# Aspectual Collaborations:
# Combining Modules and Aspects*

KARL LIEBERHERR, DAVID H. LORENZ AND JOHAN OVLINGER

*College of Computer and Information Science, Northeastern University, 360 Huntington Avenue, Boston, MA 02115, USA*
*Email: {lieber,lorenz,johan}@ccs.neu.edu*

**Complex behavior often resists clean modularization in object-oriented languages. Aspect-oriented programming tackles this problem by providing flexible module boundaries that can span and partition classes. However, in order to achieve this flexibility, valuable modularity mechanisms, such as encapsulation and external composition, are lost. The ability to separately compile or reason about a modular unit is also compromised. We propose that this tradeoff is not necessary and that by expressing aspects as part of our modules, we can restore the lost modular properties while maintaining aspectual features. As a concrete demonstration, we present the main features of Aspectual Collaborations and show how these interact to combine modularity with aspectual behavior. The expressiveness of Aspectual Collaborations, AspectJ and Hyper/J are compared using a challenge problem, allowing us to estimate the effectiveness of the approach.**

## 1. INTRODUCTION

A fundamental goal of software engineering is to enable the construction of large and complex software in a timely fashion. Several approaches to this goal are under investigation by researchers. Examples are programming methodologies, modeling tools and advanced development tools. This paper investigates a fourth approach: novel programming constructs. Specifically, we investigate the intersection of *modular programming* (MP) [1] and *aspect-oriented programming* (AOP) [2]. We show, in the context of an illustrative example, how aspectual mechanisms work in AspectJ [3] and Hyper/J [4] and we show how combining modules with aspects provides better aspect-oriented modularity. We offer a concrete language design, *Aspectual Collaboration* (henceforth AC), both to demonstrate the feasibility and to investigate the details of such a combination.

### 1.1. Modular programming

MP allows large projects to be constructed in a timely fashion by providing constructs that split the program into smaller units: modules. These modules are assembled to create the final deliverable.

Modules promote safety at the expense of *ad-hoc* flexibility. The power of modularity springs from three fundamental abilities.

***Decomposability.*** A key benefit of modularizing program development is to ease the burden of understanding the program by allowing pieces to be understood in isolation.

This property additionally isolates development teams from each other, allowing development to proceed in parallel without the danger of teams interfering with each other.

***Composability.*** While small applications can be written monolithically, larger applications need to be assembled from individual modules in order to be manageable. It is natural to want to apply modular reasoning recursively to larger modules: constructing them not monolithically, but from smaller constituent modules. This allows the programmer to balance flexibility, which is maximal in a flat architecture, against comprehensibility, which is easiest in a deeply nested hierarchical architecture.

***Adaptability.*** Developing and understanding a module costs considerable resources, so it is important to be able to amortize that expenditure over several uses of the module. A module system that allows modules to be reused in a wide variety of situations—including when names and assumptions between importee and importer may not match precisely—is important to realize the benefits of module systems.

### 1.2. Aspect-oriented programming

AOP improves the comprehension and maintainability of complex programs by localizing behaviors that would otherwise be scattered and tangled. These behaviors are referred to as *concerns* (as per Parnas [1]). Crosscutting at the concern level leads to scattering and tangling at the implementation level. Kiczales [2] defined an *aspect* as 'a well modularized implementation of a crosscutting concern'.

Aspects promote *ad-hoc* flexibility at the expense of safety. The power of aspect-oriented software development

lies in managing crosscutting concerns by controlling scattering and tangling.

***Crosscutting.*** Crosscutting concerns are issues that the programmer cares about, but cannot localize to one instance of a programming construct. For example, a synchronization concern is not crosscutting if it merely talks about one section of the program text, while a more advanced synchronization concern that specifies that some set of methods can simultaneously execute in a critical section as long as some condition holds is crosscutting. Such a concern will be crosscutting as the *ad-hoc* implementation of the policy will likely be spread out over each of the affected methods.

***Scattering.*** Scattering is the condition where a concern is implemented in several non-contiguous places in the program. Aspects control scattering by specifying places in the program's execution (*joinpoints*) where certain code (*advice*) is to be executed.

***Tangling.*** Tangling, the dual of scattering, occurs when several concerns overlap at a region in the program text. This hampers maintainability, as the programmer must mentally categorize statements in the program text by which concern they belong to. Aspects reduce tangling by localizing concerns.

### 1.3. Combine modules and aspects

MP and AOP are different approaches to a common goal of constructing large, complex software quickly. Unfortunately, neither is solely sufficient.

On the one hand, module systems alone fall prey to (significant) scattering and tangling. To be effectively distributed pre-compiled software requires foresight of how a module will be used and extended. For example, in a graphical user interface (GUI) application, one may need to provide enough hooks for the subject-observer design pattern [5] to allow all interesting events to be captured. Thus, to provide the necessary support for managing crosscutting, one would need the prescient ability to predict all the future uses of a pre-compiled module.

On the other hand, AOP alone, in its current form, lacks lingual support for reuse and composition. Teams can no longer efficiently work in parallel relying only on common module interfaces.[1] However, the biggest lost benefit is the ability to understand programs in a modular manner. If advice can be added to a joinpoint by any aspect, we must discover and understand *all* aspects globally in order to comprehend a method's local behavior. We thus need omniscient reasoning to understand the program's local behavior.

The obvious remedy is to combine the two. However, the complex nature of the interactions between concerns makes it seemingly cumbersome to apply language constructs to control how aspects interact. Notably, protection and the mechanisms to capture scattering appear to be in

direct contradiction: the former places strong emphasis on differentiating the interior of a module from the exterior, while the latter wants to allow external effects to the internals of a concern.

We show that this conclusion is a fallacy and that the abilities of modules and aspects are synergistic. We argue that to combine aspects and modules is the right thing to do and that such a combination can be both simple and powerful. A system combining the two will provide a significant step towards reaching our goal of being able to conveniently write and reason about large and complex programs.

We illustrate our argument by presenting a concrete system, called AC, combining modular and aspect-oriented features. We evaluate our design choices by comparing its expressiveness on a challenge problem to that of two existing systems, AspectJ and Hyper/J, on the same problem.

### 1.4. Outline

The paper is split into three major parts. The first part identifies the need to combine MP and AOP, deriving in Section 2 a list of the desired properties. To evaluate these claims, a challenge problem is presented in Section 3 and two solutions are investigated. By comparing the solutions against our identified properties, the first part of the paper concludes that neither solution is able to compensate for the features it lacks but are found in the other.

The second part of the paper presents a concrete system combining all the properties identified in the first part. Section 4 presents ACs, as motivated by those properties. Section 5 evaluates the expressiveness of ACs using the challenge problem, presents implementation sketches to illustrate the conceptual model of the implementation and suggests future research directions.

In the third part of the paper, Section 6 compares ACs against related work and Section 7 concludes.

### 2. DESIRED PROPERTIES

This section identifies the subset of properties provided by modules and aspects that are pertinent to our discussion.

### 2.1. Properties of modules

The capabilities of module systems stem from several intertwined principles.

***Encapsulation.*** Decoupling an interface from its implementation allows the implementation of a module to vary without affecting its clients, as long as the interface remains the same. We say that the module *encapsulates* its contents behind an interface. Encapsulation enables modules to be implemented and managed separately and in parallel. The requirements for encapsulation often allow a module to be compiled separately from the rest of the application, allowing pre-compiled modules to be distributed as commercial off the shelf (COTS) components. While separate compilation is attractive, it is not as important as the ability for the programmer to analyze and comprehend a module separately from its uses. The ability to control

---

[1]Teams can work independently, one team per use case. However, interactions would require some form of aspectual interfaces.

the visibility of its contents is the key to supporting local analysis of the module. The problem in such analysis is that it is seldom apparent how a module's contents are referenced (and possibly modified) by external modules. By controlling what is and is not exported, the programmer can control and identify exactly which parts of the module are understood completely (as they are not exported) and which are understood partially (as they are exported and may thus be referenced by unknown external modules).

***Hierarchical composition.*** Hierarchical composition of modules allows larger modules to be constructed from smaller modules, instead of being written from scratch, with absolutely no functional distinction. This allows modularity to be used to comprehend, as well as to construct, an application. It is easier to understand a large *composite* module consisting of smaller, assembled modules, than a monolithic *atomic* module of the same functionality, as the former can be understood by first understanding the constituent parts and then how they fit together, while the latter needs to be understood in its entirety to be understood at all. Composition is also important for local knowledge: a composite module can be traced in a top-down manner to 'zoom in' on local behaviors: composition describes how that piece interacts with the rest of the module, while the atomic module offers no such guides. If local contents are not exported from a composite module, encapsulation statically assures that they cannot be referenced or modified by code outside the composite module. The tradeoff is that the atomic module will be able to express complex behaviors that cannot be expressed conveniently in a composite module.

***External assembly.*** The external assembly of modules allows modules to be written not only independently of each other, but also *ignorantly* of each other. Because they are assembled externally, a module does not need to have explicit references to other modules. Rather, a module is quantified over its imports and constraints, allowing it to be used in any environment capable of fulfilling these requirements. Thus, the external assembly of modules removes hardwired assumptions about how a module will be used, opening it up to reuse in contexts other than those originally envisioned. This allows programmer effort to be saved, but perhaps more importantly, allows comprehension of the module's behavior to be reused as well. To promote flexibility, the assembly can often perform limited adaptation of modules, for example, to reconcile differences in names and types.

***Non-type-invasive extension.*** In order to achieve type safe reuse, multiple instantiations of a module must be kept separate, as they may have been assembled with incompatible imports. A guiding principle in this regard is *invasiveness*, defined by Ernst [6] as whether '[a module's influence] can be detected by inspecting the target module'. A weaker form of this property is whether it is possible to confuse one instantiation of a module with another (if instantiations cannot be detected, they cannot be confused). The danger of confusing modules can be illustrated in the context of role (or class) based modules.

It is tempting to instantiate a module containing a role model by declaring classes in the importing module as subtypes of the interfaces declared by the roles. This is invasive if it is possible to test dynamically whether an object of one of the implementing classes is a subtype of a role type. If the object is a subtype of a role type, an upcast to that type will make it possible to confuse it with other classes implementing that role type. Such upcasts are likely, as inheritance of methods from roles will likely expose role types in method signatures.

## 2.2. Properties for aspects

Aspect-oriented languages typically support two kinds of mechanism to incrementally add a concern to an application: enhancements to static program text and enhancements to dynamic call graphs. As was the case for module systems, the powers of aspects arise from a number of fundamental properties.

***Non-behavior-invasive extension.*** Concern composition will typically require that the extending concern's execution be interleaved with or controlled by the execution of the base concern. The base behavior can be implemented with such an extension in mind, by using the observer pattern or variations thereof. However, in general, such foresight cannot be assumed: we want the base to remain *oblivious* to the invariants imposed on it. This promotes both clarity, by keeping the concerns separate and flexibility, by allowing behaviors to be extended in ways not foreseen at design time.

The ability to enhance the class graph with additional behavior or relationships allows the base to remain oblivious to the extension being added.

***Method interception.*** To be able to express complex interactions between concerns, one concern may not need only to be informed about points in the execution of the other concern, but also to modify these executions. Such power could allow the programmer to gracefully recover from errors by retrying method calls with different arguments, to implement security features by only allowing method executions to proceed under some circumstances and to optimize expensive computations by diverting calls to special case methods depending on the details of the call.

This ability is called *method interception*, as once intercepted, the method is under outside control. Such interception can be seen as enhancement of an oblivious dynamic call graph.

***Generic advice.*** Exactly what enhancement happens at points in the dynamic call graph is called *advice*. There is a tradeoff between expressiveness and reusability that needs to be weighed carefully. Simple advice is likely to be quite reusable: the recurring example is a logging concern, which is generally applicable to methods of any signature. More complicated behaviors, such as performance optimizations, are likely to be more tightly bound to the details of the method. Ideally the programmer should be able to choose how to balance reusability against expressive power. This choice should not come at the cost of giving up runtime type safety.

**Listing 1.** Base program in Java.

```
 1  package base;
 2  import java.util.*;
 3  abstract class Item {
 4    String name;
 5    public abstract int weight();
 6  }
 7  class Container extends Item {
 8    Vector elements;
 9    int capacity;
10    public Container(String name, int cap) {
11      this.name = name;
12      this.capacity = cap;
13      this.elements = new Vector();
14    }
15
16    public String get_name() { return name; }
17    public void addItem(Item i) { elements.add(i); }
18    public int weight() {
19      Iterator it = elements.iterator();
20      int total = 0;
21      while (it.hasNext()) {
22        Item child = (Item)it.next();
23        total += child.weight();
24      }
25      System.out.println(
26        "Container " + name + " weighs " + total);
27      return total;
28    }
29    public void check() {
30      int total = weight();
31      if (total>capacity) {
32        System.out.println(
33          "Container " + name + " overloaded");
34      }
35    }
36  }
37  class Simple extends Item {
38    public int weight() {
39      return weight;
40    }
41    public Simple(String name,int weight) {
42      this.name = name;
43      this.weight = weight;
44    }
45    private int weight;
46  }
47  public class Main {
48    static public void main(String[] argv) {
49      Container c1 = new Container("Container 1",4);
50      Container c2 = new Container("Container 2",2);
51      Container c3 = new Container("Container 3",1);
52      Simple apple = new Simple("apple",1);
53      Simple pencil = new Simple("pencil",1);
54      Simple orange = new Simple("orange",1);
55      Simple kiwi = new Simple("kiwi",1);
56      Simple banana = new Simple("banana",1);
57
58      c3.addItem(kiwi); // Container 3 weighs 1
59      c2.addItem(c3);
60      c2.addItem(apple); // Container 2 weighs 2
61      c1.addItem(orange);
62      c1.addItem(pencil);
63      c1.addItem(c2); // Container 1 weighs 4
64      c1.check(); // cached c1:4, c2:2, c3:1
65      c1.addItem(banana); // cache cleared for c1, weighs 5
66      c1.check(); // cache hit for c2. c1 overload
67    }
68  }
```

## 3. CHALLENGE PROBLEM

In order to evaluate our design criteria, we present a challenge problem. We illustrate how the criteria are necessary properties for an elegant solution. On a formal level, we are evaluating *our solutions* in the pertinent languages, rather than the languages themselves, but we have attempted to solve the problem as idiomatically as possible and maintain that the solutions are indicative of the languages they are written in.

The problem put forth is to verify that no Container is over its capacity limit. A Container may contain a number of Items. An Item is either a Simple leaf or a Container. Each Simple item has a weight, while a Container has a capacity, which describes the maximum total weight it can contain in its nested subcontainers. The challenge is to introduce caching code so that we only have to perform minimal necessary recomputation to check the constraints.

We focus on three main concerns that seem both intrinsic to the problem, and worthy of reuse on their own. At first glance, it may seem that the differences in interfaces between the concern solutions—and thus the need to be careful when integrating them—are somewhat contrived.
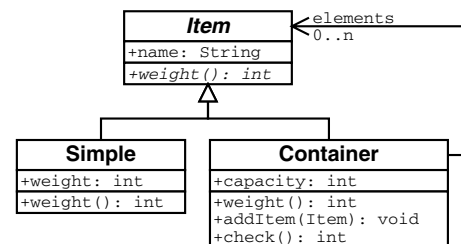


**FIGURE 1.** A UML representation of base.

The justification is the observation that we are also modeling the reuse of COTS software: in such scenarios, the interfaces are going to be what the designer of the software thought best.

***Capacity concern.*** The fundamental—or base—concern is checking whether any container's capacity has been exceeded. This concern can be easily implemented in plain Java using a recursive algorithm and the COMPOSITE design pattern [5]. Figure 1 illustrates the base collaboration's class graph as a UML diagram. The complete code is shown in Listing 1 and is common for all three solutions (AspectJ, Hyper/J and Aspectual Collaborations).
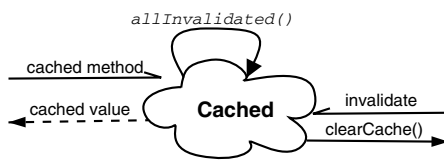
**FIGURE 2.** A schematic sketch of the Caching concern.



**FIGURE 3.** A schematic sketch of the Backlink concern.

The common superclass Item declares an abstract method weight which is implemented in the concrete subclasses Simple and Container to return the total weight of the container and its contained Items. The concrete subclasses differ in that Container has a (possibly empty) Vector of contained Items and a maximum capacity, while Simple has a weight but cannot contain any other Items. Container calculates its weight by recursing into each of its contained Items and summing their weights.

***Caching concern.*** The weight is hence computed from the subcontainers every time an item is added or deleted. An optimization that comes to mind is to cache the weight of each container. Whenever an item is added to a container, the weights of its sub- and sister containers remain valid, but the cached weight of itself and its supercontainers must be invalidated.

As a general solution, this concern for efficiency can be described in terms of the class Cached, referencing the method that should be cached, as sketched in Figure 2. The concern places a number of requirements on the context of its instantiation, which is illustrated at a high level in the figure.

An implementation of the concern needs to be provided with the ability to intercept calls to the method to be cached and to intercept calls to methods that invalidate the cached method's result. (Method interception is illustrated by the half-head arrows.) Also, it needs to call an external method, which returns all objects that have cached methods that depend on this object's cached return value (suggested by italics in the name *allInvalidated*). Given these imports, the concern provides the functionality of managing the method's cached result: intercepting the result on first invocation, intercepting subsequent calls to return the cached result (dotted arrow) instead of invoking the cached method and invalidating the cache when forced to do so by dependence on other cached values or direct invalidation—likewise, it will also inform dependent caches when it has become invalidated (normal name and arrow for clearCache).

***Backlink concern.*** In order to invalidate the proper containers, we need to identify all the supercontainers of an item, i.e. all the Container objects that the Item object is inside of. This would be possible if we tracked which Container an Item is added to.

Figure 3 illustrates the concern: given a Source object, containing a number of Target objects and a method to retrieve them (*getTargets*), the concern should augment the Target class with a Source field whose value can be retrieved by external behaviors (getSource). For any modification to the vector of Targets (associate half-head arrow), we need
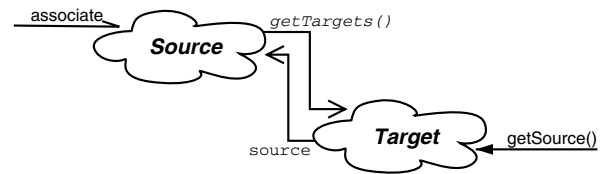
to determine whether the affected Target object was added to or removed from the vector and set the back field to the Source if added or null if removed.

Next, we illustrate the implementation of these concerns as aspects in AspectJ (Section 3.1) and as hypermodules in Hyper/J (Section 3.3), which are then evaluated (Section 3.5). The AC solution is deferred to Section 5.

### 3.1. AspectJ solution

The five aspects, three Java classes and three interfaces that make up the AspectJ solution are shown in Figure 4. The three classes are exactly the three original base classes of Figure 1, while the aspects are split into two abstract aspects aimed at reuse and three concrete aspects that attach and adapt the generic aspects to the details of the base. The abstract aspects and the base can each be compiled separately, while the concrete aspects cannot.[2]

However, separate compilation is merely a performance optimization. We are more interested in whether any aspects can be separately understood and whether they can be used in this solution without affecting the other parts of a larger system. The base (Item, Simple, Container) can certainly be understood in separation, if we compile without the concrete aspects.

#### 3.1.1. The Caching aspect

The Caching aspect in Figure 4 contains the inner interface Cached, which defines the role that the caching aspect is written over. (Readers who know AspectJ may benefit from Listing 2.) Classes playing the role Cached (by implementing the interface) can have one of their methods cached. The role requires a method *allInvalidated*, which should return all objects whose caches would be invalidated by a modification to the current object. However, unlike Java interfaces, AspectJ interfaces can also provide member implementations—we use this ability to provide a cachedValue field and a clearCache method. Note that at this stage we do not know which class(es) will implement the interface, but all implementing classes will be augmented with the provided behavior (and must in return provide an implementation of the required *allInvalidated* or be labeled **abstract**).

The Caching aspect declares some abstract pointcuts representing the (as yet unknown) locations in the program that represent the method to be cached (*cache*) and

---

[2]This situation is somewhat unclear. The three abstract aspects *can* be compiled separately, but also need to be recompiled along with the concrete aspects.
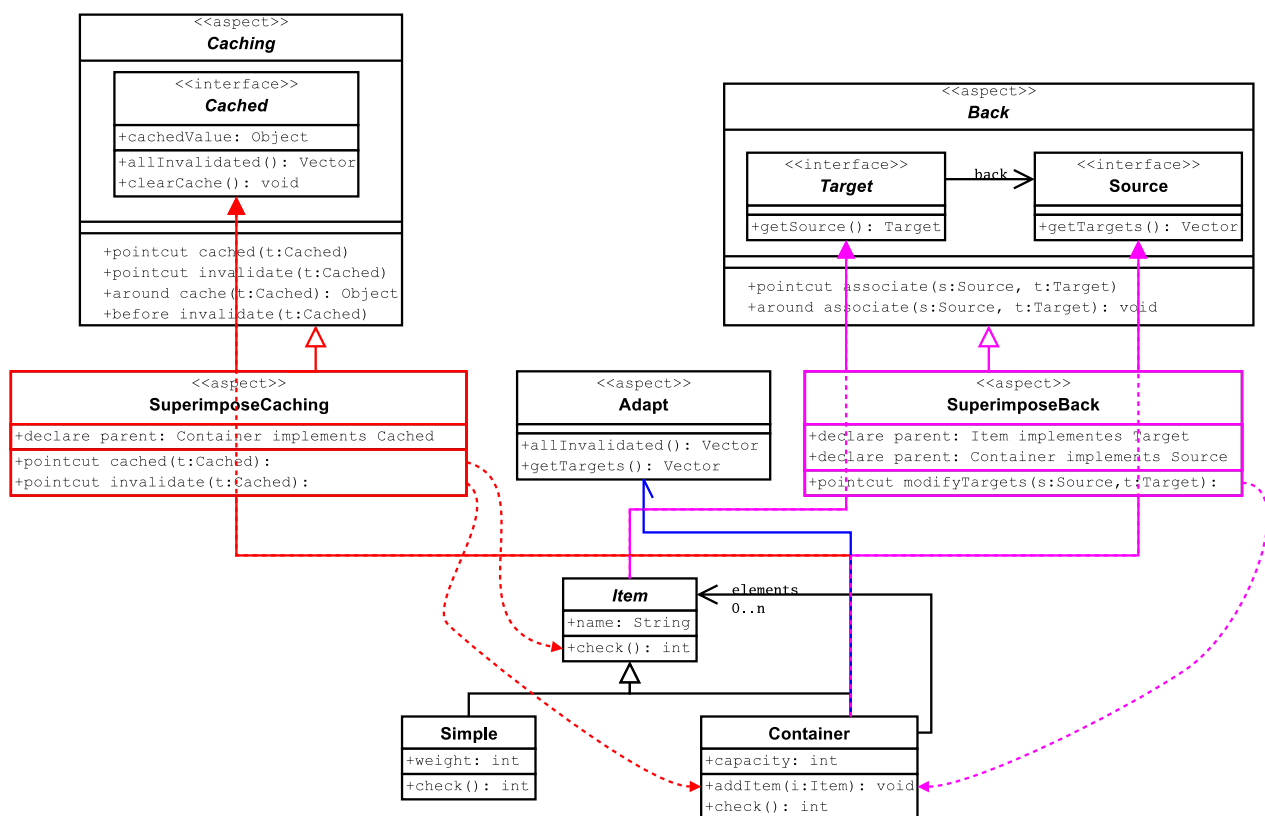
**FIGURE 4.** A graphical representation of the AspectJ solution.

the methods that invalidate such caches (*invalidate*). These points will be advised: the invalidation points will call clearCache, while the method to be cached will only be executed if the cache is clear, caching the returned value, otherwise it will just return the previously cached value.

### 3.1.2. The Back aspect

The Back aspect is conceptually similar to the caching aspect, in that we want to maintain an invariant and will do so by intercepting calls to methods that can affect the invariant. All objects playing the Source role have vectors of Target objects. The invariant we wish to maintain is that we want each such Target to have a back-link to the source in which it is contained.

We achieve this by augmenting (*introducing*, in AspectJ terminology) a backpointer to Source in all the classes implementing the Target interface. We also declare an abstract pointcut associate, which exposes the receiver and argument of each call to a method that modifies the target vector.

To maintain the invariant, we attach around advice to the abstract pointcut, so that all calls to such methods are intercepted and the argument to the method call (the Target object being added) has its back-pointer assigned the Source to which it is being added. We use 'around' advice rather than 'before' advice as we need to measure the length of the vector before and after the intercepted call: only if the length increases do we assume that the Target was added to the vector and that its back-pointer should be set.

Deducing when an item has been added is meant to exercise features promoting flexible reuse rather than to represent obvious backlink behavior.

### 3.1.3. Superimposing the aspects

Two concrete aspects connect the abstract aspects Caching and Back to the base classes. This is done by declaring that each concrete aspect inherits from an abstract aspect. The concrete aspect must provide a definition for abstract pointcuts in the parent aspect (indicated by dotted arrows in Figure 4). The concrete aspects additionally declare how the roles of the abstract aspects are played, by declaring which base classes implement the interfaces (indicated by full arrows that run underneath the declaring aspects).

The concrete aspect SuperimposeBack sets up the backlink relationship, declaring that Item play the role of Target and Container play the role of Source. It also specifies that addItem correspond to the associate pointcut.

The concrete aspect SuperimposeCaching declares that Container plays the role of Cached and that methods check and addItem should correspond to pointcuts cache and invalidate, respectively.

The concrete aspect Adapt introduces the necessary methods allInvalidated and getTargets so that the abstract aspects' requirements are met and no base classes have to be marked **abstract**. The half-head arrow indicates that Adapt provides behavior to Container without introducing any additional typing relations.

**Listing 2.** The Caching aspect in AspectJ.

```
1  package cache;
2  public abstract aspect Caching {
3    public interface Cached {
4      java.util.Vector allInvalidated();
5    }
6    Object Cached.cachedValue;
7    void Cached.clearCache() {
8      System.out.println("cache cleared");
9      cachedValue = null;
10   }
11   public abstract pointcut cache(Cached t);
12   public abstract pointcut invalidate(Cached t);
13   Object around(Cached t): cache(t) {
14     if(t.cachedValue==null) {
15       t.cachedValue = proceed(t);
16       System.out.println("value cached: " + t.cachedValue);
17     } else {
18       System.out.println("cache hit " + t.cachedValue);
19     }
20     return t.cachedValue;
21   }
22   before(Cached t):invalidate(t) {
23     java.util.Iterator it = t.allInvalidated().iterator();
24     while (it.hasNext())
25       ((Cached)it.next()).clearCache();
26   }
27 }
```

**Listing 3.** The Back aspect in AspectJ.

```
1  package back;
2  import java.util.*;
3  public abstract aspect Back {
4    public interface Source {
5      Vector getTargets();
6    }
7    public interface Target {}
8    public abstract pointcut associate(Source s, Target t);
9    void around(Source s, Target t): associate(s, t) {
10     Vector targets = s.getTargets();
11     int sizeBefore = targets.size();
12     proceed(s, t);
13     int sizeAfter = targets.size();
14     t.back = sizeBefore<sizeAfter? s: null;
15   }
16   public Source Target.getSource() {
17     return back;
18   }
19   private Source Target.back;
20 }
```

## 3.2. AspectJ code

Clarke and Walker's [7] translation of composition patterns to AspectJ is applicable to our challenge problem. We follow their transformation in spirit, if not to the letter, generating a surprisingly idiomatic AspectJ program. In our translation, we have attempted to produce reusable aspects, so that the AspectJ solution can be compared to the Hyper/J (Section 3.3) and AC (Section 5) solutions.

### 3.2.1. The Caching aspect in AspectJ
Listing 2 shows the AspectJ implementation of Caching. We delay the discussion of the use of interfaces to model participants until the Back aspect, as the discussion is more concrete in that context.

For simplicity, we assume that a cached method never returns null and use this as a sentinel for cache validity. An additional boolean variable, as in the Hyper/J example, could have been used to keep track of the validity of the cached value.

### 3.2.2. The Back aspect in AspectJ
Listing 3 shows the AspectJ implementation of Back. We write an abstract aspect which is made concrete in a subclass by providing the necessary application-specific information. AspectJ uses interfaces to declare types, which are then populated with behavior by introductions and **implemented** by the base program, while the two other approaches' concerns declare their behavior in classes which are externally composed with the base program.

The situation in AspectJ is analogous to that of Java: it is considered good programming style to program against an interface, but up to the programmer to decide when and where to do so. The AspectJ approach shares the problem with Java that one cannot instantiate an interface directly, but rather needs an abstract factory pattern. We discuss the typing issues of interfaces pertaining to attachment in Section 6.1.

Java interfaces describe the Back aspect's required interface. Interfaces can only contain methods, thus the aspect's required interface cannot contain fields. This restriction can be worked around by adding getters and setters to the interface instead and introducing methods to fulfill this interface. Thus, in AspectJ the situation with fields is even more restrictive than in Hyper/J, where we are forced to use comments to indicate a required field, but are able to compose them in the hypermodule.

Required methods are declared directly on their interfaces, while provided behavior is in the aspect body. Specifying an interface and its added behavior separately tangles the provided behavior for all the participants into one class body and separates provided from required behavior textually. The provided interface's members are added to host classes directly via AspectJ's introduction mechanism, which interacts gracefully with interfaces, adding the introductions to the implementing class rather than the interface (which of course could not be an interface were it to contain code).

Advice is declared against abstract pointcuts. AspectJ has a mature join point model and we are easily able to express our intended behavior. The format of the advice method differs from the AC and Hyper/J approach in that we explicitly pass in the receiver of associate, while this refers to the object representing the reified aspect. This allows advice to be quite general, wrapping methods of varying signatures in different classes.

### 3.2.3. Implementing concrete aspects
Listing 4 illustrates how the abstract aspects can be implemented for the current application through sub-classing.

**Listing 4.** Concrete instances of the aspects.

```
1  package connection;
2  import java.util.Vector;
3  import back.Back;
4  import back.Back.Source;
5  import back.Back.Target;
6  import cache.Caching;
7  import cache.Caching.Cached;
8  aspect SuperimposeBack extends Back {
9    declare parents: Item implements Target;
10   declare parents: Container implements Source;
11   public pointcut associate(Source s, Target t):
12     target(s)
13     && execution(void Container.addItem(Item))
14     && args(t) ;
15 }
16 aspect SuperimposeCaching extends Caching {
17   declare parents: Container implements Cached;
18   public pointcut cache(Cached target):
19     target(target) && call(int Item.weight(..));
20   public pointcut invalidate(Cached target):
21     target(target) && call(void Container.addItem(..));
22 }
23 aspect Adapt {
24   public Vector Container.getTargets() {
25     return elements;
26   }
27   public Vector Container.allInvalidated() {
28     Vector invalidated = new Vector();
29     Container s = this;
30     while (s != null) {
31       invalidated.add(s);
32       s = (Container)s.getSource();
33     }
34     return invalidated;
35   }
36 }
```

We attach the participant interfaces of the two abstract aspects to classes by having the classes implement the interface. We have moved the adapt concern into the connection package, as it is tightly bound to both of the concerns it adapts. AspectJ provides a very natural way to specify this sort of adaptation code.

The implicit effect of supplying a concrete pointcut to the abstract aspect is that its advice becomes attached to the application at the join points specified. As pointed out in Section 6.1, type-invasiveness has implications for the runtime type safety of a program.

It was also fortunate that back and cache did not have any name clashes. While AspectJ offers aspect-local methods for the concern's implementation behavior, the expected methods on the interfaces *must* be implemented by public methods with exactly that name.

*3.2.4. Summary*
The around cache advice illustrates the unconventional genericity mechanism of AspectJ's around advice [8]. Note that both the around cache caching advice and the cachedValue field are of the type Object. Around methods returning Object are able to advise methods of *any* signature—including void and int. We hope to be able to leverage subtype polymorphism to reuse this aspect for

caching methods of any signature. However, the method we wish to cache returns an int, which is not a subtype of Object, which would seem to make this a non-solution to our challenge problem. A feature of AspectJ resolves the quandary: if the return type is of primitive or void type, the cached method's result is transparently wrapped in a proxy object (in check's case, an Integer) before being passed to the around advice and likewise unwrapped before being returned to the caller of the [advised] cached method. Not all return values are thus mapped: a returned null is left unchanged. Unfortunately, this scheme is not statically type safe. AspectJ *will* catch many obvious type errors (such as so-called stupid casts [9]), but in some cases casting errors in unexpected places can result. For example, if the programmer were to assign a String to the cachedValue field, a dynamic casting error will occur when we attempt to return this as the cached value of check, but would be allowed by the AspectJ compiler, as String is a subtype of Object.

### 3.3. Hyper/J solution

Hyper/J is designed for capturing and manipulating slices of concerns. HyperSlices are extracted from compiled applications and composed into runnable applications. An interesting property of Hyper/J is that HyperSlices are in pure Java, with *all* concern-related information in the HyperModule file. This both helps and hampers readability. Each concern becomes very easy to understand, but we must look elsewhere to understand even the rudiments of how the concerns (can) fit together. Additionally, it means that concepts that should be separate become intertwined. An example is the use of **abstract** to denote both that the method should be overridden by subclasses and that the method is in the required interface of the HyperSlice.

Hyper/J labels a set of classes a HyperSlice and constructs new HyperSlices by composing several such slices together using a HyperModule specification. Deferred behavior (typically abstract methods) from one slice may be implemented in another: in general, within a composed class, a member can be replaced with another of the same signature. Some capabilities are provided to insert behavior before or after executions of a method, but to control the method's execution is quite complicated.

The Hyper/J solution is illustrated in Figure 5. Hyper/J's powerful composition mechanisms make it straightforward to create reusable slices. While different in implementation, the design is very close to the AspectJ solution. It consists of four constituent HyperSlices, Base, Back, Cache and Adapt, generating the composed HyperSlice CachedComputation. Each of the constituent slices is a separately compiled Java package. Italic names indicate that the member is deferred: in most cases, this means that we will be providing an implementation from another slice, but in the case of Item.weight (in Base or CachedComputation), this merely indicates a normal **abstract** method.

Hyper/J provides several mechanisms for creating deferred members, among which **abstract** is one. Another is to declare the method as throwing a Hyper/J-specific exception
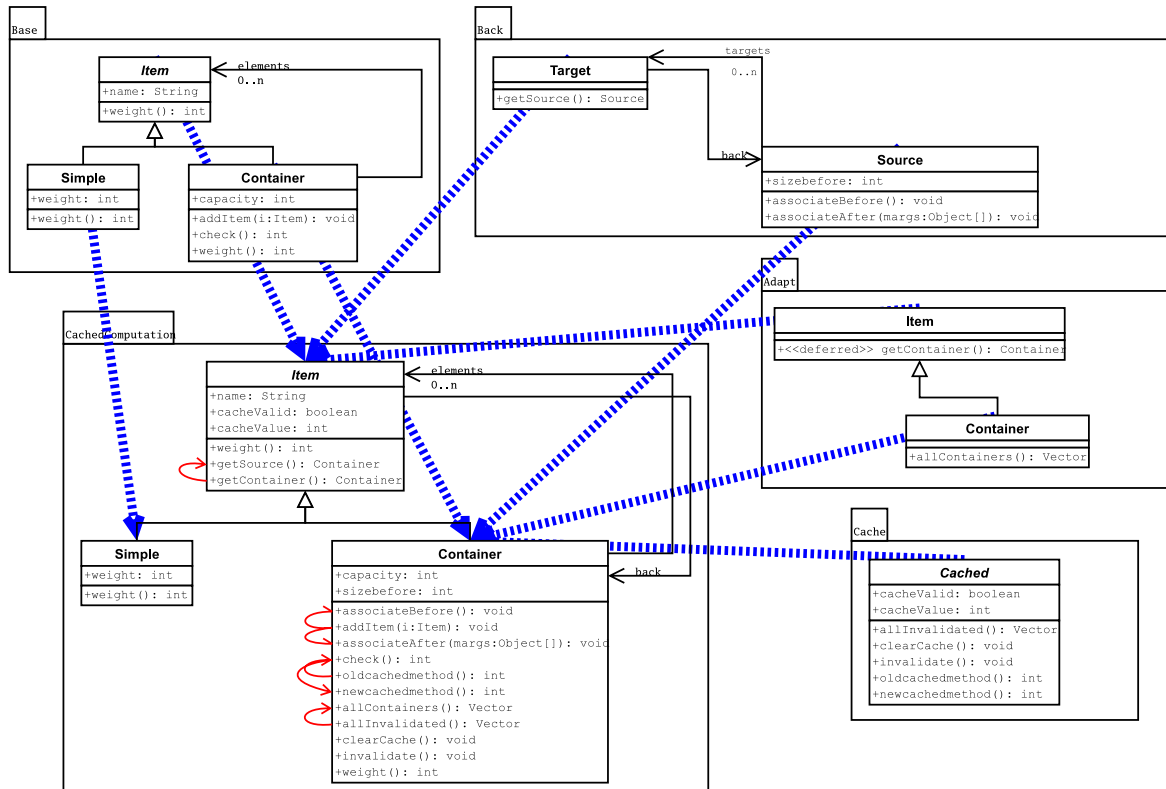
**FIGURE 5.** A graphical representation of the Hyper/J solution.

(these have been identified by the use of <<deferred>> annotations). The use of abstract makes it statically obvious when a method is deferred, but hinders the class from being instantiated. Use of the sentinel exception allows instantiation, but makes it harder to detect when a class is incomplete, with some deferred members not-yet implemented.

The thick dashed arrows indicate the composition of classes: as we can see, all the classes are composed into the same hierarchy as the base. We also note that each composed class contains the union of all the members from its constituent classes.

The rounded arrows on the left-hand sides of classes in the composed slice indicate how references have been redirected. For example, all references to *allInvalidated* are redirected to allContainers. The situations for addItem and weight are slightly more complicated. We set up addItem to bracket all invocations with associateBefore and associateAfter. The original body of weight is renamed oldcachedmethod, while the method newcachedmethod is renamed to weight. The result of this switch-around is that all method calls to (the former) weight will now go to newcachedmethod instead, which can invoke the original method via the name oldcachedmethod.

### 3.4. Hyper/J code

The Hyper/J code closely resembles the implementation sketches in Figures 2 and 3. Three so-called hypermodules are written, separately compiled, and finally composed with the base program.

#### 3.4.1. The Caching hypermodule in Hyper/J
Caching relies on 'around' advice, which Hyper/J does not have, so we must simulate this behavior.

There are two ways to achieve the required 'around' behavior in Hyper/J: splitting the 'around' into 'before' and 'after' advice, or manually mapping the advised method into the HyperSlice, so that it can be invoked explicitly.

Manual mapping gives the concern much more flexibility as to how to affect the wrapped method, but hampers reuse by requiring us to know the exact signature of the wrapped (*bracketed* in Hyper/J terminology) method. Caching (Listing 5) differs markedly from the AspectJ (Section 3.1) and AC (Section 5 versions), as it illustrates manually mapping the intercepted method call into the concern.

Hyper/J is able to provide access to the result of method invocation without needing to capture the bracketed method explicitly, but as we additionally need to be able to control *whether* the bracketed method is invoked at all, we must make it explicit in the concern. This, in turn, implies that we must hard-wire the exact signature of the method we are caching. In Section 3.4.4, we compose the hyperslices so that the original weight method on base.Container is hidden somehow and composed with oldcachedmeth (Listing 5, line 17) and newcachedmeth (Listing 5, line 18) becomes visible as weight. This allows the caching behavior to invoke the original behavior (Listing 5, line 20) when necessary. Setting this up is possible, but somewhat cumbersome and requires features that are not available in the current release

Listing 5. Cache method in Hyper/J.

```
1  package caching;
2  import java.util.*;
3  abstract class Cached {
4    boolean cacheValid = false;
5    int cachedValue;
6    abstract Vector allInvalidated();
7    void clearCache() {
8      System.out.println("clear cache");
9      cacheValid = false;
10   }
11   void invalidate() {
12     Iterator inv =allInvalidated().iterator();
13     while (inv.hasNext()) {
14       ((Cached)inv.next()).clearCache();
15     }
16   }
17   abstract int oldcachedmeth();
18   int newcachedmeth() {
19     if (!cacheValid) {
20       cachedValue = oldcachedmeth();
21       cacheValid = true;
22     }
23     else
24       System.out.println("using cached value");
25     return cachedValue;
26   }
27 }
```

Listing 6. Add and maintain backlinks in Hyper/J.

```
1  package backlink;
2  import java.util.*;
3  class Source {
4    Vector targets; // deferred
5    int sizebefore;
6    void associateBefore() {
7      sizebefore = targets.size();
8    }
9    void associateAfter(Object[] margs) {
10     Target targ = (Target) margs[0];
11     int sizeafter = targets.size();
12     targ.back = (sizebefore<sizeafter)? this: null;
13   }
14 }
15 class Target {
16   Source back;
17   Source getSource() {
18     return back;
19   }
20 }
```

Listing 7. Adapting the concerns in Hyper/J.

```
1  package adapt;
2  import com.ibm.hyperj.UnimplementedError;
3  import java.util.Vector;
4  class Item {
5    Container getContainer() {
6      throw new UnimplementedError();
7    }
8  }
9  class Container extends Item {
10   Vector allContainers() {
11     Vector v = new Vector();
12     Container c = this;
13     while (c != null) {
14       v.add(c);
15       c = c.getContainer();
16     }
17     return v;
18   }
19 }
```

of Hyper/J. However, the developers have kindly furnished us with a pre-release version that is able to perform this mapping.

Also notable is that we have chosen to use the **abstract** to indicate that the allInvalidated method (Listing 5, line 6) is required. Unlike fields, where comments are the only option, methods can be annotated as required either by declaring them abstract or implemented to throw the Hyper/J-specific UnimplementedError exception. This sentinel exception is recognized by the composition system to indicate a deferred method and thus the stub method body is omitted when two methods are composed. The benefit of using the sentinel approach is that the class remains instantiatable, while the abstract method approach has the benefit that it is statically obvious that some method in the class is deferred. The latter approach suffers from the additional drawback of forcing each subclass of the abstract class to be abstract as well, even if they have no abstract methods. This is because they inherit the deferred method, which they cannot override, as that would hide the behavior the deferred member is supposed to receive through composition.

### 3.4.2. The Backlink hypermodule in Hyper/J

Listing 6 shows the Hyper/J implementation of the backlink concern. The most striking difference from the AspectJ and AC versions here is that associate is split into two methods. We simulate 'around' advice by splitting it into 'before' and 'after' advice to a method. Splitting allows the concern to remain oblivious to the signature of the wrapped method, but is only applicable when the inner method is to be called exactly once, the result is not handled and we are certain that the splitting is thread-safe.

The before method associateBefore just stores (Listing 6, line 7) the size of the vector in an instance variable, so that it is available in the associateAfter method. This adds a small but non-zero risk of race-conditions in multi-threaded code, as a second thread could overwrite this variable before associateAfter has read it (on Listing 6, line 12). Similarly, recursive invocations need special care not to overwrite the enclosing invocation's scope.

The signature of the method advised by this simulated 'around' is constrained in the solution as well, but in this case it is constrained by a cast in the after method body (Listing 6, line 10), trading off static safety against some runtime flexibility.

Unfortunately, we are reduced to comments to indicate that targets is not implemented in this concern, as the techniques Hyper/J uses to identify deferred members only work for methods.

In Section 3.4.4, we see that the intended use of associateBefore and associateAfter will be to compose them as

**Listing 8.** Attach the slices in Hyper/J.

```
 1  −hyperspace
 2   hyperspace H
 3   composable class backlink.Source;
 4   composable class backlink.Target;
 5   composable class caching.Cached;
 6   composable class adapt.Item;
 7   composable class adapt.Container;
 8   composable class base.Item;
 9   composable class base.Container;
10   composable class base.Simple;
11  −concerns
12   class backlink.Source: Feature.Back
13   class backlink.Target: Feature.Back
14   class caching.Cached: Feature.Cache
15   class adapt.Item: Feature.Adapt
16   class adapt.Container: Feature.Adapt
17   class base.Item: Feature.Base
18   class base.Container: Feature.Base

20  −hypermodules
21   hypermodule CachedComputation
22   hyperslices: Feature.Base, Feature.Cache, Feature.Adapt, Feature.Back;
23   relationships:
24    mergeByName;
25    compose class Feature.Back.Target
26      with additionally class CachedComputation.Item;
27    equate operation Feature.Adapt.getContainer, Feature.Back.getSource;
28    compose class Feature.Back.Source, Feature.Cache.Cached
29      with additionally class CachedComputation.Container;
30    equate operation Feature.Cache.allInvalidated, Feature.Adapt.allContainers;
31    equate variable Feature.Base.Container.elements, Feature.Back.Source.targets;
32    forward operation CachedComputation.Container.weight
33      to operation Feature.Cache.newcachedmeth;
34    equate operation Feature.Cache.oldcachedmeth to operation Feature.Base.weight;
35    bracket "Feature.Base.Container"."addItem"
36      before Feature.Back.Source.associateBefore()
37      after Feature.Back.Source.associateAfter($ArgumentArray)
38    bracket "Feature.Base.Container"."addItem"
39      after Feature.Cache.Cached.invalidate()
40   end hypermodule;
```

**before** and **after** brackets to addItem. The margs argument (Listing 6, line 9) will be constructed by the HyperModule to pass in the arguments from the bracketed method.

### 3.4.3. The Adapter hypermodule in Hyper/J

As in the AspectJ solution, there is a mismatch between the interfaces of the backlink and caching packages. Listing 7 illustrates how these can be resolved. Note, however, that the deferred method getContainer is implemented to throw UnimplementedError (Listing 7, line 6) rather than being abstract. Had it been abstract, then class Item would also have been abstract (Listing 7, line 4), which in turn would have forced Container (Listing 7, line 9) to be abstract as well.

### 3.4.4. HyperSlice composition

Listing 8 shows the Hyper/J specification for identifying and composing the HyperSlices presented above. The specification consists of three parts that optionally can go into separate files. The **hyperspace** specification (Listing 8, lines 1–10) identifies which classes are participating in the composition. These classes are partitioned into hyperslices by the **concerns** specification (Listing 8, lines 11–18). Finally, the **hypermodule** (Listing 8, lines 20–40) chooses which of these hyperslices to compose and how their contents relate.

The **relationships** clause is a sequence of flat declarations. The **mergeByName** declaration is used mainly at the class level in this example, automatically composing similarly named classes (for example, adapt.Container and base.Container). The result of the composition of hyperslices is the union of all their classes, minus the classes that have been composed, either through explicit annotations or implicit by-name merging. The situation at the level of class members is analogous.

### 3.4.5. Summary

Note that we had to foresee the exact signature of weight, as oldcachedmethod and newcachedmethod are required to have the same signature in order to be able to replace the original method, which impacts how often we are able to reuse the caching behavior.

Method bracketing—Hyper/J's approach to behavioral extension—has a less severe version of the same problem. We need access to the receiver of the bracketed method in associateAfter, which is passed in via an Object array, also containing any arguments to the call. Elements from this array are downcast to their concrete types before use. Unfortunately, as what these types are assumed to be is not apparent from the signature of the bracketing method, Hyper/J has no way of statically determining whether a bracketing method is type safe with respect to the bracketed method.

Finally, we note that Hyper/J offers no support for encapsulation, or to assert that a composed slice conforms to some interface: the set of visible classes and members in a composed slice is exactly the union of the constituent slices, modulo merging and explicit renaming. It is thus complicated to update a slice's implementation without affecting its interface, which makes it impossible to perform local maintenance without possibly affecting the composed application and difficult to predict the interface of a hypermodule without running the Hyper/J tool.

A HyperSlice can be defined as a slice of a composed hypermodule, which in turn could be composed from slices, creating an import chain of arbitrary length. The lack of encapsulation means that we must start at the sources and mentally propagate through the whole chain to build up the final interface, rather than having it declared as part of the hypermodule. The two effects also interact, in that adding a method to a module early in the chain can lead to a spurious match later on, with unintended effects. The situation is

similar to that of accidental inheritance [10] or accidental method capture [11].

It *is* possible to rename members and classes, thereby implementing a naming convention to indicate which module contents should not be externally visible and minimizing unintended name clashes.

### 3.5.  Evaluation

The AspectJ solution has many subtle points, but we want to make a number of observations.

(a) The command line used for AspectJ also determines which `.java` files we must comprehend in order to understand local behavior. Since a method in a class can be advised by any aspect in the system, we must determine the totality of aspects that have been compiled and determine which methods they affect before we are able to understand the effect of calling a method.

(b) AspectJ has been designed for convenience and power. Pointcuts are integrated into Java in a very intuitive fashion, interacting with late binding to allow the reuse of aspects in several situations. Unfortunately, as a modularity construct, this is unsatisfactory, as it becomes impossible to keep the various uses of the aspect separate from each other.

(c) The genericity mechanism for advice has similar design choices, promoting convenience for programming in the small over predictable behavior in edge cases.

(d) There are no encapsulation boundaries for advice, allowing the programmer to decide how generic or how convenient they write aspects. Of course the Java concepts of private and public can be used for advice, but there are no provisions for controlling the visibility of the methods introduced by Adapt so that they are visible to the abstract aspects but not other classes in the system.

(e) Care must be taken so that multiple uses of an aspect do not become intermixed. It seems likely that an aspect's deferred behavior (expressed as methods on the aspect's interfaces) will be implemented by the concrete aspect by downcasting the interface types to the known implementing classes. This will be necessary if we wish to gain access to behavior not exposed by the aspect's interface. It is probable that such access will be needed in order to implement the methods required by the interface.

These points may seem overly negative, but should rather be read as confirmation that AspectJ is clearly (and convincingly) designed to be a convenient tool for programming in the small rather than as an advanced module system.

Hyper/J, on the other hand, promotes modular reasoning and control over direct convenience. While highly reusable, the solution for Hyper/J is not intuitive: for example, it requires the splitting of the advice to associate into before and after advice and requires significant manual relabeling

of methods in order to simulate around advice. For all this, it still faces possible casting errors when providing advice access to a method call's arguments.

Like AspectJ these points are not criticisms of Hyper/J's design, but rather evidence that modularity alone is not sufficient to elegantly provide reusable solutions to complex programming problems.

This concludes the first part of the paper. In the next section, we describe the design principles behind ACs and in Section 5 bring the AC solution to the challenge problem.

### 4.  COMBINING MODULES AND ASPECTS

An AC is a module that captures both object-oriented and aspect-oriented features. Each AC defines a formal class graph (a collaboration), which can be superimposed [12] on another class graph. The class graph consists of Java class-like **participants**, with method and field (members, collectively) declarations. Additionally, a participant can have 'holes' in its declarations: **expected** members whose signatures are known, but whose implementation is deferred. Programmers familiar with Java will see the resemblance to **abstract** members. The main difference at this stage is that **expected** members do not statically inhibit instantiation and are not provided via inheritance and overriding.

Finally, participants can contain a special sort of method: **aspectual** methods, which are able to intercept method executions while allowing those intercepted to remain oblivious to the interaction. Generally, an aspectual method is completely oblivious to the signature of methods it intercepts (thus remaining generically reusable), but can alternatively choose to (partially) constrain the signature of methods it can work with, buying expressiveness at the price of genericity. Expected and aspectual members offer two alternative ways—explicit and implicit, respectively—to transfer control and information from one collaboration to another.

An AC is either declared atomically or composed from superimposed constituent Aspectual Collaborations. We refer to the act of superimposing as *attaching* a collaboration to a host collaboration; the host collaboration is said to have been *decorated*. Composite collaborations declare an output participant graph and map participants from constituent collaborations against output participants. This mapping translates types from the constituent kind to the types mentioned in the output graph.[3] A composite collaboration is *indistinguishable* from an atomic collaboration programmed to implement the same interface.

The interface of the composite collaboration is declared by selectively exporting members from the constituents. Expected members are provided with implementations by mapping them to members on the same output participant, or exporting them to be provided later. Likewise, aspectual methods can intercept executions of methods on the same output participant.

---

[3]References to external types (such as the Java class library or third party libraries) do not need be translated.

## 4.1. Design principles

ACs have a very simple design, based on the principles of encapsulation and composition, but are still able to capture the properties we identified in Section 2.

1. **Encapsulation.** A collaboration's interface controls both how external collaborations can access its members and what requirements must be fulfilled in order for the collaboration to function. These requirements can be imports that need to be provided, or assumed structural relationships between participants.

2. **Hierarchical composition.** Composite collaborations are indistinguishable from atomic collaborations. Thus, the implementor can decide at each stage whether required functionality is best implemented by an atomic collaboration or by composing several existing ones. The composition of a constituent collaboration into a resulting collaboration is called an *attachment*.

3. **External assembly.** Composition is performed externally to constituent collaborations. This allows collaborations to be used simultaneously in various parts of the application and also to allow the precise implementation chosen to be varied. For example, a simple file-based persistence collaboration can be switched for a more advanced collaboration that talks to a third-party database if performance becomes an issue.

4. **Non-type-invasive extension.** Constituent types are translated to output types during composition; thus no constituent types occur in a composed collaboration. Thus, there is no possibility of confusing two uses of a collaboration, as neither use results in references to the collaboration's original types. By writing collaboration behavior over place-holder participant types, this feature can be leveraged to provide subtype [13] as well as class-parametric aspectual polymorphism along the lines of [14, 15].

5. **Non-behavior-invasive extension.** Collaborations can be extended in two ways. The composite collaboration contains the combined behavior of each of its constituents (even if it cannot export all contained behavior), which can be viewed as extending the constituent's behavior. Additionally, aspectual methods can intercept certain method executions in a constituent collaboration, allowing the collaboration to be transparently extended with additional behavior.

6. **Method interception.** In addition to transparent extension, aspectual methods can modify how another method is executed, allowing arguments to be inspected and exchanged, even controlling if a method execution continues at all. Encapsulation controls which methods can be modified in this manner: only methods that are exported by a constituent collaboration can be intercepted. Thus, the collaboration controls which methods can be modified and which cannot.

7. **Generic advice.** By allowing the programmer to choose how signature-specific an aspectual method is, reusability can be traded off for expressive power. The signatures in aspectual methods are affected by type translation like all other types in a collaboration. An aspectual method can require an argument of type Foo, which is a participant of the collaboration. By mapping Foo to a type determined by the signature of the intercepted method, the aspectual method can safely talk about the captured argument (including declaring variables with that type and storing the value in containers) without constraining it to any particular type. Non-type-invasive extension thus allows generic advice to access arguments of captured method executions yet remain type safe.

## 4.2. Limitation

As a research prototype and proof-of-concept, ACs have the disadvantage that they are not as expressive as AspectJ for some problems. For example, ACs are less elegant than AspectJ at expressing complex join point predicates, such as the object-form of the Law of Demeter [16]. How ACs should be extended to elegantly capture such complex join point predicates is an open question.

## 4.3. Getting concrete: an example

The distinction between atomic and composite ACs is purely conceptual: all collaborations have a (possibly empty) set of locally declared behavior and compose a (possibly empty) set of external collaborations. Typically, a collaboration will either be atomic with only locally declared behavior, or composite with only imported behavior—in general, any combination is possible. The common syntax is

> **collaboration** *name*
> { **extends** *collaboration* }*
> { **participant** *formal_class_def* }*
> { [ **match** *role_constraint*\* ] **attach** *attachment_spec* }*

An AC has an intrinsic name and declares a set of participants: participants are declared explicitly, or extended (with copy semantics) from external collaborations. These participants are called *output participants*, to distinguish them from the *constituent* participants of attached collaborations. The set of such participants is closed, in that no participants can be added to the collaboration at a later date without recompilation of the whole collaboration. This is in contrast to Java packages, which can be extended with new classes without recompiling the existing classes.

A collaboration's behavior consists entirely of the members on its participants. An output participant's members are either explicitly declared along with the participant, or added by attaching an external collaboration and mapping one of the constituent participants to the output participant.

External collaborations are attached using one or more *attach clauses*. Each clause can attach several collaborations to the output collaboration, mapping the constituent participants to the output participants. The operational effect of such a mapping is to insert all the members from the constituent participants into the corresponding output participant. Additionally, all occurrences of the constituent

**Listing 9.** Defining generic setters and getters.

```
1  collaboration addGetSet
2  participant HasAttribute {
3    expected String aString;
4    public String get() {{
5      return aString; }}
6    public void set(String aString) {{
7      this.aString = aString; }}
8  }
```

**Listing 10.** Attaching setters and getters for name.

```
1  collaboration baseGetSetName;
2  extends base;
3  attach addGetSet {
4    Container += HasAttribute {
5      provide aString with name;
6      export set as set_name;
7      export get as get_name;
8    }
9  }
```

types in the inserted members will be replaced by the corresponding output participant.

Attach clauses can either be written out manually, or generated from a template clause and a *match clause*. The match clause is evaluated against the output participant graph to generate a set of name bindings that generate attach clauses from the template clause.

A simple AC, consisting only of locally declared participants, has the same shape as a Java package, substituting **collaboration** for **package** and **participant** for **class**. A Java class is a limited form of participant and a Java package is simultaneously a collaboration. The host package base (Listing 1) is thus a legal collaboration, corresponding to Figure 1. Conversely, a collaboration containing no unprovided expected members can be used like any Java package.

A slightly less trivial collaboration—in that it uses participants and has an expected member—is the addGetSet collaboration shown in Listing 9, which given a field on a participant will add a getter and setter method for that field. It declares one participant, HasAttribute, on line 2, two methods, get and set, on lines 4 and 6. The expected field on line 3 declares that the collaboration *requires* a reference to a field. This example promotes simplicity over genericity, hard-wiring the type of the field to be String. The double braces around method bodies are merely an implementation trick—allowing us to avoid parsing method bodies—and can be read as single braces.

To use addGetSet, we need to compose it with another collaboration, mapping the participant HasAttribute against a participant (or class—remember that classes can be used in place of participants) with the field in need of get and set methods and also *provide* that field to aString. Listing 10 does exactly this against the base package of Listing 1. By extending base (Listing 10, line 2) the output collaboration baseGetSetName (as per Listing 10, line 1) will contain all the participants from base: Item, Simple, Container and Main. For pragmatic reasons, all participants declared via extension have all members exported by default, which is the opposite of the case for attached collaborations, whose members are unexported by default. To the output participants we attach addGetSet (Listing 10, line 3) and map Hasattribute to Container. Mapping is performed with the **+=** mapping operator, which is chosen to resemble the accumulate operator in Java or C, as it accumulates all the constituent participants into an output participant. In addition to the decoration, **+=** also redirects any references of the inserted type, HasAttribute, to the destination type,

**Listing 11.** Caching a return value.

```
1   collaboration caching;
2   import java.util.*;
3   participant Cached {
4     ChdRetVal cachedValue;
5     void clearCache() {{
6       System.out.println(getName()+" clear cache");
7       cachedValue = null;
8     }}
9     expected Vector allInvalidated();
10    expected String getName();
11    aspectual RV invalidate(EM e) {{
12      RV retval = e.invoke();
13      Iterator inv =allInvalidated().iterator();
14      while (inv.hasNext()) { ((Cached)inv.next()).clearCache(); }
15      return retval;
16    }}
17    aspectual ChdRetVal cache(ChdMth e) {{
18      if (cachedValue==null) { cachedValue = e.invoke(); }
19      else { System.out.println(getName()+": using cached"); }
20      return cachedValue;
21    }}
22  }
```

Container.[4] Line 5 provides the concrete field name to the expected field aString, while the provided get and set methods are exported under more suitable names on lines 6 and 7. The result is that baseGetSetName contains the behavior of base and is extended with a getter and setter for capacity.

The challenge problem will be used to motivate other features of ACs.

## 5. ASPECTUAL COLLABORATIONS

The organization of the AC solution is similar to those of AspectJ and Hyper/J. There are two generic and reusable collaborations (Listings 11 and 12), one glue collaboration (Listing 13) which adapts one interface to the other and, finally, an attachment of all three to the base package (Listing 14). To illustrate composition we present two versions: the first attaches to the version of base which has been augmented with a getter and setter for the name variable (baseGetSetName in Listing 10) and the second which instead composes the get and set augmentation into the solution, before attaching the composite solution to the original base package of Listing 1.

---

[4]By redirecting references from AttributeType to String, we achieve type parameterization as a degenerate case of attachment [14, 17, 18].

**Listing 12.** Add and maintain backlinks.

```
1   collaboration backlink;
2   import java.util.*;
3   participant Source {
4     expected Vector targets;
5     aspectual RV associate(ModifyM e) {{
6       int sizebefore = targets.size();
7       RV rv = e.invoke();
8       int sizeafter = targets.size();
9       Target targ = e.t;
10      if (sizebefore<sizeafter) { targ.back = this; }
11      else { targ.back = null; }
12      return rv;
13    }}
14  }
15  participant Target {
16    Source back;
17    Source getSource() {{ return back; }}
18  }
19  participant ModifyM {
20    expected Target t;
21  }
```

**Listing 13.** Generating a transitive closure.

```
1   collaboration adapt;
2   import java.util.Vector;
3   participant Item {
4     expected Container getContainer();
5   }
6   participant Container extends Item {
7     Vector allContainers() {{
8       Vector v = new Vector();
9       Container c = this;
10      while (c != null) {
11        v.add(c);
12        c = c.getContainer();
13      }
14      return v;
15    }}
16  }
```

Because of the similarity to the AspectJ and Hyper/J solutions, this section focuses on how Aspectual Collaborations capture the necessary language concepts, rather than explaining the functioning of the concerns.

To maintain caching and backlink invariants, the concerns need to intercept and modify executions of methods in the base. We describe aspectual methods, the mechanism ACs provide to allow *advice* to be attached to *host* methods, while allowing the definitions of both aspectual and host methods to remain mutually oblivious. We then show how this capability allows us to implement the concerns in an elegant manner.

### 5.1.  Aspectual methods

A concrete use of aspectual methods is caching a method's return value, as implemented by caching in Listing 11. The collaboration has one participant with two aspectual methods: cache, to intercept the executions of the method to be cached, and invalidate, which intercepts the methods that invalidate the cache. The collaboration also has two

**Listing 14.** Attaching our collaborations.

```
1   collaboration cachedbase;
2   extends baseGetSetName;
3   attach backlink, caching, adapt {
4     Item += Target, adapt.Item {
5       provide getContainer with getSource;
6     }
7     Container += Source, Cached, adapt.Container {
8       provide allInvalidated with allContainers;
9       provide targets with elements;
10      provide getName with get_name;
11      around void addItem(..,Item t,..) do associate;
12      around addItem do invalidate;
13      around weight do cache;
14    }
15  }
```

expected methods: allInvalidated, which should return a vector of the objects to be invalidated, and getName, returning some String to be used for printing. There are also two normal Java members: cachedValue, which stores the cached value between invocations of the cached method, and clearCache, a method which sets the field to null.

The aspectual methods are declared (Listing 11, lines 11 and 17) with the keyword **aspectual** and a stylized method signature

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

where both RetVal and MethM are user-chosen participant names that are either undefined or defined locally to the collaboration. The user thus names the types that reify the intercepted method execution and its return value.

Reifying the intercepted return value of cache as ChdRetVal (Listing 11, line 17) allows us to treat it as a first-class object: storing it in a variable (Listing 11, lines 18 and 4) and possibly returning it at a later date (Listing 11, line 20). Similar arguments hold for the benefit of reifying the method execution.

Note how the reification API, which encapsulates the details of the wrapped method and intercepted calls to it, allows us to write a caching collaboration which is completely oblivious to the signature of the cached method or its invalidator. Additionally, this illustrates a scenario in which the aspectual method chooses *not* to invoke the wrapped method, instead returning the result from a previous invocation. Both of these are made possible by our reification of the connection between concerns into the domain of the language.

To maintain type safety, the reification types chosen for the caching collaboration (ChdRetVal, ChdMth, RM and EM) are subject to two additional constraints over those of a normal participant.

1.  ***Uniqueness.***   Each reification type must occur in exactly one aspectual method. Since participant types are qualified by their collaboration, several collaborations may contain the same reification unqualified type name.

2. ***Existential quantification.*** The collaboration is existentially quantified over the reification types. This implies that the reification participants cannot appear outside the collaboration's textual extent. Aspectual methods can be exported from collaborations, but the reification participants they define cannot be mapped against output participants. Note that the types are all that are existentially quantified. There is no restriction on objects of these types escaping the scope of the collaboration: an instance of ChdRetVal may be upcast to Object and stored in a file or globally visible collection.

These rules provide the restrictions necessary to guarantee that aspectual methods remain type safe yet generically applicable to method executions of differing signatures.

Reification types, in addition to being automatically generated, if need be, are given a default API. The API for a method-execution participant (ChdMth or EM) is

> **expected** RetVal invoke();
> **expected** RetVal dontInvoke();

while default return-value participants have an empty API. These APIs can be optionally extended to provide access to method arguments, return values, and thrown exceptions. If the reification participants are completely oblivious to the signature of the method call and return value they represent, an aspectual method can intercept method executions of any signature. Adding capabilities to the API comes at the price of constraining which method executions an aspectual method can intercept. For example, if we expose an argument on the method-execution participant, the aspectual method is restricted to advising executions of methods with an argument of appropriate type.

The dynamic behavior of an aspectual method which is mapped to a *host* method is as follows (for concreteness, we will explain an execution of a method mapped to cache).

(a) The [unknown] host method is invoked by some caller.

(b) The system intercepts the execution, packages it up as a ChdMeth object and invokes cache with the method-execution object as only argument: e.

(c) If the method is not cached currently (i.e. cachedValue is null in Listing 11, line 18), the host method reified as e is invoked.

(d) The invoke method proceeds with the [still unknown] host method, passing any captured arguments and intercepting any results (including thrown exceptions) of the original method.

(e) These results are reified as a ChdRetVal object and passed back to the aspectual method, which stores them in the instance variable cachedValue.

(f) If there already was a cached value, Listing 11, line 19 prints a short message including the string from the **expected** getName method.

(g) The cachedValue is returned from cache in Listing 11, line 20. The host is only invoked if there is not a value currently cached.

(h) Lastly, the returned ChdRetVal object is unpacked and the host-method execution's return value is returned or, in the case of an exception, re-thrown. The original caller and the advised host method remain oblivious that any interception has taken place, while the aspectual method is likewise oblivious to the host method.

### 5.1.1. Caching collaboration

From a caching perspective, the design is straightforward, closely following the design of the AspectJ and Hyper/J solutions. There are two use scenarios for the caching collaboration. We have already explained the steps involved in caching a value; how to invalidate cached values remains to be explained.

To cache a method's executions, caching is attached to the host collaboration and Cached is mapped to the participant with the method that needs to be cached. The aspectual method caching is wrapped around the method to be cached and invalidate is likewise wrapped around the method whose executions invalidate the cached value. These aspectual methods rely on two expected methods: allInvalidated should return a collection of objects whose caches need to be invalidated on executions of invalidate, while getName returns some name string of the receiver object.

Like cache, invalidate is invoked whenever the method it is advising is called. The invalidate method implements after advice: it immediately (Listing 11, line 12) proceeds with the intercepted method.[5] It then iterates over all the objects to be invalidated—calculated by whatever implementation was provided to allInvalidated (Listing 11, line 13), calling clearCache on each one (Listing 11, line 14). Finally, invalidate returns the reified return value from the wrapped method.

The collaboration assumes that there is exactly one method that invalidates the cache *and* that the method is on the same object as the one cached. The restriction can be alleviated both by programming patterns and language features not introduced in this paper. However, this would have complicated an already large example.

The clearCache method just sets the cachedValue variable to null (Listing 11, line 7). One way AC's generic API differs from AspectJ is that ACs distinguish between null values and reified representations of null values. The call to an invoke or dontInvoke method on a reified-method object is guaranteed to return a non-null value, regardless of whether the intercepted method returns a value, returns a void, or throws an exception. Thus, a null value stored in cachedValue is an unambiguous representation of a clear cache.

To use the collaboration, participant Cached will be mapped to Container, with cache advising check and invalidate advising addItem. However, we have not dealt with how allInvalidated and getName will be implemented.

---

[5]It would, of course, be trivial to provide syntax for the common cases of before and after methods that do not wish to control or inspect the reified method, but would add yet another feature to present, without adding to the power of the language.

### 5.1.2. Backlink collaboration

The second 'main' collaboration is backlink (Listing 12) and corresponds to the concern of adding and maintaining a backlink from an item to its parent container. Its behavior involves two main participant roles. The Source participant expects to be provided with a variable holding a vector of Target objects and to intercept method executions that modify that vector.

When such a modifying method execution is intercepted, associate (Listing 12, line 5) takes over, with the intercepted method-execution reified as a ModifyM object. The first thing associate does (Listing 12, line 6) is to capture the size of the targets vector, before invoking the wrapped method (Listing 12, line 7). associate requires access to the object added or removed, which is presumed to be an argument of the intercepted method execution. The argument is named and typed by adding an **expected** variable t to the ModifyM participant (Listing 12, line 20), thereby constraining associate to wrap only methods taking (at least) an argument of type equivalent to Target. We can use the expected field t on ModifyM to access that argument (Listing 12, line 9). Depending on whether the vector grew or shrank, we either set the back-pointer to this (Listing 12, line 10) or to null (Listing 12, line 11), maintaining the declared invariant.

The participant ModifyM is a reification participant (as per the signature of associate in Listing 12, line 5) and is thus treated somewhat differently from other participants. Notably, it will not be mapped against another type during composition. Additionally, it is only partially specified: the system will automatically add the members that constitute the API that reification participants provide. The examples in this paper faithfully represent the prototype implementation's syntax, but it is easy to imagine an equivalent syntax in which reification participants are never explicitly declared.

Let us briefly look ahead to Section 5.2 to see how this collaboration will be related to the host collaboration, base. The result of the aspectual behavior is to maintain an invariant that Target objects know which Source object points to them. What we want to achieve is to have Items knowing in which Container they are stored. Thus Source will need to be mapped to Container (from Listing 1) and Target to Item. The method associate will be wrapped around addItem and we can guess that the variable **elements** will be provided to targets. The host classes are related by inheritance, while the collaboration classes are not. This will not pose a problem for attaching backlink to base, but will influence the requirements for the adapt collaboration.

### 5.1.3. Adapter collaboration

Since base.Container contains both the method we wish to cache (check) and the method to invalidate the cache (addItem), we have already deduced that we are going to need to attach caching.Cached to base.Container. At this point we would start to write such an attachment specification, but we notice that we do not have a suitable

implementation for the Cached's allInvalidated method: a method that returns *all* parent containers of this. We *do* have a method that returns the immediate parent container: Target's getSource. Thus we have the programming task for the adapt collaboration: create a method that returns the transitive closure of getSource.

The behavior of adapt in Listing 13 is straightforward: allContainers merely calculates the transitive closure of getContainer. This collaboration is written in a slightly different style than the others, in that it is written with intent to be used once, in one known context. Hence, we make it as easy as possible to attach, exactly mimicking the inheritance structure of the intended use and even using the same participant names.[6]

Note that the collaboration's participant-graph mimics the inheritance structure and names of base. While the similarity in names to base is immaterial, the identical inheritance structure is important. We intend getContainer to be provided with getSource, which is defined on Item, while the method we are creating, allContainers, will be provided to allInvalidated, which is defined on Cached (which will be mapped to Container). As the method required by adapt will be from a super class of the class receiving the method the collaboration provides, we are required to mimic this inheritance structure in the collaboration. The inheritance structure of collaborations can be manually adapted via composition and delegation, but the analyses needed to perform such adaptation automatically have not yet been fully investigated.

We also need a method to map to Cached's getName method. Three solutions present themselves: (a) use the composed base from Listing 10 which exports get_name and set_name; (b) attach addGetSet along with caching, backlink and adapt; or (c) compose addGetSet with caching to produce a caching collaboration that expects a field rather than a getter method. We choose option (a), accumulating behavior onto the base, with each subsequent version getting a new package name.

### 5.2. Attachment

Listing 14 illustrates the generation of a new collaboration with the name cachedbase which combines the functionality of baseGetSetName ('the base' hereon) with the collaborations we have defined.[7] We can extend and augment composite collaborations (such as baseGetSetName) just as we can extend atomic collaborations (such as base). The collaborations that we attach can likewise equally be atomic or composite. This allows us to incrementally add behavior to an increasingly enriched base, or to instead compose complex modules that can be added atomically to the original. Listing 14 illustrates the former option.

---

[6]The correspondence in names is for clarity only. All attachment is explicit and the behavior is exactly the same as had we chosen more arbitrary names. It also lets us exercise class disambiguation in the attachment.

[7]As all expected methods have been provided, this collaboration is simultaneously a normal Java package.

**Listing 15.** Attaching our collaborations.

```
1  collaboration cachingbis;
2  extends caching;
3  attach addGetSet {
4    Cached += HasAttribute {
5      export aString as name;
6      provide getName with get;
7    }
8  }
9  collaboration cachedbasebis;
10 extends base;
11 attach backlink, cachingbis, adapt {
12   Item += Target, adapt.Item {
13     provide getContainer with getSource;
14   }
15   Container += Source, Cached, adapt.Container {
16     provide allInvalidated with allContainers;
17     provide targets with elements;
18     provide Cached.name with Item.name;
19     around void addItem(..,Item t,..) do associate;
20     around addItem do invalidate;
21     around weight do cache;
22   }
23 }
```

The attachment sets up point-wise mappings between the participants of the constituent collaborations (caching, backlink and adapt) and the output participants imported from the base Item (Simple, Container and Main). Note that we only mention the participants we are interested in, as the semantics of **extends** has implicitly exported all of the base (this can be overridden). Likewise, the member mappings we have alluded to in the descriptions of the constituent collaborations are set up; for example, expected field targets is provided with field elements.

The attachment processes each participant separately. To create the output participant cachedbase.Item, we start with base.Item (implicit from **extends** in Listing 14, line 2) and insert the constituent participants backlink.Target and adapt.Item (Listing 14, line 4) into it. We do not need to fully qualify Target, as there is only one participant with that unqualified name, but there are two Item participants (from adapt and the output collaboration), so we must disambiguate. Once created, the output participant Item's members are linked, providing the expected method getContainer with the implemented getSource. Again, since these member names are unambiguous, we do not need to specify from which constituent participant each came.

Container is similar, but there are pertinent observations concerning the attachment of aspectual methods. We set up cache to intercept the invocation of check (Listing 14, line 13), at which behavior proceeds as explained for caching. Similarly for addItem.

It is not obvious which wrapper of addItem (Listing 14, line 11 or Listing 14, line 12) is executed first. One answer is that it should not matter, as each collaboration should be somewhat semantically complete, and if two collaborations are so tightly coupled that their relative order of evaluation matters, this suggests that they are better combined

than separated. A slightly more satisfying answer is that invalidate will be invoked first, as **around** wrappers are processed in program order and each processing stage builds on the previous method implementation. This allows the programmer to flexibly control the order of advice: within one attachment, advice from different collaborations can be arbitrarily interleaved.

One can view the host method and its wrappers like a *Matrioshka* doll, where each doll has control over whether the dolls inside itself are executed. The original addItem is the innermost doll, which is reached last—if at all. The outermost doll represents invalidate, which has control over the invocation of associate.

Finally, the correspondence of the argument name Item t in the partially specified signature of addItem (Listing 14, line 11) to the expected variable in backlink.ModifyM (in Listing 12) *is* important. This specifies that the first argument of type Item in method addItem is to be exposed to the backlink aspect, as the expected field t.

### 5.2.1. Alternative attachment

As a quick demonstration, Listing 15 implements option (c) from the list of composition alternatives: it creates a new caching collaboration, cachingbis, which is just like caching (Listing 15, line 2), but expects a field name (Listing 15, line 5) rather than a getName method (Listing 15, line 6). This caching collaboration can of course be attached directly to the original base collaboration, which is illustrated in Listing 15, lines 10–21.

### 5.3. AC compiler

The prototype AC compiler, acc, follows a fundamental design: collaborations are compiled separately and composed at the object-code level. The object code the compiler works on is Java byte code, which turns out to be very convenient since it provides a fully disambiguated version of the collaboration. Byte code is managed as standard Java .class files, which results in the easy renaming of members, re-targeting references to point to other members and moving call graphs from one group of classes to another.

Figure 6 is a data-flow road-map of how collaborations are compiled to executable programs. Rather than compiling the whole language directly, input collaborations are translated into *core ACs* (CACs), which are transliterated into Java, compiled and finally manipulated at the bytecode level. As several important steps will be elided in the following presentation, we refer the interested reader to the compiler itself [19].

CAC participants are compiled by transliterating them into Java and compiling with an off-the-shelf Java compiler. The transliteration replaces **collaboration** with **package**, **participant** with **class**, comments out all additional keywords we have introduced (such as **aspectual** and **expected**) and creates stub bodies for expected methods. After compilation, the .class files are annotated according to the commented-out keywords to allow the compiled collaboration to be treated correctly in the subsequent
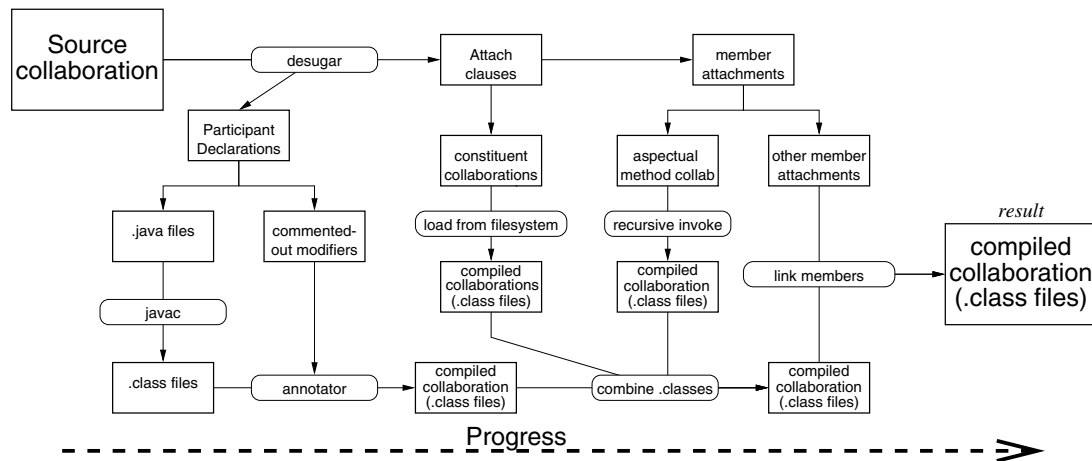
**FIGURE 6.** A data flow over-view of the compiler for AC.

stages without requiring manifests or interface files to be maintained in the file-system. The JVM definition [20] requires that a JVM ignore all annotations it does not understand, ensuring that the compiled collaborations consist of 100% legal `.class` files.

Having collaborations as legal Java packages leads to a useful feature. The JVM requires that all non-abstract methods have method bodies, so normally acc generates short stubs for expected methods. Allowing the user to optionally provide stub method bodies for expected methods is a convenient means to unit test collaborations in isolation.

To compose two collaborations according to an attachment, we first generate output participants. CACs do not have **extends** clauses, instead having explicit definitions of all participants, including those representing reified methods and return values. These are compiled as per the atomic case.

Each constituent collaboration is then inserted into the output collaboration, according to the participant map of the attachment. Each constituent participant is inserted into the appropriate output participant, with each of its members renamed to names which are both fresh (to avoid name clashes) and illegal in Java (to avoid spurious name clashes).[8] References to participant types and member names are systematically renamed to maintain collaboration internal references. We call the systematic renaming of an entity and all references to it *alpha renaming*. Alpha renaming is possible because ACs are closed: the compiler will be able to statically find all references by searching all participants in the collaboration.

Some care must be taken not to break overriding relationships: methods must be renamed like the methods they override. If a method overrides an external method (such as toString, defined on the global superclass Object), it cannot be renamed at all and name clashes can become unavoidable. In such cases, manual disambiguation is required. Java constructors have similar restrictions (all constructors are hard-wired with the name <init>), in addition to restrictions on how they can be invoked.

References inside the now-populated participants are redirected as per the attachment specifications. Exporting a member is implemented by alpha-renaming the member to the exported name. Providing a member to an expected member is implemented by redirecting all references from the expected member to the provided member.

Aspectual methods' interception of method executions is implemented by generating a collaboration with three participants corresponding to the host participant and two reification participants. The generated collaboration is based on the signature of the now known host method. A new intercepting version of the host method is generated, whose body instantiates the reified-execution participant (storing in it the receiver) and arguments to the method call and passing it to the aspectual method. The reified method participant's invoke method invokes an **expected** method that will be mapped to the original host method, passing it the captured arguments and instantiating a reified return object with the results of that call. Finally, the now generated collaboration is composed into the output collaboration as if it were a manually written collaboration—requiring no special language support—so that the new host method is exported to replace the original host method and the original host method is provided so that invoke can access it.

The generated code needs to be carefully crafted to catch exceptions and null values, but contains no locations where dynamic errors (such as casts) can occur. Also note that the scheme requires no modification of either the aspectual method or the host method: the extent to which the composition modifies object-code is to redirect references to members and types.

### 5.4. Future work

There are several subtle implementation issues that need to be dealt with in future work. These are issues that we have solved at the language design level, but have not yet implemented.

***Parameterization.*** By adding types to collaborations, it becomes feasible to express attachments which are

---

[8]The JVM has a larger set of legal identifiers than Java does.

abstracted over the exact collaboration attached. By passing collaboration-valued parameters as arguments to attached collaborations, very complicated behaviors can be succinctly expressed. The challenge is to develop a type system that allows flexible use while capturing errors early. Errors will always be caught at compilation time, but it is desirable to catch them when the parameterized collaboration is defined, rather than instantiated.

**Point-cuts in the interface.** The interface to an AC currently contains only participants and members. In order to add advice to a member, it must be in the interface, visible and, hence, invokable. It is desirable to decouple these concepts, by putting sets of execution points that can be advised directly into the interface of an AC. Such sets of points correspond to point-cuts in AspectJ, but will generally be sets of tuples of points rather than singleton points (a singleton point is a special case of a tuple). We are currently investigating the precise semantics of such a system and are in the process of investigating what repercussions such a change implies.

**Constructors.** The main problem with constructors is that they are the only methods which should be merged, rather than kept separate. When two constituent participants are mapped to the same output participant, we want the creation of an output-participant object to invoke all three (two constituent and one output) constructors. Each participant may actually contribute several constructors, which in turn may lead to several inheritance chains of constructors. Since a constructor may invoke any method of a class, we have to be careful about the order in which class initialization methods are invoked. Our current solution is to have a sensible heuristic and ask the user to specify in the situations where that does not apply. We note that Jiazzi [21] has a similar restriction, requiring all constructors in an inheritance chain to have exactly the same signature.

**cflow.** AspectJ introduced the **cflow** construct for capturing that the target join-point occurs within the dynamic extent of an enclosing join-point—similar to how one might inspect the run-time stack to see whether a program point is within the dynamic extent of some method. We can easily achieve similar effects by generating aspectual methods and attaching them to the enclosing and target methods. Our systematic reification of method calls makes it a natural option to expose the enclosing method call to the target advice as well. This opens up the exciting concept of allowing an aspectual method to possibly modify the enclosing method call's arguments and then restart the whole call chain.

**Change the back-end.** Currently, `acc` translates a collaboration to plain Java, compiles it and then uses the generated byte-code for the attachment and composition operations. However, the mechanics of the byte-code manipulation are tedious and are not the contribution of our research. Instead it may be possible to offload this development burden to a back-end based on Jiazzi, which would allow us to focus on developing the module interface language and aspectual features.

### 5.4.1.  Other extensions

In addition to implementation issues, the following are natural extensions to this work.

**Object graph constraints.** A key concept of collaborations is that each has its own class-graph, which are fused when one is attached to another. The behavior of a class-graph will, in general, instantiate classes of that class graph and store the objects in variables. In effect, each collaboration will build its own object-graph. In addition to *building* an object graph, the collaboration also makes assumptions about it—these assumptions are encoded in the code of the collaboration and take the form of invariants over the object-graph.

Examples of invariants are that a non-zero value for one variable indicates that another is ready to be read, or that two variables of the same type in fact *alias* the same object. The key insight here is that the fused collaborations must make compatible assumptions about their object-graphs, as in addition to sharing a fused class-graph after attachment, they will also share an object-graph at runtime.

It would be helpful to capture these constraints in the interface of the collaboration, so as to be able to catch such attachment errors at compile-time. This can be seen as a special case of contract checking, where perhaps machine analysis can help derive the object-graph (run-time) constraints to be checked at compile-time.

**Garbage-collect participants.** Because collaborations are closed behind an encapsulation interface, we can statically determine all participants and members reachable from the exported interface. Thus, we can safely remove all non-reachable participants and members, retaining only the exported interface and any non-exported but internally referenced members. This smaller collaboration is very similar to a teased-out HyperSlice in Hyper/J. To completely simulate a HyperSlice, we would need to add the ability to 'unprovide' a member—to drop a member's implementation and thus make it **expected**: a small modification to the current implementation.

## 6.   EVALUATION AND RELATED WORK

Hyper/J and multi-dimensional separation of concerns [4] generalizes the ideas behind subject-oriented programming [22, 23] by moving to finer grained units of combination. A HyperSlice is a named set of classes containing sets of methods and fields and can be added to new classes in a very similar way to collaborations.

Our comparison found that HyperSlices fare well in capturing and composing reusable concerns in a type-safe manner, but lack sufficient expressiveness to conveniently perform behavioral crosscutting. While Hyper/J's HyperModule definitions make it explicit *how* concerns are composed, alleviating the 'come-from' nature of aspects, the seeming lack of encapsulation makes understanding *what* concerns are composed of impossible without performing a full trace of the concern composition history to gather up the

accumulated interfaces for the slices and opening up slice composition to inadvertent name capture [10].

Hyper/J additionally offers several features that did not come up in the example. *Post-hoc* remodularization allows a HyperSlice to be teased out of a set of classes (possibly generated by composing slices, or by compilation of `.java` files) and used separately. Hyper/J also provides several dimensions of composition, of which our examples only used the feature dimension.

AspectJ from Xerox PARC [3, 24] resulted from the initiative to factor out commonalities in several domain specific aspect languages. Lopes' thesis [25] investigated two of those languages in detail: COOL [26] for specifying the synchronization aspects and RIDL [27] for specifying data transfer aspects.

By integrating aspect features tightly into the language, AspectJ forgoes the semantic and syntactic overhead of module systems, but also the benefits of encapsulation and separate compilation. While this gives rise to very natural specification of aspectual behavior, it comes at a cost of program comprehension, as the lack of encapsulation boundaries for advice forces the programmer to read the whole program to determine whether a join-point has advice. Analogously, there is no way to protect a join-point against being advised. These deficiencies have been addressed with tool support, rather than language features, aiding the programmer in identifying advice attached to methods.

AspectJ aspects are reusable by programming abstract aspects against interfaces that are attached at a later time to the host program. Aspectual advice achieves surprisingly good reuse, both by mentioning only the types necessary in a point-cut signature and also by the somewhat unorthodox genericity mechanisms of around methods (see Section 3.2.4), which appear to work very well in practice.

Unfortunately, programming abstract aspects against interfaces suffers from low levels of type safety: generating casts in programmer-invisible code that can fail and forcing multiple uses of an aspect to share types (recall Section 6.1), which opens up the program to further casting errors. The use of interfaces interferes with reuse, as it restricts expected members to be methods and requires name equivalence between expected and provided methods.

Hannemann and Kiczales [28] show how this style of programming can be used to implement standard design patterns. It identifies four properties by which such an implemented pattern can be considered modularized. The properties focus on textual invasiveness and locality of the design, rather than properties such as encapsulation and composition identified in this paper. The organization removes confusion about which instance of a pattern source code belongs to, but whether objects playing pattern roles can be confused between pattern instances is not discussed.

## 6.1. Invasiveness

Ernst [6] defines *invasiveness* as the ability to tell, from within a module, how it is being extended and used.

An extension can be both type and behavior invasive. Using Java interfaces can result in both kinds of invasive extensions.

The use of interfaces and 'implements' to model roles affects the static correctness of a program, as well as its dynamic type safety. The key property of a Java interface is to assert that a class satisfies the properties necessary for the type defined by the interface. Thus, a class must provide implementations of all methods in an implemented interface, or be annotated as **abstract**—in which case its subclasses inherit the burden of providing implementations.

An aspect can render previously correct base classes incorrect by introducing an 'implements' relationship between a base class and an interface: unless the class has the required methods, it must be marked abstract. Consequently, care must therefore be taken to additionally introduce implementations of all methods required by an interface, lest the extension behavior invasively (as per Ernst and Section 2.2) affects the base. In our example, the Adapt aspect introduces the necessary methods to allow the base classes to fulfill the requirements of the abstract aspects.

The use of interfaces can also render an aspect type-invasive. A fundamental difference between how Hyper/J and AspectJ model roles is that in Hyper/J there is no possibility of confusion between a Source object in one use of the Back slice and another. This follows from the fact that none of the input types in a composed HyperSlice remain after composition.

The opposite situation is true for any use of interfaces (including AspectJ's external 'implements' declarations)—all classes that implement an interface share a common supertype: the interface. Thus, if the Back aspect is applied to two different source and target pairs, each of the source classes will be upcastable to Source. This has two effects: invasiveness and the possibility of type confusion.

Type invasiveness follows from the code anItem instanceof Target testing whether the back concern has been attached to anItem. Ironically, while it *is* apparent at runtime that Item implements Target, this is not apparent statically by inspecting the code, unless we come across SuperimposeBack. A naming convention can easily ameliorate this, however.

The objection to runtime composition tests may appear more of an aesthetic rather than software engineering issue, but the mechanism of attaching participants by implementing interfaces has consequences for type safety as well. If the back collaboration were attached twice in an application, there would be two classes that now have the common supertype Target (there may be additional such classes in library code). The problem is that the behavior provided to the application by the concern is written against the interface types, while the methods implementing the concern's required interface are likely to be casting these types to the implementation classes. That is, it is possible to pass *any* Target object to a method with that argument type, but that method's implementation may likely assume that the Source returned by getSource is actually a Container. This assumption would be foolish in normal

code implementations of an interface, but since a concern is declared as a unit, it makes sense for the programmer to assume that the attachment is also a unit. Our examples do not expose participant types in the expected method signatures, so this case does not appear with ACs.

## 6.2. Module systems

Jiazzi [21] is the implementation of the Units [29] module system for Java. Jiazzi reuses Java's core composition feature—inheritance—and the Open Class Pattern to construct the resulting classes from partial implementations. Explicit late binding (via the Open Class Pattern [30]) is used to allow mutually recursive dependencies between modules. A Jiazzi implementation of the caching example would probably look very similar to the Hyper/J version, but it is unclear whether it is possible to express generic interception of method executions in Jiazzi.

It is interesting to note that although developed completely separately, the current back-end for ACs and Jiazzi are strikingly similar. The main differences are how the finished classes are assembled (Jiazzi favors inheritance, while we manually combine and link participant `.class` files) and the fact that we favor intrinsic typing for collaborations, while Jiazzi allows the reuse of a unit type for several unit implementations.

## 6.3. Component systems

ACs are not able to conveniently express, nor guarantee the communication integrity presented in ArchJava [31, 32] for dynamic component connection, but are able to do so quite well for the static case. Using encapsulation, we are able to make statements about not only which components a component is able to talk *to*, but more strongly which components it can talk *about*. If a component does not import another component's type, direct communication between them is impossible. This additionally applies 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 field), then we can statically guarantee that this field 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.

Adaptive plug and play (AP&P) components [33] and the follow-on report [34] are the immediate precursors to ACs. AP&P components are rooted in Holland's executable contracts [35] and in Rondo [36]. This work builds on [34], but with significant modifications from experience with implementation and with a very different attachment and matching model stemming from clarified semantics. The difference in attachment and matching reflects that AP&P components are aimed at being language-level components rather than system structuring modules. Additionally, the AP&P components offered somewhat weaker aspectual capabilities.

Herrmann and Mezini [37] discussed a software engineering environment capable of combining dynamic pluggability, separate compilation and aspectual attachment. It is unclear how their PIROL system deals with type safety.

Mezini and Ostermann [38] propose a component system that incorporates ideas from AP&P components, but separates implementation from interface. The interface of the components is expressed as Java interfaces, which are implemented by the implementation types of the component. They suggest the use of families of dependent types [39] to solve the typing issues of shared supertypes for multiple instances of a component. Type families are believed to be statically sound, but this has not been formally proven. Type families would probably prove a convenient resolution to the issues we have raised concerning the type-invasive use of the Java interface for reusable aspects in AspectJ.

The components are connected to their host objects by runtime attachment, which is used to provide implementations for expected methods—a commonality to Aspectual Collaborations which exposes the shared ancestry of AP&P components. Runtime attachment buys great flexibility, allowing attachment to be performed at the granularity of individual objects rather than classes and behavior to be varied over an object's lifetime. Mezini and Ostermann suggest that such runtime attachment may be a good base for *fluid AOP*, which likewise would allow aspects to be attached and detached dynamically.

## 6.4. Modeling languages

Clarke and Walker [40] add the concept of composition patterns to UML [41]. The implementation suggestions for composition patterns [7, 42] strongly influenced the expression of reusable aspects and hyperslices in this paper and we propose that ACs do a particularly elegant job of representing composition patterns. It is unclear whether composition patterns capture multiple attachments of a collaboration and how the sharing of members between such attachments would be expressed.

## 7. CONCLUSION

The mechanisms that AOP languages use to incrementally add a concern to an application at first seem contradictory to those that a module system uses to provide manageability and productivity. We illustrate that this is not the case and that the combination indeed can be very straightforward, requiring few exotic features or implementation strategies.

A known AOP problem addressed by several authors is the difficulty in teasing out and reusing aspects that are tightly integrated into host code [43]. The reason why aspects are often so tightly integrated with the host code is the lack of an encapsulating interface between the aspectual unit and the rest of the system. This paper addresses this issue by combining the power of aspects with the encapsulation power of modules. We demonstrate that writing aspects against formal participant graphs and attaching them to other participant graphs helps in making the aspects both more abstract and reusable.

We have presented ACs, which combine the static properties of modules, encapsulation, external composition and separate compilation, with the flexible programming power of aspects. ACs are a wrapper around Java, adding a module system and support for aspectual behavior and separate compilation. We have shown how the system implements the separate compilation of aspectual and additive behavior, allows the composition and parameterization of collaborations and can transparently interface with existing Java programs. We have elided the description of the implementation, which is implemented as a somewhat involved desugaring of aspectual features to a module language back-end.

Summarizing an external comparison, we find that there is a low overhead in using the encapsulation offered by ACs, when compared to AspectJ's aspects or Hyper/J's HyperSlices when these are programmed to promote reuse, while offering the considerable benefits of separate compilation and type safety.

We expected ACs by design, to be better than AspectJ at reuse, but underperform on a small-sized example, since the extra syntax of a module language may be comparatively cumbersome for a small program. Much to our delight, we found that compared with reusable aspects in AspectJ, ACs are only somewhat more verbose, but at the considerable benefit of separate compilation and type safety.

We expected ACs to offer modular power similar to Hyper/J, but with better aspectual features. This was borne out. We did find that Hyper/J brackets allow before and after advice to be written fairly easily, but simulating around advice was quite tricky, composing hyperslices with differing names was quite verbose and controlling the details of the generated output was awkward. Hyper/J's features work well when working with a set of hyperslices with common and recurring names, allowing its composition functions to be used with the mergeByName matching.

The ACs advantage shows up when composing very different collaborations, with differing names and participant graph shapes. Additionally, ACs will do even better when precise control is needed over which members are to be visible from a composed collaboration and when one collaboration is to be reused several times in different contexts, with the same advice applied to signatures of different types.

## REFERENCES

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

[2] Kiczales, G. (2001) Is cross-cutting a good way to define aspects? Email message archived at http://aspectj.org/pipermail/users/2001/000723.html.

[3] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W. G. (2001) An overview of AspectJ. In Knudsen, J. L. (ed.), *Proc. 15th Eur. Conf. on Object-Oriented Programming (ECOOP 2001)*, Budapest, Hungary, June 18–22. *Lecture Notes in Computer Science*, **2072**, 327–353. Springer.

[4] Tarr, P., Ossher, H., Harrison, W. and Sutton, S. M. (1999) N degrees of separation: multi-dimensional separation of concerns. In Garland, D. and Kramer, J. (eds.), *Proc. 21st Int. Conf. on Software Eng. (ICSE 1999)*, Los Angeles, CA, May 16–22, pp. 107–119. IEEE Computer Society.

[5] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1993) Design patterns: abstraction and reuse of object-oriented design. In Nierstrasz, O. M. (ed.), *Proc. 7th Eur. Conf. on Object-Oriented Programming (ECOOP'93)*, Kaiserslautern, Germany, July 26–30. *Lecture Notes in Computer Science*, **707**, 406–431. Springer.

[6] Ernst, E. (2000) Syntax based modularization: invasive or not? In Tarr, P., Bergmans, L., Griss, M. and Ossher, H. (eds.), *Workshop on Advanced Separation of Concerns (OOPSLA'00)*. Department of Computer Science, University of Twente, The Netherlands.

[7] Clarke, S. and Walker, R. J. (2001) *Separating Crosscutting Concerns Across the Lifecycle: From Composition Patterns to AspectJ and Hyper/J*. Technical Report TCD-CS-2001-15 and UBC-CS-TR-2001-05, Trinity College, Dublin and University of British Columbia.

[8] AspectJ Team. *AspectJ Programming Guide*. http://aspectj.org/doc/dist/progguide/apbs03.html.

[9] Igarashi, A., Pierce, B. and Wadler, P. (1999) Featherweight Java: a minimal core calculus for Java and GJ. In *Proc. 14th Ann. Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'99)*. *ACM SIGPLAN Notices*, **34**(10), October, Vancouver, BC, Canada, October 18–22, pp. 132–146.

[10] Meyer, B. (1988) *Object-Oriented Software Construction. Series in Computer Science*. Prentice-Hall.

[11] Mezini, M. (1997) Maintaining the consistency of class libraries during their evolution. In *Proc. 12th Ann. Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'97)*. *ACM SIGPLAN Notices*, **32**(10), pp. 1–21, Atlanta, GA, October 5–9.

[12] Katz, S. (1993) A superimposition control construct for distributed systems. *ACM Trans. on Programming Languages and Systems*, **15**(2), 337–356.

[13] Ernst, E. and Lorenz, D. H. (2001) *Aspectual Polymorphism*. Technical Report NU-CCS-01-09, College of Computer Science, Northeastern University, Boston, MA.

[14] Agesen, O., Freund, S. N. and Mitchell, J. C. (1997) Adding type parameterization to the Java language. In *Proc. 12th Ann. Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'97)*. *ACM SIGPLAN Notices*, **32**(10), pp. 49–65, October, Atlanta, GA, October 5–9 1997.

[15] Bracha, G., Odersky, M., Stoutamire, D. and Wadler, P. (1998) Making the future safe for the past: adding genericity to the Java programming language. In *Proc. 13th Ann. Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'98)*. *ACM SIGPLAN Notices*, **33**(10), pp. 183–200, Vancouver, BC, Canada, October 18-22.

[16] Lieberherr, K., Lorenz, D. H. and Wu, P. (2003) A case for statically executable advice: checking the law of Demeter with AspectJ. In *Proc. 2nd Int. Conf. on Aspect-Oriented Software Development (AOSD 2003)*, Boston, MA, March 17–21, pp. 40–49. ACM Press.

[17] Abadi, M. and Cardelli, L. (1996) *A Theory of Objects*. Springer, New York.

[18] Wegner, P. (1987) The object-oriented classification paradigm. In Shriver, B. and Wegner, P. (eds.) *Research Directions in Object-Oriented Programming*, pp. 479–560. MIT Press.

[19] Ovlinger, J. (2003) *Modular Programming with Aspectual Collaborations*. PhD Thesis, Northeastern University. Forthcoming. http://www.ccs.neu.edu/home/johan/research/.

[20] Lindholm, T. and Yellin, F. (1999) *The Java[tm] Virtual Machine Specification*. Addison-Wesley.

[21] McDirmid, S., Flatt, M. and Hsieh, W. C. (2001) Jiazzi: new age components for old-fashioned Java. In *Proc. 16th Ann. Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'01). ACM SIGPLAN Notices*, **36**(11), pp. 211–222, Tampa Bay, FL, October 14–18.

[22] Harrison, W. and Ossher, H. (1993) Subject-oriented programming (a critique of pure objects). In *Proc. 8th Ann. Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'93). ACM SIGPLAN Notices*, **28**(10), pp. 411–428, Washington, DC, September 26–October 1.

[23] Ossher, H., Kaplan, M., Harrison, W., Katz, A. and Kruskal, V. (1995) Subject-oriented composition rules. In *Proc. 10th Ann. Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95). ACM SIGPLAN Notices*, **30**(10), pp. 235–250, Austin, TX, October 15–19.

[24] Videira Lopes, C. and Kiczales, G. (1998) Recent developments in AspectJ. In Demeyer, S. and Bosch, J. (eds.), *Object-Oriented Technology. ECOOP'98 Workshop Reader*, *Lecture Notes in Computer Science*, **1543**, 398–401. Workshop Proceedings, Brussels, Belgium, July 20–24.

[25] Videira Lopes, C. I. (1997) *D: A Language Framework for Distributed Programming*. PhD Thesis, Northeastern University.

[26] Videira Lopes, C. and Lieberherr, K. J. (1994) Abstracting process-to-function relations in concurrent object-oriented applications. In Tokoro, M. and Pareschi, R. (eds.), *Proc. 8th Eur. Conf. on Object-Oriented Programming (ECOOP'94)*, *Lecture Notes in Computer Science*, **821**, 81–99, Bologna, Italy, July 4–8. Springer.

[27] Videira Lopes, C. (1995) Graph-based optimizations for parameter passing in remote invocations. In Cabrera, L.-F. and Theimer, M. (eds), *4th Int. Workshop on Object Orientation in Operating Systems*, Lund, Sweden, August, pp. 179–182. IEEE Computer Society Press.

[28] Hannemann, J. and Kiczales, G. (2002) Design pattern implementation in Java and AspectJ. In *Proc. 17th Ann. Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'02). ACM SIGPLAN Notices*, **37**(10), pp. 161–173, Seattle, WA, November 4–8.

[29] Flatt, M. and Felleisen, M. (1998) Units: cool modules for hot languages. In *ACM Conf. on Programming Language Design and Implementation*. Montreal, Canada, June 17–19, pp. 236–248. ACM Press.

[30] Clifton, C., Leavens, G. T., Chambers, C. and Millstein, T. (2000) MultiJava: modular open classes and symmetric multiple dispatch for Java. In *Proc. 15th Ann. Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'00). ACM SIGPLAN Notices*, **35**(10), pp. 130–145, Minneapolis, MA, October.

[31] Aldrich, J. and Chambers, C. (2002) Architectural reasoning in ArchJava. In Magnusson, B. (ed.), *Proc. 16th Eur. Conf. on Object-Oriented Programming (ECOOP 2002)*, *Lecture Notes in Computer Science*, **2374**, 334–367, Málaga, Spain, June 10–14. Springer.

[32] Aldrich, J., Chambers, C. and Notkin, D. (2002) ArchJava: connecting software architecture to implementation. In *Proc. 24th Int. Conf. on Software Engineering (ICSE 2002)*, Orlando, FL, May 19–25, pp. 187–197. ACM Press.

[33] Mezini, M. and Lieberherr, K. (1998) Adaptive plug-and-play components for evolutionary software development. In *Proc. 13th Ann. Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'98). ACM SIGPLAN Notices*, **33**(10), pp. 97–116, Vancouver, BC, Canada, October 18-22.

[34] Lieberherr, K., Lorenz, D. and Mezini, M. (1999) *Programming with Aspectual Components*. Technical Report NU-CCS-99-01, College of Computer Science, Northeastern University, Boston, MA.

[35] Holland, I. M. (1993) *The Design and Representation of Object-Oriented Components*. PhD Thesis, Northeastern University.

[36] Mezini, M. (1997) *Variation-Oriented Programming Beyond Classes and Inheritance*. PhD Thesis, University of Siegen.

[37] Herrmann, S. and Mezini, M. (2000) PIROL: a case study for multidimensional separation of concerns in software engineering environments. In *Proc. 15th Ann. Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'00). ACM SIGPLAN Notices*, **35**(10), pp. 188–207, Minneapolis, MA, October.

[38] Mezini, M. and Osterman, K. (2002) Integrating independent components with on-demand remodularization. In *Proc. 17th Ann. Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'02). ACM SIGPLAN Notices*, **37**(10), pp. 52–67, Seattle, WA, November 4–8.

[39] Ernst, E. (2001) Family polymorphism. In Knudsen, J. L. (ed.), *Proc. 15th Eur. Conf. on Object-Oriented Programming (ECOOP 2001)*, Budapest, Hungary, June 18–22. *Lecture Notes in Computer Science*, **2072**, 303–326. Springer.

[40] Clarke, S. and Walker, R. (2001) Composition patterns: an approach to designing reusable aspects. In *Proc. 23rd Int. Conf. on Software Eng. (ICSE 2001)*, Toronto, Canada, May 12–19, pp. 5–14. IEEE Computer Society.

[41] Booch, G., Rumbaugh, J. and Jacobson, I. (1999) *The Unified Modeling Language User Guide*, Object Technology Series. Addison-Wesley.

[42] Clarke, S. and Walker, R. (2002) Towards a standard design language for AOSD. In *Proc. 1st Int. Conf. on Aspect-Oriented Software Development (AOSD 2002)*, Enschede, The Netherlands, April, pp. 113–119. ACM Press.

[43] Ernst, E. (2000) Separation of concerns and then what? In C. Videira Lopes, Bergmans, L., D'Hondt, M. and Tarr, P. (eds), *Workshop on Aspects and Dimensions of Concern*, Cannes, France, June 11–12. ECOOP 2000. Department of Computer Science, University of Twente, The Netherlands.