Aspectual Collaborations and Modular Programming

Johan Ovlinger

Northeastern University and Skyva International
johan@ccs.neu.edu

Abstract. We present a rewriting approach for combining aspect-oriented programming and role-based collaborations to create reusable programming constructs called Aspectual Collaborations. These are separately compiled units of encapsulated behavior that can be adapted to fit a variety of uses.
Our approach is to combine sets of compiled Java `.class` files according to an adaptation specification. A lightweight (based on field and method modifier keywords only) source-to-source transformation lets us use any Java compiler for type checking and compilation to bytecodes, and an equally lightweight bytecode-to-bytecode transformation removes the need for auxiliary interface files. The Java type system is leveraged to protect the programmer from inadvertent type errors without additional constraints on how collaborations may be programmed.
Simple but powerful concepts allow us to provide separate compilation of aspects, composition, and flexible object code reuse with a natural and intuitive operational semantics.

1 Introduction

Object-oriented programming has delivered great wins in software architecture, but has failed to untangle the various concerns that necessarily overlap in any non-trivial application. While careful architectural design of applications can minimize this tangling, it is often infeasible for the various concerns to be completely separated in traditional programming environments.

In response to this problem, two related fields of study have quickly risen to prominence: Aspect-Oriented Programming and (Multi-Dimensional) Separation of Concerns. Exact definitions of the two are hotly debated. We hesitate to fan the flames, but to provide some definitions to base our discussion on we present our own definitions below. These are not meant to be definitive, and best taken with a healthy dose of salt and vigorous hand waving.

Both concerns and aspects are broadly concerned with localizing behaviors that are normally distributed throughout the application. Concerns focus on how to reuse such behaviors in different base programs [TOHS99,OKK+96,CHOT99], while aspects focus on how to express the attachment of the behavior and the base program [KLM+97].

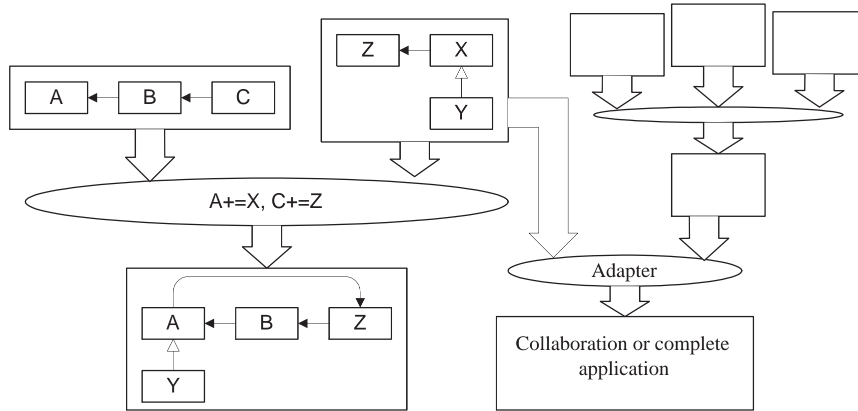

Fig. 1. Collaborations are combined by adapters to form other collaborations.

A Concern is typically a large multi-class behavior written over its own class model, which is mapped to the host class model at attachment time. A concern is typically attached only once to the application, with class-to-class mappings specifying exactly how it is to fit. An identifying example might be a file cache that needs to be updated from several methods in the application and invalidated by others.

Aspects are often written over the class model of the application or are small enough to not need a class model. An Aspect is often attached to the host program several times, often wrapping existing methods specified using globs[1] Recurring examples are logging, remote procedure dispatching, or multi-method synchronization.

1.1 This Paper

This paper is a detailed proposal for yet another aspect/concern programming system, which we call Aspectual Collaborations, or just Collaborations. Our proposal is novel in that it combines the ability to write concerns over separate class models (section 2.1) with the light-weight multiple attachments of aspects (section 2.2), allowing us to write generic aspectual behavior that can be adapted to host applications with differing class models (section 3).

Adapters produce new collaborations or complete applications by combining collaborations with other collaborations or base applications, as suggested by figure 1. Simple but powerful adaptation concepts combine to provide a very expressive programming system.

[1] A glob is a wildcard expression (commonly *) which can is used to match against a large set of names [Ray00].

```
package host;
class HasUsers {
  User[] users;
  void initUsers() { ... }
  void addUser(User u) { ... }
  void sortusers() { System.err.println("No sorting. Yet."); }
  User firstuser() {
    sortusers();
    return users[0];
  }
}
class User {
  String name;
  int uid;
  boolean lexbefore(User that) {
    return this.name.compareTo(that.name) > 0;
  }
  boolean uidbefore(User that) { return this.uid < that.uid; }
}
```

Fig. 2. A host application for a running example.

This paper suggests a novel implementation strategy that reuses any existing Java compiler for semantic analysis and type-safe[2] separate compilation of collaborations and the applications in which they are used . The suggested implementation leverages features of the Java `.class` file format to avoid external interface files for pre-compiled collaborations. For the type safety to persist across adaptation, our adapters we must observe a number of constraints (section 5).

We round out the presentation by detailing some unsolved issues we have run into during the elaboration of our design (section 6). Lastly, there is a review of related work (section 7).

2 Concepts and Terms

The underlying concept we work with is that of a **class model** or **class graph**. We use the terms interchangeably to mean classes and their *is-a* and *has-a* relationships. Users of UML class diagrams [FS97] will be familiar with the concept. A sample host application is shown in figure 2 [3], to which we will add sorting behavior.

[2] By type-safe, we mean that we avoid having to use unsafe casts to make things compile. We do use casts in our implementation, but not in the code written or compiled by the user.

[3] We elide some methods for brevity.

We define program components called **Collaborations** to express encapsulated behaviors that may flexibly be combined with host applications at a later date.

A collaboration is a closed class graph where we call each class a **Role**. It is closed in that all the roles of the collaboration are known at compile time and cannot be added later, unlike a package to which classes can be added at any time. It defines its own class graph with object-oriented behavior, independent the class graph of any applications it is to be used with. Additionally, we restrict each collaboration to consist of one package, but this is a merely simplifying assumption. Since a collaboration is a generalization of a package, we will no longer refer to packages, with the understanding that a package is equivalent to collaboration with no missing behavior.

Roles differ from normal classes in that they may be missing methods or fields. We call these missing parts **expected**, as we expect them to be provided later on. Similarly to collaborations and packages, we will speak of roles in general, with a class being a special case of a role in a collaboration with no expected parts.

Collaborations are our units of compilation, which is achieved by filling in the expected parts of roles with dummy implementations[4], allowing them to be compiled separately of their uses. The dummy definitions are discarded as real parts are provided by adapters.

In order to use a collaboration's behavior in an application, we need to provide its expected parts and reconcile the differences between its and the application's class models. This is performed by an **adapter** specification, which is interpreted by an aspectual compiler to generate a new collaboration.

The adapter generates a collaboration by combining roles from a number of compiled input collaborations. By default, parts are not shared between collaborations thus combined, unless explicitly exported. Renaming[5] is used both to hide parts from other collaborations and to export them under different names.

2.1 Collaborations

Collaborations are our unit of reuse, encapsulation, and compilation. Our discussion will assumes that they are written in Java, but the concepts apply to most object-oriented languages.

The `sort` collaboration in figure 3 implements an insertion sort on an array of items that know how to compare themselves. Additionally, each item is augmented with a variable that tracks how many times it has been moved, which is used to report the geometric mean number of moves performed during the sort.

The collaboration is compiled in three steps: Collaborations are written using Java augmented with four keywords (**role**, **collaboration**, **expected**, and

[4] For Unit testing purposes, we might want to supply our own testing code which is discarded when the collaboration is adapted.

[5] Because we assume collaborations to be closed, we can systematically find and rename all occurrences of an identifier, effectively hiding it from outside reference.

```
collaboration sort;
role ArrayHolder {
  expected Item[] arr;
  void insertionsort() {
    for (int i=0; i<arr.length; i++) for (int j=i; j<arr.length; j++)
        if (arr[j].before(arr[i])) swap(i,j);
    int m=0;
    for (int i=0; i<arr.length; i++) m += arr[i].moved*arr[i].moved;
    System.out.println("geom. average number of moves: "+(m/arr.length));
  }
  private void swap(int i, int j) {
    arr[i].moved++;  arr[j].moved++;
    Item o = arr[i]; arr[i]=arr[j]; arr[j]=o;
  }
}
role Item {
  expected boolean before(Item other);
  int moved = 0;
}
```

Fig. 3. A collaboration that sorts an array of Items.

aspectual), all of which occur outside method bodies. A simple transliteration suffices to transform the collaboration to plain Java for compilation to `.class` files. Lastly the keywords are used to add annotations to the `.class` files.

Instead of classes, collaborations contain roles; our example `sort` contains ArrayHolder and Item. Roles contain full-fledged object-oriented behavior (`swap`, `moved`, and `insertionsort`), and may take full advantage of interfaces, subclassing, and external library classes[6]. In addition, each role may be incomplete, expecting fields or methods to be filled before it is used. Examples are the array `arr` and the method `before`.

Expected parts are similar to **abstract** parts, but differ in that an abstract class will always be abstract, so can never be instantiated. In contrast, expected parts will be provided before the collaboration is used, so the restriction against instantiation does not apply to roles.

The reuseability and power of collaborations stem largely from expected parts; they are the hooks that allow the collaboration to be both generic (with late bound behavior) and fine grained (with direct access to necessary parts in the host role). Additionally, encapsulation and subsequent mapping of the collaboration's class model allows the collaboration to be programmed against a convenient data model that will be stretched to fit over that of the application. Thus the collaboration can add state and behavior to the roles of the host application. Because the coupling is loose, they are easy to attach to a wide variety

[6] The Collaboration is closed only in membership, not references, as evidenced by the call to `System.out.println`.

```
adapter sortedhost;
combine {sort,host} {
  host.HasUsers += {sort.ArrayHolder} {
    sort::Item[] arr provided-by host::User[] users;
    host::void sortusers() provided-by sort::void insertionsort();
  }
  host.User += {sort.Item} {
    sort::boolean before(Item) provided-by host::boolean uidbefore(User);
  }
}
```

Fig. 4. An adapter and host application for `sort`.

of situations, and because the interface is fine grained, it is possible to write very light weight reusable behavior that would be impractical if a more granular interface were used.

2.2 Adapters

Collaborations define generic behavior, but need to have missing behavior provided before they can be used. Additionally, the different class models need to be resolved to one coherent class model. Finally, the resulting collaboration needs to be given a name.

These tasks boil down to copying while renaming methods, fields, classes and packages. This is exactly the operational meaning an adapter has; it creates a new collaboration or complete application by copying the contents of the participant collaborations into a new collaboration, hiding unexported parts by renaming them to unpronouncable names and renaming exported parts according to the clauses of the adapter.

The adapter in figure 4 creates a new package `sortedhost` by combining the old application `host` and the sorting behavior `sort`. New roles need to be named. In this example, we choose to use HasUsers as a template[7], keeping the unqualified class name and exporting all of its visible parts. Then the contents of ArrayHolder are copied into the new HasUsers, also under renaming.

Exported and Expected Parts

Expected parts that have been provided in the adapter will be renamed to the providing part. In our example, references to `arr` will be replaced by references to `users`. The dummy body used to compile the role will be discarded; an example of this is the error message in the original `sortusers` body. A small enhancement would allow us to leverage the replacement of dummy bodies to provide unit

[7] `role += oldrole` indicates that `role` is to be used as a template with `oldrole` inserted. The alternative of creating a role from scratch – using the syntax `role = {oldrole1, oldrole2}` – requires that we use explicit exports.

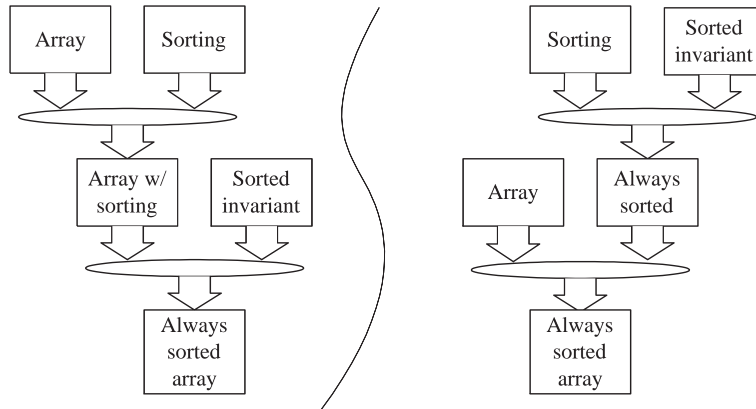

Fig. 5. Composition allows us to reuse adapters as well as collaborations

testablility to collaborations: if we were able to provide our own dummy bodies, these could be used to test the collaboration in isolation. Just like the original dummy parts, these would be automatically discarded when the real definitions are provided.

Similarly, exported parts will not be unguessably renamed, but instead renamed to whatever names they are exported under.

Expected and exported parts both induce the possibility of name clashes as they are not automatically renamed. In the case of a clash, we report an error and require the adapter to be rewritten to rename the part to something else. For example, had we attempted to export `moved` as a variable named `uid`, an error would have been reported.

Encapsulation

The renaming described above does more than avoid name clashes. Because unexported parts are renamed to unguessable and unpronouncable Java identifiers, outside collaborations are unable to reference them. Renaming thus encapsulates each collaboration so that only exported parts are visible after adaptation. If so desired, there are mechanisms available in class files to hide these parts from Java's introspection features as well, although that has not been investigated further.

Any unexported expected parts become unprovidable as we are unable to name them in subsequent adapters. As a collaboration which can never be completed seems like a bad idea, we make it a static error to fail to export or provide an expected part. Other restrictions are detailed in section 5.

2.3 Composition

Whether adapters are reusable or not is highly dependent on the order in which components are combined. Figure 5 illustrates a hypothetical case: the left side

of the figure illustrates first adding sorting to an array and then a collaboration to maintain the invariant that the array is always sorted. That sequence does not allow us to reuse any of the adapters, since they all depend on the details of use (in this case the host). If we wanted to perform the same attachment to a vector instead of array, we would have to rewrite all adapters in the system.

It would be better to proceed according to the right side of the figure, in which sorting and invariant maintenance are first composed, and then added to the host. By first composing them, we are able to reuse the composed invariant array as a unit. Attaching it to a vector rather than array would only require that we rewrite the last adapter in the system.

The composition on the right side of the figure is possible because adapters are symmetric in general, fusing together several input collaborations to form one output collaboration. In general, we need to specify exactly what the interface of the new classes will be, determining which parts are to be exported, and which to be hidden. However, because we assume that the asymmetric case will be common, we provide a shorthand notation (`+=`) for exporting all the parts from one collaboration implicitly and only explicitly from the other[8].

2.4 Relation to Module Calculi

Composition via adaptation seems at first glance to be very much like importing modules in a flexible module calculus such as Units [FF98], where smaller modules are linked together to form larger modules, and required interfaces for the inner modules can be exported into the required interface of the encapsulating module.

Modules and collaborations differ mainly in their runtime representation. Because of our implementation strategy of code duplication at adaptation time, there is no one runtime instance representing a collaboration, while modules are implicitly shared, all clients referencing the same instance. Multiple interfaces exported by a module's implementation all reference this shared instance, as do multiple clients of these interfaces.

Sharing encapsulated state between multiple clients is natural to express using modules, but is cumbersome with collaborations. Conversely, since collaborations are implemented by code insertion, it is natural to maintain per-client state (by keeping it directly on the client), while modules require some external mechanism.

2.5 Implementation Overview

One of the attractive features of our approach is that it provides separate compilation of collaborations and aspects very cheaply. In this section we outline the proposed implementation, describing what steps are needed to compile a set of collaborations and adapters to executable code.

[8] The notation plays on its widespread use to increment variables.

Collaborations look like sets of Java source files, with the differences that they use **collaboration** and **role** inplace of **package** and **class**, respectively. In addition, parts may be annotated with the keywords **expected** and **aspectual**. A preprocessor transliterates the input files to plain Java, stripping the keywords but remembering which parts were thus annotated. The preprocessor fills in dummy bodies for expected methods and generates classes according to the aspectual methods' signatures as described in section 3.2.

After the generated Java has been compiled, the postprocessor annotates `.class` files with the expected and aspectual keywords, leveraging the fact that attributes can be attached harmlessly to fields and methods entries in a `.class` file. The post processor also renames the classes generated for the aspectual methods so that their names cannot occur in Java source files.

The adapter combines several collaborations by copying all the entries from corresponding roles into single class files. A renaming environment derived from the adapter allow us to process each set of corresponding roles separately, minimizing the amount of memory needed by the process. Most operations carried out by the adapter are covered by this renaming environment; export, encapsulation and providing expected methods are all different renaming operations. Generic aspectual methods do however require some code to be generated into the target role, as described in section 3.2. This code is very straightforward, and is fully described by the signatures of the aspectual and wrapped methods.

3 Aspects

While we have advertised aspects both in the title and abstract of the paper, we have yet to back up our claims.

We define an aspect operationally as a collaboration whose behavior is invoked implicitly by the host application, rather than explicitly[9]. This is in line with the opinion that aspects are behaviors of which the rest of the program are ignorant [FF00]. Both the host application and aspectual collaboration are mutually ignorant; only the adapter is aware that and how they are linked.

The adapter is the natural location to define aspectual connection; it is the only component of our system that is aware of connection between components, and since it operationally weaves a new collaboration as a result, it natural to take the opportunity to insert aspectual behavior at that point.

We speak of an aspectual method **wrapping** a host method. By this we mean that the aspectual method will be invoked in the place of the host method, and may control how the host method is invoked. This is a more limited form of aspect than AspectJ provides, which is due partly to our unwillingness to modify the bytecodes of individual methods and our desire for type safe separate compilation of aspect and host.

[9] Since aspectual behavior is invoked via a method but can then call other methods in the collaboration, we use the terms Aspect, Aspectual Method, and Aspectual Collaboration interchangeably.

```
collaboration nodups;
role ElemHolder {
  expected Elem[] elems;
  expected void addElem(Elem e);
  void insertMaybe(Elem e) {
    for (int i=0; i<elems.length; i++)
      if (elems[i].equals(e)) return;
    addElem(e);
  }
}
role Elem {
  expected int id;
  public boolean equals(Object o) { return ((Elem)o).id == id; }
}

adapter sortednoduphost;
combine {nodups,sortedhost} {
  sortedhost.HasUsers += {nudups.ElemHolder} {
    nodups::Elem[] elems provided-by sortedhost::User[] users;
    sortedhost::void addUser(User)
        replaced-by nodups::void insertMaybe(Elem)
        provides nodups::void addElem(Elem);
  }
  sortedhost.User += {nodups.Elem} {
    export nodups::boolean equals(Object);
    nodups::int id provided-by sortedhost::int uid;
  }
}
```

Fig. 6. Adding checks against duplicates.

It turns out that there are two forms of aspectual behavior that we are able to express: precise and generic aspects. Precise aspects are more limited in how they may be used, but require only a very small addition to the system as described; namely the ability to replace an existing method while retaining the existing body under another name. Generic aspects are much more flexible in how they can be attached to the host program but conversely more limited in what they can express and require a bit more system support.

While we don't describe it here, adapters could very well be extended with globbing behavior commonly associated with aspects. In the next two sections we describe how these two forms of aspectual methods are implemented.

3.1 Precise Aspects

Precise aspectual methods are restricted in that they must exactly match the signature of the method they are wrapping, down to declaring the same excep-

tions. The exact match allows them access to the arguments of the host method, being able to call methods or swap the order of arguments.

An instance where this makes sense is memoizing an expensive method. The logic of when to cache a result for future use and when not to is a perfect example of a concern; it is orthogonal to what the method computes, as long as we are able to make a determination when two sets of arguments would have produced the same results.

These aspectual methods are implemented by the aspectual collaboration expecting the host method under one name and exporting the aspectual method under another. The adapter renames the host method to the new name and replaces it with the aspectual method. Because the signatures by definition match precisely (after adaptation, of course), we are guaranteed that it is safe to replace the host method's body with the aspectual method. The reference to the former host method in the aspectual method is guaranteed to be safe by the same argument.

As an example, we return to the sorted array of Users. We want to avoid duplicating users in the array, only adding them if they don't already exist. The aspectual collaboration in figure 6 implements this behavior. The former application method `addUser` is referenced in the collaboration as `addElem`[10], and the aspectual method `insertMaybe` is inserted as the new `addUser`. Additionally, the adapter adds an `equals` method to the User class.

Thus, when `addUser` is invoked by application code, it is actually the body of `insertMaybe` which is invoked and in turn calls the original – now `addElem` – if the element doesn't already exist.

The main issue with this form of aspectual behavior is that it is really quite limited in its applicability; the signature matching must be precise, even disallowing subtyping. This is both due to resolution rules for overloaded methods and to ensure type safety. For example, an aspectual method expecting an Object argument could not be used to wrap a method that expected a Vector, even if it did nothing but pass argument to the wrapped method.

This approach is also useful as it leaves the original `addUser` method visible to the rest of the collaboration outside `invokeMaybe`, from where it could be re-exported, if so desired.

3.2 Generic Aspects

While being able to affect arguments and results conveys a lot of power to the programmer, it is very restrictive in reuse. In order to gain reuse with generic aspectual methods, we need to restrict the expressiveness of the methods, and also introduce additional logic into our adapter compiler.

The difficulty here is that we are ignorant of the signature of the host method to be wrapped by the aspectual method. Hence we are unable to directly reference the host method or its arguments, in our aspectual method, making it difficult to invoke the host method. Had we been working at the source code

[10] This, too, is renamed to avoid polluting the name-space of the application class.

```
collaboration keepsorted;
role Thing {
  expected void sort();
  aspectual ReturnVal sortafter(ExpectedMethod e) {
    ReturnVal r = e.invoke();
    sort();
    return r;
  }
}

adapter keepsortedhost;
combine {keepsorted,sortedhost} {
  sortedhost.HasUsers += {keepsorted.Thing} {
    keepsorted::void sort() provided-by sortedhost::sortusers();
    keepsorted::sortafter wraps sortedhost::void initUsers();
  }
} and {
  sortedhost.HasUsers += {keepsorted.Thing} {
    keepsorted::void sort() provided-by sortedhost::sortusers();
    keepsorted::sortafter wraps sortedhost::void addUser(User);
  }
}
```

Fig. 7. Sorting the list.

level, we could have implemented our aspectual method as a template to be specialized depending on the details of the host method. A similar approach could be worked with bytecodes as well, but as soon as we start modifying bytecodes, we must be very careful so as not to break the static type safety we gain by separate compilation.

Our approach is rather to reify the wrapped method call (method and arguments) into a thunk: an object with a single method which invokes the wrapped method. Abstract classes allow us to reify expected methods and their return values in such a way that the aspectual collaboration is separately compilable. The adapter will generate code to create and unwrap these objects depending on the signature of the wrapped method.

As the adapter will need to generate wrapper code for aspectual methods, we require them to be written with a specific signature:

aspectual ReturnVal `methodname(` ExpectedMethod `exp)` { }

The **aspectual** keyword indicates to the preprocessor to process the type names ReturnVal and ExpectedMethod in a special manner, and also that the compiled method should me marked with the bytecode attribute `Aspect`, which causes it to be treated specially by the adaptive compiler. The ReturnVal and ExpectedMethod are not hardwired class names; rather, whatever the program-

```
package keepsorted;
class Thing {
  void sort() { /*expected*/  return; }
  ReturnVal$A sortafter(ExpectedMethod$A e) {
    /*aspectual*/
    ReturnVal$A r = e.invoke();
    sort();
    return r;
  }
}
abstract class ReturnVal$A { }
abstract class ExpectedMethod$A {
  abstract ReturnVal$A invoke();
}
```

Fig. 8. Compiled Java Code from `keepsorted`.

mer writes in those positions will be the names of auxiliary roles generated by the preprocessor.

Adapters need to treat `Aspect` methods differently from others, generating wrapper code to create a closure object (a subtype of ExpectedMethod) which is able to invoke the original method with supplied arguments. This object is passed to the aspectual method as its only argument. An object of a subclass of ReturnVal reifies any result from the invocation of the wrapped method. It is returned from the aspectual method to be unwrapped by the generated wrapper code, allowing any results to be returned to the caller of the wrapped method.

Figure 7 shows an aspect and its attachment to our sortable users host. We would really like to maintain the list in sorted order so that we don't need to sort before each time we want the first element. To this end, we write an aspectual method that sorts the list after certain methods, namely `initUsers` and `addUser`.

Notice that `sort` is mapped twice in the figure; once per attachment (strung together with the **and** keyword). It would be perfectly feasible for it to be mapped differently in each case, so we require that each attachment be complete. This is true not only for aspects, but for collaborations in general.

Operational Details of Generic Aspects

To compiling `keepsorted`, the preprocessor first generates source corresponding to the code in figure 8. Two abstract classes are generated corresponding to the signature of each aspectual method. These classes become part of the collaboration, and will be subclassed by classes generated by the adapter from the signature of any wrapped methods. After compilation, the bytecode postprocessor adds the necessary annotation attributes (hinted at by comments) to the `.class` files.

```
package keepsortedhost;
class HasUsers {
  ...
  void sortusers() { ... };
  void addUser(User e) {
    ExpectedMethod$A ex = new ExpectedMethod$A$addUser(this,e);
    ReturnVal$A$addUser r =
      (ReturnVal$A$addUser) sortafter$addUser(ex);
    return;
  }
  ReturnVal$A sortafter$addUser(ExpectedMethod$A e) {
    /*aspectual*/
    ReturnVal$A r = e.invoke();
    sortusers();
    return r;
  }
  void addUser$A(User e) { ... /*old method body, untouched*/ }
}
class ReturnVal$A$addUser extends ReturnVal$A { /*void result*/ }
class ExpectedMethod$A$addUser extends ExpectedMethod$A {
  User arg1;
  HasUsers self;
  ExpectedMethod$A$addUser(HasUsers s, User u) { self=s; arg1=u; }
  ReturnVal$A invoke() {
    ReturnVal$A$addUser r = new ReturnVal$A$addUser();
    self.addUser$A(arg1);
    return r;
  }
}
```

Fig. 9. Compiled Java Code from `keepsorted`.

After processing `keepsortedhost`, the system generates classfiles according to figure 9. Only the code pertaining to the `addUser` method is shown; `initUsers` is similar and the others have been omitted for brevity.

The first thing to notice are the `$A` and `$AaddUser` suffixed to methods and classes. The `$` is a legal identifier in `.class` files, but not in Java. We use it here to both hinder the programmer from accessing these identifiers and to indicate that these examples are Java level representations of what happens at the bytecode level. The reasons for this modification is explained below, in section 3.2.

Note also that none of the original method bodies have been modified and that the generated classes are very simple, consisting of a holder for the return value and a method to call one application method with a given set of arguments. In particular, what needs to be generated is dependent only on the signature of the wrapped method.

The API provided by the expected method and return objects is very limiting; currently consisting only of the ability to invoke expected methods to generate a

result. However, there are a number of extra methods that can be envisioned. It would be a small matter to add the ability to query the expected method for the name of the method it is wrapping, the signature, or even an array of arguments. More useful might be the ability to return a null result – `null` for reference types, or some arbitrary value for primitives – should the aspectual method choose not to call the wrapped method. Likewise, the return object could be queried for whether the call succeeded or threw an exception. While these abilities would be useful time constraints have hindered us from pursuing them further.

Aspectual Types

We have stated that the ReturnVal$A class is generated in response to the **aspectual** modifier on the role. This section aims to explain the reasons for mangling the names instead of just using ReturnVal and ExpectedMethod supplied by the programmer.

It is obvious that each attachment needs to generate new subclasses of ReturnVal$A and ExpectedMethod$A, as the method being wrapped and the types of the arguments and result being returned will be different from attachment to attachment[11]. In order to extract the value, exception, or void returned from a wrapped method we must downcast the ReturnVal$A object to its exact type. This type is known a-priori, as both the code to create and to unwrap these objects is generated by our aspectual compiler. We must make sure that the programmer actually returns a ReturnVal$A object that corresponds to this particular wrapped method or the downcast will result in a runtime casting error.

There are two reasons why casting errors occur at this stage are particularly undesirable:

1. The collaboration compiled. We want our typing to compose; if the individual pieces compile, and the adapter is legal, then no new casting errors should be introduced. We need to design a system that is flexible enough to write interesting code, yet strict enough so that if the collaboration compiles it can only break on purpose, not inadvertently[12].
2. Our code is invisible to the programmer. The stack trace generated from a casting error in our wrapper code will be completely incomprehensible to the programmer, as it refers to code they have no access to. Thus, we have to ensure that any errors that occur will happen in the programmer's code, not ours.

Let us illustrate with a motivating example, shown in figure 10. The collaboration does unexpected – but legal – things with the return value, storing it between invocations and returning the previous value if it exists.

[11] We currently generate one subclass per attachment. We suspect that reflection could be used instead of generating classes, but we have not explored this further.

[12] The Java type system in general, and casts in particular, make it impossible for us to ensure that no casting errors will occur. However, we can ensure that they cannot be caused without the programmer first having inserted a cast of that object.

```
collaboration fail;
role Foo {
  RV old;
  aspectual RV meth(EM e) {
    ReturnVal r = e.invoke();
    ReturnVal o = old;
    old = r;
    return (o!=null):o?r;
  }
}

package host;
class A {
  A get_a() { return new A(); }
}
class B {
  void print() { System.err.println("hi"); }
}

adapter whatever;
combine {fail,host} {
  host.A += {fail.Foo} {
   export fail::RV old
   fail::meth wraps host::A get_a();
  }
} and {
  host.B += {fail.Foo} {
   export fail::RV old
   fail::meth wraps host::void print();
  }
}
```

Fig. 10. Why some types cannot be exported.

We attach this collaboration both to `get_a` and `print`, and export the `old` variable. Were we to use the declared types – as opposed to the modified ones with `$` suffixed – there would be nothing to stop us from writing `anA.old = aB.old`. It would pass the typing rules, as both `old` variables would be of type RV.

However, the objects they hold will be different subclasses (as they wrap different signatures) so when our generated code for `get_a` tries to downcast the returned RV the cast will fail (as it wraps a void return, not an A).

Our fix is to make RV and EM existentially quantified[13] . We simulate this by replacing all occurrences in the collaboration with renamed versions. Thus, all intra collaboration references to RV and EM remain unbroken, but since the

[13] An existential type is one which is temporarily named within a quantification scope, but is unexportable, because the exact type is unknown.

name is now changed, the attempt to export it will fail. Encapsulation guarantees that the only way to share information between collaboration attachments is via exported or expected parts. Because we are unable to pronounce the mangled type names, we are unable to export any part with that type, effectively giving them existential quantification.

The **aspectual** attribute in role definitions and `.class` files signals that the methods signature should be or has been mangled, respectively. Postprocessing a collaboration after compilation, the system will simply append some unpronouncable string to the types, while processing an adapter the system will use those types as base classes for the generated attachment classes.

This approach allows the programmer to write code in the collaboration without stylistic constraints. However, it does come at the cost of a bit of expressive freedom. If we need to share a RV between collaborations – for example to store into an external Vector – the programmer is free to cast to and from Object. The aspectual method's signature will require programmer to perform a downcast before returning it. It is still possible for casting errors to occur in generated code, but in all such cases the programmer will have needed to perform a cast as well, which will catch some errors, and at least signal that they are aware of what is going on behind the scenes.

Exceptions

Instead of return values, a wrapped method may throw an exception. It is a simple matter to catch this exception and to store it, instead of a return value in the ReturnVal object that `invoke` returns. The wrapper code can then re-throw the exception or return the return value to the caller.

4 Collaboration Scopes

In earlier sections we have seen how collaborations and adapters can be used to write simple aspectual and generic behavior. However, the observant reader will have noticed that we have been silent on scoping issues, allowing the reader to draw conclusions from context.

Java provides us with three scopes: local, instance, and static. Local variables are visible only inside methods, instance variables are local to an object, and static are shared between objects of a certain class.

Instead of instances, we think about how the collaboration is attached. A collaboration may be attached to a class several times; this is most common for aspectual methods, but holds for collaborations in general. We need to differentiate between variables that are private to one attachment, and those which are shared between attachments.

The new scopes interact with old scopes in the following ways:

- **Per-Attachment.** Each attachment has attachment local parts. This is the default scope. These may be static or instance parts, but since they are alphaconverted at insertion time, they will be private to the attachment unless exported.

```
collaboration var;
role CountHolder {
  int movedshared=0;
}

adapter varhost;
combine {host, var} {
  host.HasUsers += {} {}
  host.User += {var.CountHolder} {
    export var::int movedshared as mvar;
  }
}

adapter sortedhost;
combine {varhost, sort} {
  varhost.HasUsers += {sort.ArrayHolder} {
    sort::Item[] arr provided-by varhost::User[] users;
    varhost::void insertionsort() provided-by sort:void insertionsort();
  }
  varhost.User += {sort.Item} {
    sort::boolean before(Item) provided-by varhost::boolean uidbefore(User);
    sort::int moved provided-by varhost::int mvar;
 }
} and {
  varhost.HasUsers += sort.ArrayHolder {
    sort::Item[] arr provided-by varhost::User[] users;
    export sort::void insertionsort() as void othersortusers();
  }
  varhost.User += {sort.Item} {
    sort::boolean before(Item) provided-by varhost::boolean lexbefore(User);
    sort::int moved provided-by varhost::int mvar;
 }
}
```

Fig. 11. Adding a variable to a class, followed by sorting.

- **Per-Host.** Multiple attachments to the same host class can share parts. Instance parts will be on a per-object basis, static parts will be shared among all attachments to that class. This is implemented by using the expected keyword; if we provide the same host variable to all attachments to satisfy an expected variable, then it will necessarily be visible to all attachments.
- **Per-Application** Sharing at the global level is not supported directly by our adapter language. However, this can be simulated by static variables on a special class, analogously to how static variables can simulate global variables in normal object-oriented idiom.

As an example of scoping, recall the sorting collaboration in figure 3 which keeps a count on each item of how many times it has been moved. The adapter

```
adapter sortedhost;
combine {sort, host, shared var} {
  host.HasUsers += sort.ArrayHolder {
    sort::Item[] arr provided-by host::User[] users;
    host::void sortusers() provided-by sort::void insertionsort();
  }
  host.User += {sort.Item,var.CountHolder} {
    sort::boolean before(Item) provided-by host::boolean uidbefore(User);
    sort::int moved provided-by var::int movedshared;
  }
} and {
  host.HasUsers += sort.ArrayHolder {
    sort::Item[] arr provided-by host::User[] users;
    export sort::void insertionsort() as void othersortusers();
  }
  host.User += {sort.Item,var.CountHolder} {
    sort::boolean before(Item) provided-by host::boolean lexbefore(User);
    sort::int moved provided-by var::int movedshared;
 }
}
```

Fig. 12. Adding a variable and sorting at once.

```
adapter composesort;
combine {sort, shared var} {
  export sort.ArrayHolder
  Item = {sort.Item, var.CountHolder} {
    sort::int moved provided-by var::int movedshared;
    export sort::expected boolean before(Item);
  }
}
```

Fig. 13. Composing variable and sorting collaborations.

in figure 4 attaches it to our host application, using one of the two available before methods. Imagine that we want to attach the sorting collaboration again, but with the other before method. Do we want to have the number of moves for an Item to be specific to an attachment, or shared between the two of them? The former would be per-attachment, while the latter per-host.

As written, the sorting collaboration has the `moved` variable scoped on a per-attachment basis. In order to scope it per-host, we must take advantage of the scoping of expected parts. To count moves for both sort methods together, we need to add an `int` field to class User in figure 2, make the `moved` variable expected, and then provide it to the collaboration.

We don't want to recompile our host, so we write a very simple collaboration to add just the variable to the host. We are then able attach sorting, as shown

in figure 11. However this is verbose, overly operational, and necessitates that `mvar` be exported into the public namespace.

It would be much more elegant to do both attachments in the same adapter (as in figure 12), or even better, to compose the two collaborations to automatically attach the variable before attaching sorting (as in figure 13). The complication is that while sorting is to be attached number of times, we want attachment the variable once, and share it between the other attachments.

4.1 Shared collaborations

To implement composed collaborations with one subcollaboration attachment shared between multiple attachments of the other sub-collaborations, we need an additional adapter keyword.

The **shared** keyword in figure 12 indicates that while `sort` be duplicated for each **and** clause, `var` be attached only once. The `+=` notation implies that `host` not be duplicated either, but had we used the more general merging notation, we would have had to be careful about which method of the host would be shared.

Sharing annotations are added to the generated collaboration along with information about the sharing scope. This allows the composed collaboration to be attached without knowledge that it contains a shared component. Figure 13 is an adapter that does exactly that; it produces a collaboration with exactly the same interface as the original sort, but one that shares its `moved` variable between attachments by means of the shared `var` collaboration.

5 Constraints on Adapters

Because we compose an application from individually compiled and checked parts, the correctness of the finished applications relies on the composition not to break typing. We need to constrain what an adapter can express so that no type errors are introduced. Unfortunately, we do not yet have a proof that our constraints are sufficient to allow typing to compose. We discuss one approach in our section on future work 6.

Wellformed. The most obvious requirement is that the adapter be wellformed. We require all parts and classes references to exist and have exactly the expected signature. Exact matching is required both by the design of the JVM, and also to remain type safe (otherwise we would require downcasts to be inserted, which might lead to runtime failure). If a method such as a getter is missing, it may be conveniently defined in a separate collaboration and inserted at the same time adapter.

Closed. Additionally, we require that only roles from the composed collaborations be attached in the adapter. We need to assume that we can find all occurrences of an identifier in order to justify systematic renaming. This rule is overly strong, but it catches many small complex errors with one easy rule.

Concrete and Complete. an Abstract role can cause problems if it is combined with a concrete role, producing a necessarily abstract role. The formerly

concrete role may have been instantiated in the other collaboration, which would cause bytecode verification to fail. Furthermore, we require that all abstract and unprovided expected parts be exported, as failure to do this would imply that the role effectively never can be instantiated or completely provided.

External Inheritance. Some care must be taken when roles subclass or implement external classes or interfaces. The concrete nature of the role, as well as the inherited behavior, may depend on methods in the role overriding inherited ones. Thus, any methods that override externally visible ones (either exported or inherited from outside the collaboration) must be exported without name change. This restriction is on the output role, so that any name clashes incurred if multiple input roles inherit the same interface can be resolved in the adapter.

Name clashes. We require that no names clash result from the adaptation. This is only an issue with exported methods and class constructors. Constructors and static initilizers of classes are compiled to methods with names that are known and have special meaning to the Java Virtual Machine. Hence they cannot be renamed, and we must create a new set of initializer methods from the old ones, while respecting the requirements on these methods. Because of how Java is compiled, we require that all roles have a no-argument constructor.

Shared Collaborations. Because they are not duplicated in the same way as other collaborations, collaborations imported as shared may not have parts exported. For the same reasons, any shared expected parts can only be provided by other shared parts.

Circularity. In principle there is nothing hindering the ambitious programmer from circular imports, but because generated `.class` files grow monotonically, we think this would be a bad idea. Additionally, we forbid circular expected-provided-by relationships, as they result in ill-formed class files.

Compatible class models. Lastly, for the resulting collaboration to wellformed, we require that the role models be compatible with the how the adapter puts them together.

Standard class models have two kinds of relations between classes: **has-a** and **is-a**, corresponding to variables and inheritance respectively. There are a number of models involved in the adaptation process; the output collaboration model and the n input collaborations.

The adapter sets up relations between each output role and the corresponding input roles. We need to verify that this mapping doesn't invalidate the individual class models' relations.

For instance if there if the adapter composes roles X and S and roles Y and T. If X is a subclass of Y, then it must be the case that S is a subclass of T.

Checking of has-a relationships is easier, and follows from wellformedness.

Note that the problem of compatible class models is different from the more difficult problem of compatible object models discussed in the section on future work (section 6).

6 Future Work

We have run into several issues in the elaboration of the system design, and present the two most interesting below. These are highlighted at the expense of other design considerations because they are intrinsic to the system as a whole while the other issues we have noted in passing are local to one or other facet of the system.

6.1 Object Graph Constraints

Each collaboration in an application is programmed to its own class model. The adaptation process combines these separate models to create the resultant collaboration's class model.

In our adapter, we specify pointwise equivalences between the various model's roles, which suffices to verify that the type checking done for each input collaboration will be valid for the output collaboration.

However, a class model is only an approximation of the runtime object model of each collaboration. Since these object models are implicitly fused along with the class models, we want to statically verify that they are compatible.

One situation where this situation becomes important is cache objects; if the path from class X to class Y is long, instances of X may choose to cache Y locally. The problem is that the logic to maintain the invariant that `anX.z.y == anX.cachedY` is encoded in the methods of the collaboration rather than explicitly stated.

If another collaboration has roles mapped to Z and Y, we run the risk of cache incoherence if this other collaboration were to modify the relation directly instead of using the observed setter.

In general, each collaboration's object model will have a set of implicit constraints in addition to typing information. If we were able to derive these constraints from code inspection or explicit annotation by the programmer, we would be able to verify their compatibility.

As a work around, we can note (or hope) that such constraints are relatively rare, so that clear documentation of the invariant can go a long way towards assuaging the problem. Since the programmer of the collaboration needs to be aware of the constraints anyway, such documentation should not prove too burdensome. Good programming practices such as using getters and setters allows us to attach observer patterns to insure program invariants. Some care must be taken, as conflicting invariants may cause infinite recursion as two collaborations try to maintain incompatible invariants.

Instantiation

Similar to the problems of maintaining consistency between object graphs is that of drawing object boundaries. If two class graphs differ in that one collaboration has a long path `anX.z.y` that is mapped to a shortcut path `anS.t`, with correspondences X:S and Y:T, then we are faced with the question of what to do should the program do for `asS.setT(new T());`

`T` corresponds to `Y`, so after adaptation the `new` will instantiate their combined role. The problem is that it is unclear what to do about `Z`. If it binds tighter to `Y`, as would be the case for a handle object or facade, then a new one needs to be instantiated to hold the new `Y`. Alternatively, it could bind tighter to `X`, in which case the `anX.z` relation is left alone.

The issue is that we are ignorant of the assumptions in the rest of the program; whether there is a one-to-one relationship between `Z` and `Y`, or `anX.z` is cached implying that we cannot modify it without breaking consistency.

Our current system requires that adapter writers provide code to perform these non-local set operations, forcing them to decide which is the proper solution. In more polished implementations, it would be preferable to automate the process via shorthand notation (perhaps this could be inferred from object model constraints).

6.2 Composition of Typing

We believe that typing composes; if two collaborations type check, then so will the resulting composition of the two as long as the adapter obeys some constraints. This claim is in need of a proof.

The most attractive approach seems to be to build on any of the existing light-weight semantics available for Java [IPW99,FKF98], and show that the preprocessor preserves typing, then to show that the constraints on adapters are sufficient to show that adaptation to compose roles also preserves typing.

Additionally, we need to show that the existential typing of return values and expected methods is sufficient to insure no casting errors occur in our inserted code for generic aspectual methods.

7 Related Work

Because we cross a number of domains, there are many areas of research which are related. They fall into four groups.

7.1 AspectJ

Aspects were identified and popularized by Lopes and Kiczales[KLM+97,Lop97], with the AspectJ implementation being their primary test bench. The focus is on integrating aspects into Java as full fledged feature such as inheritance. The system is implemented as a production compiler, taking extended source files and producing `.class` files. The reference implementation doesn't support separate compilation of aspects, although it is unclear whether this is an artifact of design or implementation.

The aspects offered by AspectJ are defined over pointcuts, which are points in the dynamic execution of the program such as a method invocation or variable lookup. This is more flexible than our aspects, which are limited to wrapping method bodies. Aspects are tightly integrated with the application to which they

are added, as pointcuts hardwire application names and classes into the aspect. Pointcuts can be abstract, and it is unclear whether this could be leveraged to loosen the coupling of an aspect to the rest of the application.

7.2 Collaborations

The idea of writing generic behavior in role based collaborations is an oft recurring theme.

Ian Holland's Ph.D. thesis [Hol93] had generic contracts that were written over separate class models, and attached to a host application via lenses that translated between the two models.

More recently, Mezini's and Seiter's Pluggable Components [ML98,MSL00] have researched dynamically attachable separately compiled generic behavior. Following Holland, they attach generic behavior to an application by means of an adapter object. The method is closely related to the Facade pattern [GHJV95], but includes automatic proxy wrapping and unwrapping of objects when passed from one class model to the other. Pluggable Components trade static type safety for dynamic flexibility. The implementation precludes aspectual attachment, unfortunately.

Mezini and Herrmann [HM00] discuss a software engineering environment capable of combining dynamic plugability, separate compilation, and aspectual attachment. It is unclear how their PIROL system deals with type safety.

This paper is based on many of the same foundations as Mezini and Seiter, but with goals of static type safety at the expense of flexibility.

7.3 Concerns

Concerns, like collaborations, are units of generic and task specific behavior. Subject-Oriented Programming [HO93,OKK+96] promotes the composition of a class from several partial class definitions. The SOP compiler offers several mechanisms to correctly combine classes, including overriding or sequentially ordering methods. This is very close to our own implementation, but while we focus on maintaining separate class models for each collaboration and the problems this introduces, SOP focuses on how two inheritance hierarchies are safely composed. Subject-Oriented Programming predates the popularization of Aspects, and thus does not use aspectual terms, but the its mechanisms are flexible enough to implement cross-cutting behavior [Cla00].

Multiple Dimensional Separation of Concerns and the Hyper/J [TOHS99] generalizes the ideas behind Subject-Oriented Programming by moving to finer grained units of combination. A HyperSlice is a named set of methods and fields in a set of classes. The slice can be added to new classes in a very similar way to collaborations. Similarly to our reuse of a Java compiler for type checking, we could probably have reused Hyper/J for weaving together our collaborations; we chose not to as our needs are very simple, and it seemed an equal amount of work to write our own class munger or to interface with Hyper/J.

7.4 Modules

Since we claim in our title to support modular programming, it is perhaps surprising that we don't call our units Modules or something similar. This is because the term's connotations do not match how a collaboration behaves. The reasons for this have been touched on section 2.4. In summary, modules behave like objects; many references point to one *thing*, which shares some hidden state for all references. In contrast, collaborations are more like library code that is inlined in the importer; there is no runtime representation of a collaboration. Hence, keeping encapsulated state shared between clients is more convenient with modules, while per client state is more natural with collaborations.

Ancona and Zucca [AZ00] detail JavaMod, a module calculus specialized for Java. Their module composition facility allows them to compose class models by adding new classes under renaming, but the do not support the modification of existing classes. The complication of a module system in Java is that the formal names of imports need to be linked to the actual imported classes. Java complicates matters by defining a name to be intrinsic to a class[14]; a class is *never* referred to by two different names.

Units [FF98] is a flexible module calculus that consistently applies extrinsic naming. In such a system, type equality amounts to graph search; if two types are defined in the same node, they are equal. This system allows a very natural specification of module composition. Units have been implemented for a MzScheme implementation system.

8 Conclusion

We have outlined an approach for combining separate compilation and reuse of multi-class behavior with aspectual attachment. Each collaboration is separately typechecked, and it is believed that this persists across composition.

We found that we were able to support two kinds of aspectual methods, with differing capabilities, and that these could be implemented with a simple pre/post processor pass around any Java compiler.

The system has proven surprisingly expressive, even in this early stage, allowing us to implement multi-role aspects such as a caching observer pattern in a completely generic manner. Small extensions to the wrapping machinery will allow us to provide intercessionary behaviors, allowing us to catch exceptions thrown by wrapped methods and decide whether to handle the error or rethrow it.

Powerful scoping constructs allow us to express precise sharing of state between aspectual methods, and straightforward composition semantics offer us

[14] Classical languages such as `Java`, `Modula 2`, `Pascal` assume intrinsic names to behavioral components such as methods and classes, which follows from the component not being first class values. In contrast functional languages tend to use extrinsic names, as they often treat such components as first class values. Imports in an extrinsically named language are easier, as the language already has mechanisms for applying different names to the same language component.

flexible mechanisms for building reusable behavior from simple collaboration building blocks.

The elaboration of the system uncovered a difficult problem with object graph consistency, which seems intrinsic to any approach which aims to combine various class models.

References

[AZ00] Davide Ancona and Elena Zucca. True modules for java classes. Technical Report DISI-TR-00-12, DISI - Università di Genova, August 2000.

[CHOT99] S. Clarke, W. Harrison, H. Ossher, and P. Tarr. Subject-oriented design: Towards improved alignment of requirements. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* in *Special Issue of SIGPLAN Notices*, pages 325–339, 1999.

[Cla00] Siobhán Clarke. Designing reusable patterns of cross-cutting behavior with composition patterns. In the Second Workshop on Multi-Dimensional Separation of Concerns in Software Engineering (ICSE 2000), 2000.

[FF98] M. Flatt and M. Felleisen. Units: Cool modules for hot languages. In *ACM Conference on Programming Language Design and Implementation*, pages 236–248, 1998.

[FF00] Robert Filman and Daniel Friedman. Aspect-oriented programming is quantification and obliviousness. In the Second Workshop on Multi-Dimensional Separation of Concerns in Software Engineering (ICSE 2000), 2000.

[FKF98] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *ACM Symposium on Principles of Programming Languages*, pages 171–183, 1998.

[FS97] M. Fowler and K. Scott. *UML Distilled. Applying the Standard Object Modeling Language.* Addison-Wesley, 1997.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[HM00] Stephan Herrmann and Mira Mezina. Pirol: A case study for multidimensional separation of concerns in software engineering environments. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* in *Special Issue of SIGPLAN Notices*, 2000.

[HO93] William Harrison and Harold Ossher. Subject-oriented programming (A critique of pure objects). In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* in *Special Issue of SIGPLAN Notices*, volume 28, pages 411–428, 1993.

[Hol93] Ian M. Holland. *The Design and Representation of Object-Oriented Components.* PhD thesis, Northeastern University, 1993.

[IPW99] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight java: A minimal core calculus for java and gj. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* in *Special Issue of SIGPLAN Notices*, 1999.

[KLM+97] G. Kiczalez, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. M. Loingtier, and J. Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming*, volume 1241, 1997.

[Lop97] Christina Videira Lopes. *A Language Framework for Distributed Programming.* PhD thesis, Northeastern University, 1997.

[ML98] Mira Mezini and Karl Lieberherr. Adaptive plug-and-play components for evolutionary software development. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* in *Special Issue of SIGPLAN Notices*, pages 97–116, 1998. NU-CCS-98-3.

[MSL00] Mira Mezini, Linda Seiter, and Karl Lieberherr. Component integration with pluggable composite adapters. In *Software Architectures and Component Technology*, 2000.

[OKK+96] Harold Ossher, Matthew Kaplan, Alexander Katz, William Harrison, and Vincent Kruskal. Specifying subject-oriented composition. *Software Practice and Experience*, 2(3):179–202, 1996.

[Ray00] Eric Raymond. http://www.tuxedo.org/ esr/jargon, 1975–2000.

[TOHS99] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *International Conference on Software Engineering*, pages 107–119, 1999.