

# 3 Analysis

While the Aspectual Collaboration language is relatively small and orthogonal, it extends and interacts with Java. Java is anything *but* a small language, with semantics that are infeasably large and complex to work with. As such, we are hindered from proving any properties about our language, as that would involve taking in the whole of Java.

Instead, we hand-wavily de£ne the semantics of ACs, arguing rather than proving that the type system is sound. To add weight to this argument, we are investigating the semantics of how ACs interact with a a much smaller language than Java. If we prove that combination sound, it would suggest that the combination of ACs and Java may be sound by the same principles.

## 3.1 Type Safety

Collaborations are separately compiled by a standard Java compiler, and as such are well typed according to the rules of Java. In this section we argue that composition of collaborations results in a collaboration which is *as type safe* as the constituents.

Java's type *Soundness* theorem states that a well-typed Java program will run (inde£nitely, or to completion), or fail either in an exception (including nullpointer dereference) or a cast. Since exceptions and casts are dynamic properties, which may occur on some execution paths but not others, and the paths taken will clearly be dependent on the details of composition, we cannot state any theorems about which errors will occur dynamically. However, we *can* talk about *where* in the program these errors can happen, and make statements about how composing collaborations can affect these locations.

Given two collaborations which are both well typed according to Java's type checking rules, a successful composition of the two will generate a new collaboration which is also well typed. In other words, while the composed collaboration may uncover previously unexercised casting and nullpointer errors, it will not increase the locations where such errors could occur.

To argue this point, we note that

- [Rename] A systematic renaming of participants and members of a collaboration should not affect its typing.
- [Extend] Adding a set of members to the participants of a collaboration should not affect its typing, given that the added members form a well-typed set themselves, and they do not name-clash with the existing members.

- [Encapsulate] The composition of two collaborations will not introduce relationships between types that did not exist in the constituent collaborations.
- [Generated] Inspection of the code generated by attachment reveals that it is devoid of casts, and can be analyzed as a well-typed collaboration itself.
- [Compose] Thus, composing two well typed collaborations will result in a well typed collaboration that does not have any additional locations where dynamic errors can occur.

Notably, AspectJ's generic around advice does not have the [Generated] property. AspectJ has a somewhat unconventional genericity mechanism for around advice:

- If the advice returns void, it can match any signature. The language takes care of propagating the result of the `proceed` call to the caller of the advice.
- If the advice returns Object, it can match any signature. The `proceed` call will return a result Object to the advice, which must return that (or some other object) as the result of the invocation. This allows the advice to return completely different results than the advised method. Primitive types and void are automatically wrapped in wrapper objects inside the advice, and correspondingly unwrapped and returned after the advice.

  The *body* of the advice is type-checked against each join point. It is important that the body be analyzed rather than a method call, because the implicit upcast to Object to match the return type would invalidate the *stupid-cast*[15] check performed by the Java compiler.

  Unfortunately, all that is required to circumvent this check is an explicit upcast to Object, which will result in a casting error in the code generated by AspectJ. Combined with separate compilation and abstract aspects, it is quite possible to write code that will never cause a static type error, yet cause casting errors at run time, and require a change to the [possibly third-party] abstract aspect to £x.
- All other return types much exactly match the signature of their use.

The possibility for casting errors are inherent in all APIs that return a common supertype and require the client to downcast. We make a point of the use of casting in AspectJ's advice for four reasons: it points out the danger of using in-band signaling—it is now not possible to restrict advice to wrap only methods that return void (and similarly Object); it also creates a false sense of security, as some cases are caught, but not others; the errors occur in code that is beyond the programmer's control; and the type error is delayed until the aspect is used, which may occur in a situation where the aspect's development has been closed (bought from a third-party, or version branch).

Hyper/J also uses casts (to downcast arguments passed in as an Object array), but the casts are optional and must be inserted by the programmer, rather than being automatically generated. Thus, any casting errors will be obvious and £xable by the programmer.

In addition to a composed program not causing runtime cast errors not present in an uncomposed program, we also want it not to allow casts to succeed that would otherwise have failed. If a participant type is mapped to two different host classes in two attachments, those two host classes should not be castable to each other.

AspectJ's interfaces will likely cause casting in implementation of interface methods.

[15] See [16] for a good discussion on the subtlety of stupid casts.

## 3.2 Semantics of Aspectual Units

To enable a complete soundness proof, we will develop a semantics for Aspectual Collaborations with a much simpler language substituted for Java. The semantics are currently incomplete, as they are in a state of ¤ux from being used to analyze several missing features in ACs. When these features have been resolved, the next task will be to prove useful properties of the semantics. Finally, the semantics will guide the process of adding crosscuts (here called join-point tables) to the interface of ACs.

### *3.2.1 Basic Model*

The two main issues that we need to model are how advice is hooked up to a join point, and how join-point sets interact with modular encapsulation boundaries. A straight forward semantics can be achieved by modeling advice as a program transformation and join points as locations in the program text. The resulting semantics closely model the implementation strategy of `acc`, making it more likely that we have implemented the language we meant to. Unfortunately, a transformation-based semantics becomes very awkward when attempting to model any other operation than adding advice to locally visible methods. Notably, it is very dif£cult to put sets of such join points into the interface of a collaboration.

The solution is to model the join points directly in the semantics. We introduce a simple language over join points, allowing them to be de£ned, conditioned, and combined. This interacts with encapsulation in that we must maintain some structure between related join points, so that rather than putting sets of join points in the interface of modules, we need sets of *tuples* of join points.

A tuple captures the complete context needed to attach advice to the related join points. Part of the context captured by the tuple are type bindings for types that are generic in the advice: Since every attachment of the advice will have different bindings, a join-point tuple consists of a set of program locations, and type bindings for the generic types of the advice. Type safety requires that each tuple in a set contain the same type of join points; it is convenient to visualize the set of tuples as a join-point *table*, where each row is a tuple, and each column represents a particular join-point type. The type of a table tells us the number of columns and types of the join points they contain, but not how many rows a table has. It is infeasible to expect to be able to £nd several tuples of join points that have exactly the same types; however, it *is* feasible to £nd join points with similar types; for example, all pairs of getter and setter methods will have the types $() \to \tau$ and $\tau \to ()$, respectively, for some type $\tau$ which depends on the variable being affected. Thus, the types in a table can be quanti£ed over some *existential* types ($\tau$ above) differing for each row.

Note that ACs do currently have a the more simplistic module interface, where only participants and members are in the interface. A concrete bene£t of a careful analysis of the semantics will be to facilitate the task of adding join-point tables to AC interfaces.

A common example are the getters-and-setters of Listing 2. In Listing 6, we see a module that has advice that wants two related join

points: each such set of related join points will share a counter. It would be straight-forward to add a generically-typed variable to the counter, in order to ignore redundant sets. In our use of the counter collaboration (Listing 7) we explicitly set the type of the £eld for each attachment of the collaboration. Had we exported the three attachments as a join-point table, we also need to capture the different types that the type-variable `<FType>` takes. The join point table captures these as values for existential types.

### *3.2.2 Bene£ts*

The main reason for developing a semantics of to model the interaction between advice and encapsulation is to prove that our minimal semantics are sound: a well-typed program guarantees that unforeseen type-errors do not occur at runtime (in our toy language, that covers all possible type-errors, but other languages may have more powerful type systems that allow casting errors to occur at runtime in a type-safe program).

The proof that our toy language is sound will strengthen our argument that the real implementation—using Java rather than a small functional language, but a very similar module-system—at least *can* be sound. Dif£cult cases discovered in the process of working through the proof will guide where we are most careful when adding the join-point tables to ACs.

## 4 Related Work

We compare our work against previous work in the AOP £eld, the £eld of modules, that of composable components or classes, and also the £eld of modeling.

## 4.1 Aspects and Concerns

AspectJ and Hyper/J are the two most mature and well known AOP languages. Each focuses on a different facet of AOP: adding aspects to the base language and separation of concerns, respectively.

**Hyper/J** [31] emphasizes structured software engineering rather than aspectual features. Rooted in the subject-oriented programming paradigm [14], Hyper/J supports merging and decomposition of separately speci£ed class hierarchies: *hyperslices*. Unlike AspectJ, Hyper/J treats the base and the aspectual behaviors symmetrically, that is, a hyper-slice can model both the base and the aspectual unit.

Hyper/J's composition mechanisms focus on merging of structural aspects, but limited support for behavioral aspects are also possible. This is achieved via the several ways to implicitly combine methods, each one with differing constraints on the methods that can be combined.

Hyper/J's main restrictions are that the implicit connections are quite restricted in what can be combined, and that it does not attempt to enforce any encapsulation, instead exposing the whole slice to composition.

**AspectJ** [18, 19] takes a completely different tack to managing concerns than Hyper/J. Rather than view concerns as modular units, AspectJviews them as language features. An aspect is a class-like construct whose behavior is scattered throughout the program.

As a language feature, AspectJ aims at programmer convenience and expressiveness rather than reuse or software architecture. Concerns in AspectJ have no encapsulation whatsoever, but as a trade-off join points can be very succinctly expressed.

The AspectJ team de£ne an aspect as a "modular unit of cross-cutting implementation". Note that this is not how we de£ned an aspect in the introduction. We prefer to say that an aspect is "the external speci£cation of the interaction between several oblivious concerns". Sullivan et. al [30] point out that the AspectJ de£nition of an aspect is relative to the module system and programming language in question, and that aspects satisfy Parnas' [27] information-hiding criterion.[16] They go on to demonstrate a bound on the modularity offered by AspectJ, showing that AspectJ is not modular for the interaction of aspect instances (in the language of AspectJ) — not unsurprising, as AspectJ lacks the notion of aspect instances (collaboration attachments, in our language). However, it is unclear what the exact parameters used in this study were, as Aspectual Collaborations are able to solve the problem posed in Sullivan et. al with only one collaboration attachment. One possibility is that structural modularity provide ACs crucial expressiveness to avoid multiple attachments that would otherwise be necessary.

**Semantics** of around advice for method execution join points are modeled in [32], but does so for an untyped language. This work is extended to model the semantics of AspectJ's compilation in [23]. The two papers above do not consider separate compilation. Method call interception is investigated by [20], but does not investigate around advice, which avoids describing how to control an intercepted method call. The interceptions he models are amenable to separate compilation. It appears that the three papers do not model grouping join points or sharing state between advice.

## 4.2 Module Systems

Module systems are effectively the dual of AspectJ, offering powerful encapsulation and reuse support, but no support for aspectual or concern-oriented features. However, just as AspectJ can still achieve reuse without encapsulation, it is possible to achieve some aspectual features without tool support—as in Hyper/J's implementation of an around method. Indeed, we are currently investigating the possibility to use a third-party module system as a back end, rather than our own solution, in order to free up developer resources.

**Jiazzi** [24] is the implementation of Units [12] for Java. Jiazzi reuses Java's core composition feature—inheritance—and the Open Class pattern to construct the resulting classes from partial implementations. Late binding is used to allow mutually recursive dependencies between modules. A Jiazzi implementation of the caching example would likely look very similar to the Hyper/J version, but instead of capturing the original method explicitly and then relinking the resulting class to swap in the cached version instead of the original, In Jiazzi we would specify that we expected the cached method to be declared on a superclass, and then proceed to override it, calling the original method with a super call.

It is interesting to note that although developed completely independently, the composition we use for structural concerns in `acc` and the modular composition provided by Jiazzi are strikingly similar. The main differences are how the £nished classes are assembled

[16] The summary of Parnas presages the *weaving* implicit in AOP—for performance rather than expressibility—suggesting that modules should *composed* into base language procedures, rather than *containing* such procedures.

(Jiazzi favors inheritance, while we manually combine and link participant `.class` £les) and the fact that we favor intrinsic typing for collaborations, while Jiazzi allows the extrinsic signature of a unit to be reused for several unit implementations.

**Mixin-Layers** [29] represent a collaboration as a layer of mixin classes. This layer is treated as a unit, so that the interface to the mixin layer is a set of superclass imports, and a set of class exports. Mixin layers can be composed creating composite layers, allowing modular construction of complex programs. Both Jiazzi and Aspectual Collaborations generalize layers, in that both can represent layers as a programming pattern, but can also represent other forms of modular construction.

## 4.3 Component Systems

Component Systems can be seen as a dynamic form of module system, where the modular units, components, have a representation in the program, and are often instantiated and composed programmatically at runtime, rather than than module's declarative composition at compile time. Furthermore, the result of linking components typically does not generate a new type, but rather links existing component instances together to construct complex behaviors from simpler ones.

**Dynamically Attachable Aspectual Components** share a common history with Aspectual Collaborations. Adaptive Plug&Play Components [25] were developed in parallel and collaboration with the author's tentative work on Class Graph Views. The shared history explains the similarity between ACs and the work that has evolved from AP&PCs, including the use of separate participant graphs for each component, and attachment speci£cations to map components to each other. Notably, Aspectual Components [21] suggests the speci£cation of generic "replace" methods to implement method interception. However, neither AP&PCs or Aspectual Components has a convincing argument for type safety, and are described in a very informal manner. Typing issues are addressed in [26] by introducing an asymmetric system that is claimed to be type-safe through virtual types.

Only ACs have even a partial implementation as described. A variation [15] on Aspectual Components has been implemented using the interpreted language Lua. The implementation offers both dynamically and statically attached components, but seems not to be able to advise methods that return results, nor is it type safe.

All the above systems are dynamic, in that a component attachment is rei£ed at runtime, and can by dynamically applied or removed. Unfortunately, the dynamic nature causes a few problems, in that the necessary wrapping and unwrapping of objects to transport them from the base's type kind to the component's type kind needs to be exposed to the programmer. Although automatable to a certain extent, the inability to declare two types equal means that the programmer needs to be aware that wrapping is taking place. For example, to transparently pass a container of objects between the base and component worlds, the container needs to be traversed to £nd and translate all sub-objects from one kind to the other. Such translation would be necessary for all objects crossing from one type kind to the other.

**ArchJava** [4, 3] is a a modern example of a component system. Components communicate with each other over named sets of methods, called ports. Component instances can be connected both statically or dynamically, and in both cases, the system guarantees that components only communicate to their neighbors, ensuring *communication-integrity*. However, the authors point out that this applies only to method invocations—shared object references can still be propagated through the system allowing communication to pass via the shared object.

ACs are not able to conveniently express, nor guarantee communication integrity for dynamic component connection, but are able to quite well for the static case. Using encapsulation, we are able to make statements not only which components a component is able to talk *to*, but more strongly which components it can talk *about*.

If a component doesn't import another component's type, direct communication between them is impossible. This applies additionally to auxiliary classes and objects that are passed between components. If a component has only a limited view of a class (for example omitting a sensitive £eld), then we can statically guarantee that this £eld cannot be directly manipulated by the component. If a component does not know about a class at all, it cannot communicate via objects of that type at all.

## 4.4 Modeling Languages

Moving away from implementation to the modeling arena, we see some connections to modeling efforts.

**Composition Patterns** [7] adds the concept of composition patterns to UML [5]. From the implementation suggestions in [6], we believe that Aspectual Collaborations will be a very good match to implement models described in this notation. It is unclear whether Composition Patterns capture multiple attachments of a collaboration, and how sharing of members between such attachments would be expressed.

**Catalysis** methodology [9] has a strong emphasis on modeling collaborations. Catalysis uses a *common model of attributes*. In comparison, we use a participant graph, and have built-in support to express aspectual decompositions.

## 4.5 Change in Terminology

As noted earlier, some terms we use are subtlety different from how they are normally used in AOP discourse. The terms *aspect* and *concern*—and consequently de£nitions thereof—stem from AspectJ and Multi-dimensional Separation of Concerns, respectively. At the time, the connections between aspects, separation of concerns, and composable modules were not fully understood, and the terms' de£nitions thus overlap somewhat. This section will present slightly modi£ed de£nitions of these terms, with the goal of making them orthogonal.

The de£nition of aspect from [19] is:

> With respect to a system and its implementation using a GP-based language, a property that must be implemented is:
>
> **A component,** if it can be cleanly encapsulated in a generalized procedure (i.e. object, method, procedure, API). (...)
>
> **An aspect,** if it can not be cleanly encapsulated in a generalized procedure. Aspects tend not to be units of the system's functional decomposition, but rather to be properties that affect the performance or semantics of the components in systemic ways. Examples of aspects include memory access patterns and synchronization of concurrent objects.

Which is summarized in a statement about the goal of AOP [17] as:

> The job of AOP is to turn a tangled and scattered implementation of a crosscutting concern into a well-modularized implementation of a crosscutting concern.

while [31] doesn't present a direct de£nition of concerns, they do describe requirements on their decomposition:[17]

> Many common maintenance and evolution activities result in high-impact invasive modi£cations. (...) These somewhat diverse problems are due (...) to limitations and unful£lled requirements related to *separation of concerns* [27]. Our ability to achieve the goals of software engineering depends fundamentally on our ability to keep separate *all* concerns of importance in software systems. All modern software formalisms support separation of concerns to some extent, through mechanisms for decomposition and composition.
>
> (...)
>
> Decomposition according to concerns along a single, dominant dimension is valuable, but usually inadequate. Units pertaining to concerns in other dimensions end up "scattered" across many modules and "tangled" with one another. Separation according to these concerns is, therefore, not achieved.

Notice that all de£nitions involve behavior that has been modularized, and which has some interaction with the rest of the system: an aspect is a concern whose interaction is crosscutting in the current module system, while a separated concern is a concern whose interaction is de£ned along multiple dimensions of (de)composition.

Thus, the common vocabulary to describe both kinds of decomposition would separate *what* a concern does, from *where and how* it interacts with other concerns. Hyper/J correctly realizes this, and has separated concerns and their interactions into HyperSlices and HyperModules, respectively. AspectJ combines a concerns and how it interacts with the system into what it calls an aspect.

Whether the interaction of a concern is crosscutting depends both on the language it is written in and the details of it's interaction: A logging concern for one method is hardly crosscutting. Therefore, we choose to separate out what makes both Multi-Dimensional Separation of Concerns and Aspect-Oriented programming powerful, and call that the aspect: the *speci£cation of concern interaction*, possibly in a crosscutting way.

Our terminology allows us to analyze both Hyper/J and AspectJ with the same language. An AspectJ aspect can be decomposed into the composition of a behavioral concern (the advice) and one of our aspects (how it interacts with the rest of the program: the join points). Additionally we can compare separation of concerns with aspect orientation: Hyper/J provides strong support for structural aspects, but fairly weak support for behavioral aspects, while AspectJ is the opposite, providing weak structural aspects, but strong behavioral aspects.

# 5 Conclusion

Much of what the thesis will contain has been worked out in detail already, but there remain a few factors that need to be cleared up, and a few speculative branches we can investigate as time allows.

[17] The quoted reference has been moved into our bibliography.

## 5.1 Remaining Tasks

At the programming level, there remains much feasibility testing to be done. While we have presented a number of language features, we have not characterized what problems they solve well. Ongoing work with BBN on the AIRES project allows us to evaluate the usefulness of Aspectual Collaborations in a number of real-world scenarios. Additionally, a self-hosting reimplementation of `acc`, the AC Compiler, using Jiazzi as the back end will simultaneously highlight the similarities of the underlying module language, the additional power afforded by aspectual programming, and validate ACs for medium scale projects.

### *5.1.1 Join points in Collaboration Interfaces*

Perhaps the biggest remaining task is to include join points into the interface of an AC.

ACs, as described in this report and implemented, do not support the export or import of Join points (JPs). This causes two problematic effects: (1) Currently, a method can be advised iff it it is visible: the ability to advise a method is equivalent to the ability to provide it to be explicitly invoked. (2) Furthermore, the advising attachment must refer to advised methods by name.

Neither of these is a desirable property; a method may maintain an invariant that only holds when invoked from within a context, and requiring an attachment to name—and hence know how many—all advised methods restricts it to work under exactly these circumstances: an additional method cannot be logged without adding it explicitly to the interface of the exporting collaboration **and** the attachment of the importing collaboration.

The obvious solution to the former effect (1) is to put join points representing the static ability to advise executions of some method into the interface of a collaboration. This would allow a collaboration to export the ability advise a method without making the method itself visible. The latter effect (2) suggests that instead of a number of single, named, join points, a named set of join points would be a better alternative. By writing the attachment as a set comprehension, it would apply equally to singleton sets as well as sets of unknown size.

However, a set of join points is not suf£cient for our needs.[18] As we mentioned in the discussion of multiple attachments (Section 2.2.2), we need to specify the attachment of several advice simultaneously in order to maintain shared state between the advising methods. For example, if we have several `push-pop` method pairs, it would not suf£ce to make one set of push methods, and one set of pop methods, as this would break the cohesion of the pairs.

***Join-point Tables*** The set in the interface needs to contain enough information so that the attachment clause can perform the multiple simultaneous attachments that Match Templates (MTs) are able to specify. The same information that we can gain by interpreting the MT in the context of an attachment template must be put into the set in the interface. We will need this information when advising the join points in the table, but will be unable to interpret textual names against an attachment, as we won't be able to see the attachment (due to encapsulation) nor the context in which it was written.

[18] In general, there could be several differing sets of join points in the interface, but for simplicity we'll refer to *the* set.

Rather than Match Templates consisting of textual strings, the set needs to contain tuples of Join Points.

Each tuple in the set can then be an argument to an attachment template to generate a complete attachment. Because the exact details of the join points in the set is [purposefully] obscured, we need to type the join points so that we can be sure that our attachment will be well typed.[19] The type of a join point is merely the details it is willing to export to advice: types and values of exposed arguments and return values, and possible exceptions will be re¤ected in the join point's type. The type of a tuple of join points is the tuple of the types of the join points it contains. All tuples in the set comply to the declared type of the set. This allows us to statically verify whether a set of tuples will be compatible with an attachment template.

Since the set contains tuples complying to a single type, it is convenient to view the set of join-point tuples as a **Join-Point Table** (JPT). Each row in the table corresponds to one tuple, while each column is one position in the tuple. We statically know the number and type of columns, while the number of rows and the contents of the cells (actual methods to be advised) is knowable only at compile time by elaborating all compositions.

As a concrete example, when the MT that is generated by the match clause in Listing 7:

| | | |
|---|---|---|
| (Part ↦ Vars, | `name` ↦ `foo`, | FType ↦ String) |
| (Part ↦ Vars, | `name` ↦ `bar`, | FType ↦ Baz) |
| (Part ↦ Baz, | `name` ↦ `var`, | FType ↦ Vars) |

is interpreted against that listing's attachment template, it speci£es that each pair of methods: `Vars:get_foo` and `Vars:set_foo`, `Vars:get_bar` and `Vars:set_bar`, and `Baz:get_var` and `Baz:set_var`, should be advised by `inc` and `reset`, respectively, as one attachment. [20]

Expressing this as a join-point table, we cannot rely on textual concatenation to generate names like `get_bar` from the match `name` ↦ `bar`. Rather, we detail each method explicitly:

| ∃Part, | ∃FType | Part :: FType exec(), | Part :: void exec(FType) |
|---|---|---|---|
| Vars, | String | Vars::String get_foo(), | Vars:: void set_foo(String) |
| Vars, | Baz | Vars:: Baz get_bar(), | Vars:: void set_bar(Baz) |
| Baz, | Vars | Baz:: Vars get_var(), | Baz:: void set_var(Vars) |

The text above the horizontal line is the signature of the JPT, telling us that it exports two participant types, and two method-execution join points whose types reference those participant types. The rows below the line are the actual values of the cells. These will not be visible to the programmer unless he simulates the compiler manually. We have not decided on a syntax for generating JPTs, but it is reasonable to assume that something close to the current matching clauses can be used.

***Manipulating JPTs*** However, the signature of this JPT is overly precise for our purposes. To attach `inc` and `reset` we don't need access to the precise signature of the methods in the join point, and if we want to concatenate this with another JPT, their signatures need to match exactly. We will need to provide operators to allow JPTs to be manipulated without revealing the contents of their rows.

[19] If there were a mismatch, this would be discovered at compile-time, as all attachments are resolved before run-time, but without types the programmer would be unable to discover the problem other than by trial-and-error compilation.

[20] We'll ignore the exported variable for now (c.f. line 7.10).

We expect that these operations will be needed: (1) Reorder and drop columns from the table. (2) Filter rows according to some sort of condition. The exact nature of the condition is yet unclear. (3) Hide details about the join-point type of a column to make it more general. This may require exporting more participant types from the JPT, but can also result in fewer exports.

The third of these operations will allow us to transform the JPT to the more generic signature shown below. Notice that only the signature has been changed: the join points are untouched by hiding details of the type. Furthermore, since we no longer reference Part2 in the signature, we no longer export it. In this signature, neither join point publishes any type information, returning the *any* type (_), and taking no arguments.

| ∃Part | Part :: _ exec(), | Part :: _ exec() |
|---|---|---|
| Vars | Vars::String get_foo(), | Vars:: void set_foo(String) |
| Vars | Vars:: Baz get_bar(), | Vars:: void set_bar(Baz) |
| Baz | Baz:: Vars get_var(), | Baz:: void set_var(Vars) |

The types Part1 and Part2 are existential types scoped over each row. Thus we know that for each row in the JPT, there exists a mapping of collaboration participants to the existential types of the JPT, so that the type of the join points in the JPT signature correspond to the join points in the row. It is necessary to export the name of the participant, as otherwise type mapping becomes dif£cult and the signature of the join point potentially ill-formed. This JPT signature tells us that both execution join points of a row are de£ned on the same class.[21]

***Attaching advice to a JPT*** To use the JPT, we would attach its de£ning collaboration, and write the attachment as a set comprehension. The £nal syntax has not been worked out, but it might look something like:

```
1  ...
2  attach hasJPT {
3     ... // exports and provides from hasJPT
4    for (ThePart, getter_jp , setter_jp ) in hasJPT.theJPT {
5      attach counter {
6        <ThePart> += Counted {
7          around <getter_jp> do inc;
8          around <setter_jp> do reset;
9        }
10     }
11   }
12 }
```

***Remaining Remaining Tasks*** It is unclear what sorts of join points we want to have in a JPT. We have discussed method-execution join points above, but we may want to be able to reference additional members from advice attached to the JPT. For example, if we have advice that veri£es an invariant, we want to intercede in the execution of potentially invariant-breaking methods, but also access any variables that form the variant. The signature of a JPT that gives us access to executions of a getter and the variable whose value is being gotten, might be: ∃Part, ∃Var | Part :: Var exec(), Part :: Var field Indeed, we may want to nest JPTs inside JPTs. This would allow us to very conveniently tackle 1e-to-*n* relationships like caching advice. Caching

[21] ∃Part1, ∃Part2 | Part1 :: _ exec(), Part2 :: _ exec() would not tell us anything about the relative locations of the methods.

advice needs to have 1 cached method, but *n* invalidator methods. Nesting JPTs will allow us to specify several cached methods, each with its own set of invalidators.

### *5.1.2 Constructors*

When collaborations are attached, we compose the participants of the constituent and base collaborations per the attachment specification. During this process, members of the constituent participants are systematically renamed, along with their references, to avoid name clashes with other members. Unfortunately, some members cannot be renamed without breaking the semantics of the program. Methods overriding non-participant methods—such as `toString` overriding a method from java.lang.Object—cannot be renamed without breaking overriding, as we are unable to rename the external method being overridden. Luckily, by overriding an external method, the method would have been necessarily exported anyway, so the situation only becomes problematic when a name-clash occurs. This is likely to be a relatively uncommon problem, so we can just force the programmer resolve the name-clash manually at composition time.

Unfortunately, constructors face a similar, but trickier, problem. Constructors—both instance and class initializers—are intricately tied into the semantics of the JVM, with hardwired method names and special instructions to invoke. The first complication is that every participant will have constructors, which means that we are guaranteed to get name clashes for every composition. Furthermore, there is a predefined order in which constructors will be invoked: superclasses are initialized before subclasses. For the purposes of discussion, we'll summarize this order as the invariant that a participant's constructor must be entered before any of that participant's methods are invoked on the object.

A partial solution is to transform constituent constructors into normal methods, and build a new "spine" of constructors where each built constructor calls the super-constructor first, then invokes the original constructor from each participant.

Unfortunately, we cannot guarantee the desired invariant, as a constructor is free to invoke any method, including an expected method that is provided from another participant. This would require that the providing participant had already been initialized, which we cannot guarantee in general as participants can be mutually dependent, implying that a safe order of initialization isn't guaranteed to exist. However, the situation is acceptable, as a similar interaction happens in plain Java, when a constructor calls an instance method which has been overridden in a subclass: that subclass's method will be invoked on an uninitialized object.

Finally, constructors can be overloaded, leaving us with an unclear situation as to which constructor from one participant to combine with a constructor from the other. Fortunately, encapsulation of collaborations allows us to avoid the combinatorial explosion that would have resulted had we needed to cater to all possible pairings. We are faced with a choosing a heuristic or always making the programmer decide: we choose to combine constructors with identical signatures by default, and allowing the programmer to optionally specify other pairings.

Jiazzi [24] has a similar restriction, requiring all constructors in an inheritance chain to have exactly the same signature.

### *5.1.3 Semantics*

While we have convinced ourselves of the static safety of ACs, a more convincing argument is necessary. This argument will take the form of a fully worked out semantics for a version of ACs with Java replaced by a very simple language. Both ClassicJava[13] and FeatherweightJava[16] are candidates. The former provides a more accurate reflection of the intricate features of Java, but is thus also more complex to work with than the latter.

By proving a simpler version of ACs type sound, we will show that the concepts are not inherently unsafe, and by carefully choosing how we simplify ACs to achieve this proof, we will *argue* that ACs are probably sound as well, at least at the language design level.

### *5.1.4 Jiazzi as a Back End*

The current back end of `acc` is implemented using the *Byte Code Engineering Library* [8]. It manually composes classes by inserting bytecode from constituent collaborations into the output collaboration's `.class` files. Serendipitously, the `.class` file format [22] makes such moving very simple, as all member and class references are indirected via handles (constant pool entries, in the JVM terminology), so all that is required to combine two compiled classes is to copy all constant pool entries from the destination into the source constant pool, and adjusting their referencing byte codes to use their new indices.

Ultimately, the implementation details of the byte-code munging (as we call the composition of compiled classes) do not contribute towards the novel aspects of ACs, and the maintenance of the munger consumes resources better used in other parts of the implementation. Jiazzi is a mature implementation of the Units module system, and chooses to compose classes via inheritance rather than direct composition. As we note in Section 4.2, Units in general—and Jiazzi in particular—have a notion of module composition that is compatible with how we compose structural concerns.

By using Jiazzi as a back end, we get the double benefit of clarifying the exact nature of our contribution, and also reducing out programming burden by reusing a mature implementation. Unfortunately, using Jiazzi is not side-effect free: class composition via inheritance will be unable to provide expected *fields* in collaborations. However, the workaround of using getters and setters is fairly well accepted by the community, so this is not a significant shortcoming. Other than that, we don't foresee any change in the surface syntax or semantics stemming from such a move.

## 5.2 Summary

We have outlined a thesis that will present a module system tailored to reusable aspect-oriented programming in Java. It improves on current state-of-the-art by combining the reuse and modular reasoning capabilities of module systems with the encapsulation of concerns offered by aspect-oriented programming. To our knowledge, no other aspect-oriented language offers separate compilation or significant reuse.

Aspectual Collaborations share design with a much smaller language of modules and events. This smaller language's type system will be proven sound, and we argue that by extension ACs share this property. However, the complexity of Java makes proving this claim infeasible.

ACs also present a novel join point model, which is integrated into the module language to allow points inside a module to be advised without exposing the details of those points in the interface of the module.

The price for this power is that interfaces must be explicitly declared and composed for each AC. We expected this to be a signi£cant programming burden made worthwhile by the power and safety it enabled. However, initial tests suggests that the burden is comparable to writing reusable aspects in AspectJ or working in Hyper/J, which is signi£cantly less burdensome than expected.

# 6 References


[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, NY, 1996.

[2] O. Agesen, S. N. Freund, and J. C. Mitchell. Adding type parameterization to the java language. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* in *Special Issue of SIGPLAN Notices*, volume 32, pages 49–65, 1997.

[3] J. Aldrich and C. Chambers. Architectural reasoning in archjava. In *European Conference on Object-Oriented Programming*, 2002.

[4] J. Aldrich, C. Chambers, and D. Notkin. Archjava: Connecting software architecture to implementation. In *International Conference on Software Engineering*, 2002.

[5] G. Booch, J. Rumbaugh, and I. Jacobson. *The Uni£ed Modeling Language User Guide*. Object Technology Series. Addison Wesley, 1999. ISBN 0-201-57168-4.

[6] S. Clarke. Designing reusable patterns of cross-cutting behavior with composition patterns. In *The Workshop on Multi-Dimensional Separation of Concerns in Software Engineering at ICSE*, 2000.

[7] S. Clarke and R. Walker. Composition patterns: An approach to designing reusable aspects. In *International Conference on Software Engineering*. ACM Press, 2001.

[8] M. Dahm. Byte code engineering library. http://jakarta.apache.org/bcel/manual.html.

[9] D. D'Souza and A. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, 1998.

[10] E. Ernst. Syntax based modularization: Invasive or not? In *Position papers from the workshop on Advanced Separation of Concerns at OOPSLA*, 2000.

[11] R. Filman and D. Friedman. Aspect-oriented programming is quanti£cation and obliviousness. In *The Workshop on Multi-Dimensional Separation of Concerns in Software Engineering at ICSE*, 2000.

[12] M. Flatt and M. Felleisen. Units: Cool modules for hot languages. In *ACM Conference on Programming Language Design and Implementation*, pages 236–248, 1998.

[13] M. Flatt, S. Krishnamurthi, and M. Felleisen. A programmer's reduction semantics for classes and mixins. In *Formal Syntax and Semantics of Java*, volume 1523 of *LNCS*. Springer Verlag, 1999. expanded version of the POPL 98 paper, corrections in TR 97-293.

[14] W. Harrison and H. Ossher. Subject-oriented programming (A critique of pure objects). In *Proceedings OOPSLA '93, ACM SIGPLAN Notices*, pages 411–428, Oct. 1993. Published as Proceedings OOPSLA '93, ACM SIGPLAN Notices, volume 28, number 10.

[15] S. Herrmann and M. Mezini. Combining composition styles in the evolvable language lac. In *Advanced Separation of Concerns Workshop at ICSE*, 2001.

[16] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* in *Special Issue of SIGPLAN Notices*, pages 132–146, 1999.

[17] G. Kiczales. Is cross-cutting a good way to de£ne aspects?, 2001. email message archived at `http://aspectj.org/-pipermail/users/2001/000723.html`.

[18] G. Kiczales, E. Hilsdale, J. hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *European Conference on Object-Oriented Programming*, 2001.

[19] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming*, 1997.

[20] R. Lämmel. A semantical approach to method-call interception. In *Proceedings of the International Conference on Aspect-Oriend Software Development*, 2002.

[21] K. Lieberherr, D. Lorenz, and M. Mezini. Programming with Aspectual Components. Technical Report NU-CCS-99-01, College of Computer Science, Northeastern University, Boston, MA, March 1999. www.ccs.neu.edu/research/demeter.

[22] T. Lindholm and F. Yellin. *The Java[tm] Virtual Machine Speci£cation*. Addison-Wesley, 1999.

[23] H. Masuhara, G. Kiczales, and C. Dutchyn. Compilation semantics of aspect-oriented programs. In *Foundations of Aspect-Oriented Languages workshop at AOSD*, 2002.

[24] S. McDirmid, M. Flatt, and W. C. Hsieh. Jiazzi: New age components for old-fashioned java. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* in *Special Issue of SIGPLAN Notices*, 2001.

[25] M. Mezini and K. Lieberherr. Adaptive plug-and-play components for evolutionary software development. Technical Report NU-CCS-98-3, Northeastern University, April 1998. To appear in OOPSLA '98.

[26] M. Mezini and K. Osterman. Integrating independent components with on-demand remodularization. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* in *Special Issue of SIGPLAN Notices*, 2002.

[27] D. Parnas. One the criteria to be used in decomposing systems into modules. In *Communications of the ACM*, volume 15, pages 1053–1058, 1972.

[28] R. Schantz, J. Loyall, M. Atighetchi, and P. Pal. Packaging quality of service control behaviors for reuse. In *IEEE International Symposium on Object-Oriented Real-time distributed Computing*, 2002.

[29] Y. Smaragdakis and D. Batory. Implementing layered designs with mixin-layers. In *European Conference on Object-*

*Oriented Programming*. Springer Verlag, 1998.

[30] K. Sullivan, L. Gu, and Y. Cai. Integration as a crosscutting concern for aspectj. In *Proceedings of the International Conference on Aspect-Oriend Software Development*, 2002.

[31] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *International Conference on Software Engineering*, pages 107–119, 1999.

[32] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. In *Foundations of Object-Oriented Languages*, 2002.

[33] P. Wegner. The object-oriented classi£cation paradigm. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 479–560. Cambridge, Massachusetts, MIT Press, 1988.