

# Modular Programming with Aspectual Collaborations

Research Abstract

Johan Ovlinger

johan@ccs.neu.edu

## 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 active intersection between 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 significantly speed up construction.

However, not all behaviors fit neatly into a module. Common examples are error handling and context-sensitive behavior (the exact behavior depends on where it was invoked from). In general, such behaviors rely on, and intercede in, events that occur in other parts of the program. Code in one module may affect an invariant maintained by another module. For example, we may wish to maintain a mirror of a disk directory in memory, without having to re-write either the *i/o* module or its client. The specification 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 sufficient to capture all the kinds of concerns the program-

mer 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* invoked whenever a file operation is executed from a certain context, we cannot use the same interface that is offered to clients that *explicitly* invoke file operations. 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 how to achieve the best of both worlds: modules and aspects, and gives one data point of how we needed to define 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 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 well with aspect oriented features.

## 2. ASPECTUAL COLLABORATIONS

Our solution is called *Aspectual Collaborations* (henceforth ACs), and captures the two different kinds of concerns alluded to in the introduction. We first describe concerns that interact explicitly (structural), followed by implicitly (behavioral).

**Structural Concerns.** The basic building block of our modules is a class-like construct we call a *participant*. The difference from classes is that participants explicitly declare modular properties such as which members are *expected* (will be provided later by some currently unspecified collaboration) or *exported* (visible to other collaborations).

We define an Aspectual Collaboration as a closed and named set of participants. The participants form a graph with *is-a* and *has-a* edges. An AC can either be declared (and compiled) directly, or composed from already compiled ACs (its *constituent* collaborations). Composition declares a participant graph for the result (typically, but not necessarily, isomorphic to a constituent), and point-wise maps participants from constituent ACs against participants in the result. Finally, the members (fields or methods) of the output participants are linked up: members can be exported (possibly under a different name), or provided to an expected member with the same signature (and on the same participant). When there remain no expected members, the

collaboration can be used like any Java package.

We refer to the details of how a constituent contributes to the resulting collaboration as an *attachment*. Reuse is achieved by attaching a collaboration in different ways. We can implement a type-safe List collaboration once, and instantiate it for varying types of items in our application. Each attachment will generate distinct types, so the List containing Integers is different from the List of Booleans.

At this point, we have sketched a fairly capable module system for an OO language, incorporating encapsulation, generic reuse, and external composition. However, we have not justified the “aspectual” part of the name.

**Behavioral Concerns.** The idea behind AOP is that we want to certain points in a *base* concern to transfer control to a *advising* concern. AOP’s key contribution is to specify the details of which points and what advice externally to the base. Thus, the base *implicitly* invokes advice. ACs argue that these details should also be external to the advising concern.

Previous AOP systems have maximized the expressiveness of the advising concern, allowing it basically free reign over the base program. This leads to a very flexible ability to advise concerns, but at a high cost: (a) Program readability suffers, as we now have to comprehend all advising concerns to determine whether a point in the program is advised. (b) Comprehending the program becomes harder, as the lack of encapsulation means that we have to understand it as a monolithic whole, rather than as individual modules and their composition. (c) It becomes very difficult to separately compile the base or the advice, as both rely intimately on the other.

It has been unclear how to moderate these drawbacks without giving up almost all the benefits of AOP. AC’s contribution is demonstrating that modules interact very well with AOP, ameliorating all the drawbacks above, with only small modifications to module and aspect languages.

Thus, we augment our collaborations to additionally capture the needs of behavioral concerns. We add the ability to export three new entities into the interface of ACs: (a) Behavioral join points, which expose a method’s executions to external advice without also allowing external code to invoke the method. Join points are typed, indicating what advice can be attached and any additional information (such as arguments, or the kinds of exceptions that can be intercepted) that can be used by the advice. (b) Sets of tuples of join points, allowing us to advise multiple join points as a group (such as matched setter-getter method pairs) and to control how state is shared within and across such tuples. All tuples in a set have the same (known) type, so it is attractive to visualize it as a *Join Point Table (JPT)*, where each tuple is a row. (c) Advice (*aspectual methods*) that can be attached to join points. When an advised method is executed, the execution is reified as an object and passed to the aspectual method, which has the controls if, how, and when the advised method’s execution continues.

Because we layer these behavioral features on the foundation of a strong module system, we immediately realize several benefits. We are able to reason modularly about our aspects: only join points that have been exported can be advised, and by tracing the composition tree, we quickly see which advice have been applied to a join point. We can use the composition and encapsulation features to quickly build larger aspectual behaviors from smaller ones. Our as-

pectual behaviors are reusable, as they export advice rather than talk directly about the base join points. By varying the attachment, different join points of the base can be advised *without recompiling the advice*. Lastly, there exists a type-safe API for aspectual methods, allowing them to control execution of base methods of any signature, while optionally trading off genericity for greater knowledge and influence of the details of the intercepted execution.

### 3. OPEN ISSUES AND RELATED WORK

We have sketched the main features of Aspectual Collaborations, illustrating that modules and aspects are not only not contradictory, but actually synergistic.

However, several issues remain to be tackled. (a) The module language described for structural concerns is strikingly similar to that of Jiazzi [?]. To minimize our programming burden, and to highlight the exact nature of our contribution, we are currently investigating whether Jiazzi can be used as the back-end of our implementation rather than working with byte-code directly. (b) We would like to prove that the module language, in light of attachments of advice to JPTs, is type safe. Performing a soundness proof over Java is intractable, but similar proof over a smaller language, such as ClassicJava or FeatherweightJava will show that the concepts are sound. We can then argue that our full implementation is at least possibly safe. (c) ACs remain untested on large programs, so it is unclear to what extent additional features are needed. This will be rectified by the reimplementing around Jiazzi, which will be self-hosting, using our current prototype as the bootstrap. We foresee perhaps wanting to filter JPTs on some suitably abstract property, and parameterizing composition over ACs, which would require the introduction of signatures to capture AC interfaces.

**Previous Work** This work is related to [?], which describes a dynamic collaboration language. However, the dynamic approach has a high price: questions concerning type safety, implementability, semantics of advising multiple join points, and whether the components offer complete encapsulation, are left unanswered. Our work shares common roots with their language, but by choosing a more static approach can precisely answer those questions. Additionally, aspect reuse is discussed in [?], focusing on the two prominent AOP projects: AspectJ [?] and Hyper/J [?].