Eliminating Cycles in Composed Class Hierarchies

Robert J. Walker
Department of Computer Science
University of British Columbia
201-2366 Main Mall
Vancouver, BC V6T 1Z4
Canada
walker@cs.ubc.ca

Technical Report TR-00-07

8 July 2000

Abstract

Multiple class hierarchies can be used each to represent a separate requirement or design concern. To yield a working system, these disparate hierarchies must be composed in a semantically meaningful way. However, cycles can arise in the composed inheritance graph that restrict the space of composable hierarchies. This work presents an approach to eliminating these cycles by means of separating the type hierarchy from the implementation hierarchy; separate solutions are provided for languages permitting multiple inheritance, such as C++, and those permitting only interfaces, such as Java. The resulting acyclic class hierarchy will maintain the significant constraints imposed by the original, separate hierarchies, such as type-safety.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-Oriented Programming; D.1.m [**Programming Techniques**]: Miscellaneous—*composition*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*object-oriented design methods*; D.2.3 [**Software Engineering**]: Coding Tools and Techniques.

General Terms

Algorithms, Design, Languages.

Keywords

Subtype, subclass, subject, flattening, correspondence, forwarding, integration, summary function, subject-oriented programming, effective subtype, class hierarchy, implementation hierarchy, type hierarchy.

1 Introduction

When one has, or needs to have, different views of an object-oriented software system, one can consider there to be more than one class hierarchy in existence, each modelling a different facet of the system. For example, each class hierarchy can represent a separate requirement or design concern [4, 3]. To create a working system, these separate class hierarchies must be combined together (*composed*) in such a way that the resulting classes define attributes and methods that are semantically correct combinations of the original hierarchies. The ways in which one may or should combine class hierarchies is part of the problem addressed by such work as subject-oriented programming [6, 9], where each class hierarchy is called a *subject*.

One of the difficulties that arises in such a composition process is the creation of cycles within the composed subject. We present a method for removing such cycles by means of separating the type hierarchy from the implementation hierarchy while preserving the significant constraints imposed by the individual input subjects prior to composition.

In Section 2, we describe background regarding composition of subjects, and how cycles arise in composed subjects. Section 3 relates our approach to eliminating these cycles. Section 4 details the complete algorithms for general graphs. We conclude the paper with Section 5.

2 Composing Subjects

Clarke *et al.* [4] detail why one would have different views of an object-oriented software system, each of which can be modelled in a separate subject. To create a working system, these separate subjects must be composed in such

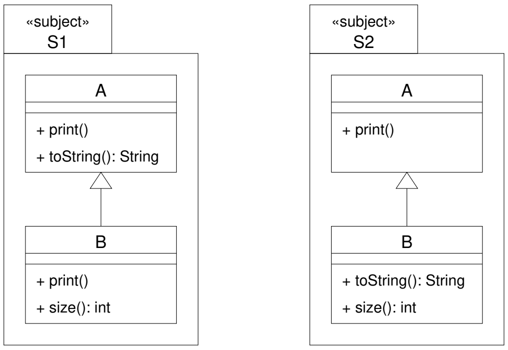

Figure 1: Two simple, input subjects.

a way that the resulting classes define attributes and methods that are semantically correct combinations of those from the original subjects.

Elements (such as classes, methods, and attributes) in separate subjects can be deemed, by a human being, to overlap or to represent the same concepts, and so, should be composed. Therefore, the correspondences between such elements and a means of integrating (i.e., combining) them need to be specified. A common means in subject-oriented programming for specifying correspondence is to specify that elements with identical names correspond, and for integration that invocation of the composed methods cause each of the original implementations to be invoked in some arbitrary order; this combination is called *merge-by-name*. For example, consider the toy example in Figure 1. Here, we see that the classes `A` and `B` are defined differently by the two subjects,[1] although the inheritance relationship is identical.

Composing these hierarchies through the merge-by-name relationship yields the composed hierarchy in Figure 5. Here we see that, where a method was defined in a class in both input hierarchies, the composed method simply calls one of the original methods, followed by a call to the other. But where class `B` does not override a method (e.g., `print()` in hierarchy `S2`, or `toString()` in hierarchy `S1`), the composed method still calls a method from each of the original hierarchies—in each such case, one is an inherited method.

Performing such a composition consists of a number of steps: flattening (Section 2.1), specification of correspondence (Section 2.2), integration (Section 2.3), unflattening (Section 2.4), and transformation of references (also Section 2.4). In Section 2.5, we demonstrate how this composition process may lead to cycles in the composed subject's class hierarchy.

[1] The separate subjects are shown as stereotyped UML™ packages, after the notation of Clarke [3]

2.1 Flattening

Composition of subjects can be performed by initially *flattening* the class hierarchy [10]. That is, every attribute declaration, operation declaration, and method implementation is copied down from the class that originally declared it to each of its subclasses. Flattening does not alter the interface to a class or its relationship to other types in a subject. In a flattened hierarchy, each class declaration is complete in itself, although a record is maintained of the original inheritance relationships for later use. For the sake of space and time efficiency, the actual copying does not occur until the end of the composition process because, in a later step, some of these methods may be *unflattened*, i.e., returned to their declaring classes; instead, references to the appropriate methods are made within auxiliary data structures used by a compositor tool. Unflattening is discussed further in Section 2.4.

Ossher *et al.* state that flattening is performed for three reasons [9, p. 183]:

1. "Since different subjects can have substantially different hierarchies, inheritance makes sense within a single subject, but often not when considering multiple subjects together."
2. "Combining inheritance hierarchies so as to preserve their separate effects can yield cycles."
3. "Languages with multiple inheritance tend to differ as to the semantic details, and we wish to avoid these differences [...]."

Points 1 and 3 indicate that flattening makes the algorithms for composition easier. Point 2 will be discussed in Section 2.5.

As an example, Ossher *et al.*'s subject-oriented compositor would flatten the hierarchies of Figure 1 to those shown in Figure 2. Here, we see that `S1.A.print()` and `S1.A.toString()` need to be flattened to `S1.B` and that `S2.A.print()` needs to be flattened to `S2.B`. `S1.A.print()` is renamed `S1.B.super_print()` when it is flattened, to avoid a name clash with the existing `S1.B.print()` method[2]. `S1.A.print()` needs to be flattened to `S1.B` for the sake of any calls to the superclass's methods within the implementations for `S1.B`'s methods; we will discuss the consequences of this in Section 2.4.

2.2 Correspondence and Forwarding

In order to compose a set of input subjects, we need to know how the entities within each subject *correspond*

[2] The particular renaming scheme used depends on the implementation of the compositor tool. This particular renaming is similar to, but different from, that used by Ossher *et al.*

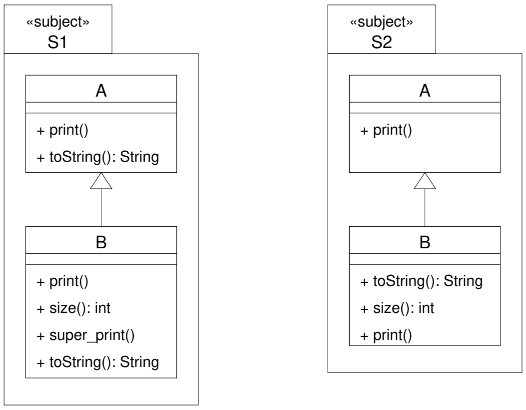

Figure 2: The input subjects after flattening.

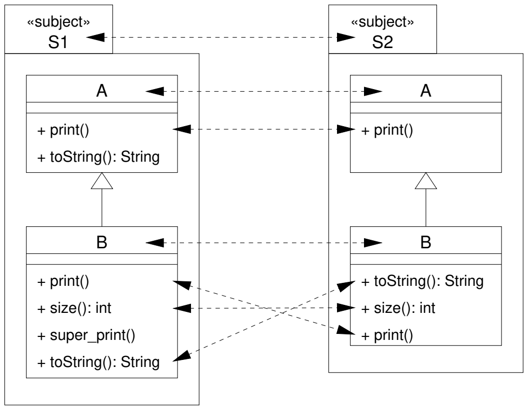

Figure 3: A correspondence specified for the flattened input subjects.

to those in the other subjects, and hence, which entities should be combined in some way to form the composed subject. There are some practical limitations to the denotation of which entities correspond, which we will describe in Section 2.3 in the context of integration.

There are many ways in which one could potentially specify correspondence. Ossher *et al.* [9] provide *composition rules*, whereby one can state, at a high level, rules for determining which entities correspond within an entire subject. Clarke [3] provides *composition relationships*, whereby one can state, at a lower level, which entities and their component entities correspond on the basis of simpler rules. Regardless of the particular interface provided for the task, at some point we obtain a specification stating which entities correspond. Two (or more) entities can correspond only if their parents also correspond. For example, it makes no sense in general to attempt to compose methods when we are not composing their declaring classes since each could be referencing the attributes and operations of their declaring class; if the classes are not composed, there is no way to guarantee that these references could be resolved, particularly in any semantically meaningful way. Figure 3 illustrates one possible correspondence between our example, flattened, input subjects, which is based upon entities with identical names being considered to correspond.

A given entity might exist simultaneously in multiple correspondence relationships. For example, it would be possible to specify that, in the subjects of Figure 1, `S1.A` corresponds with `S2.A` and that, separately, `S1.A` corresponds with `S2.B`. This leads to difficulties in transforming references after integration, however, as pointed out in Section 2.4. A solution is the concept of *forwarding*, where each input entity designates one correspondence as receiving its "identity" in a sense—the composed entity that will result from this correspondence is deemed to be the one that must take the place of the input entity. "Forwarding" takes its name from the fact that, when a call is made to a method of an input subject, this call is forwarded to a particular method in the composed subject.

The other correspondences involving this input entity are still useful for the sake of combining functionality in different ways. Consider wanting to combine two methods `a()` and `b()` in such a way that calling `a()` causes both implementations to be invoked while calling `b()` causes only the implementation of `b()` to be invoked. Such a situation will result if we specify that `a()` and `b()` correspond but that `b()` is also in a correspondence with nothing else and that `b()` should forward to the latter correspondence; such a situation is termed a *one-way correspondence*.

Within the context of this paper, we will sometimes wish to consider only the correspondences in which the participating entities forward to the correspondence—we are not interested whether other entities are involved in these correspondences but do not forward to them. We shall refer to this as a *restricted correspondence*.

2.3 Integration

Once we have a correspondence specification in hand, we may proceed to compose the input subjects accordingly. One issue remains that is not covered by correspondence alone: the means of *integration*. Two simple extremes for integration are *merge*, where the specification of a composed entity contains a true combination of input entities, and *override*, where a composed entity takes its specification from only one input entity thereby replacing the specifications of other input entities. For the most part, we will not be concerned with the specific kind of integration desired within this paper.

Merging the implementations of methods requires that all the participating input method implementations be invoked when the composed method is in-

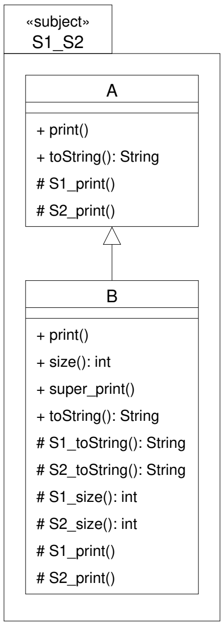

Figure 4: The integrated subject, prior to unflattening.

voked. The simplest means of performing this is to have a composed method be implemented as delegating [7] to each of the input method implementations, which, having been flattened, are placed into protected methods. For example, in Figure 4, `B.toString()` delegates to `B.S1_toString()` and `B.S2_toString()`, which contain the implementations from `S1.B.toString()` (originally from `S1.A.toString()`) and `S2.B.toString()`.

Note that arbitrary kinds of entities cannot be integrated: how does one invoke a method that is "integrated" with an attribute, for example? For this reason, correspondence is limited to entities of like kind: subject with subject, class with class, and so forth. Furthermore, there is no simple means by which to integrate methods with different signatures. Automated transformation of types is beyond our capabilities at present. Even if the parameter types are identical except for the addition of one or more parameters to one of the methods delegated to, there is no obvious means to fill these in when default values will not suffice. The work of Walker and Murphy [12] suggests one possibility, where such information can be reconstructed from the history of calls within the system, but that work is presently too much in its infancy to provide a sure solution. Instead, we simply consider such a correspondence to be untenable and that the two methods conflict.

Another problem remains: when we are merging methods with return values, we need to somehow combine the values that are returned. Tarr and Ossher [11] allow *summary functions* to be specified for this purpose.[3] A typical summary function might return the value of the last called implementation (hence, solely utilizing the others for their side-effects).

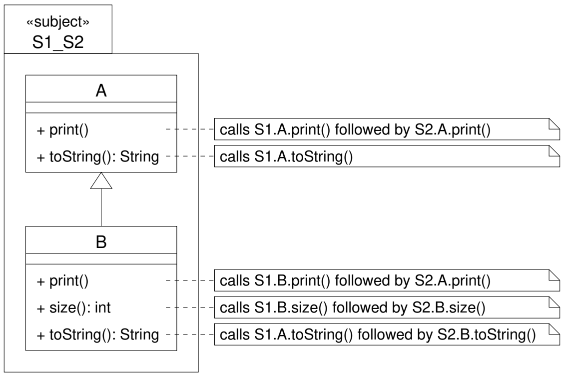

Figure 5: The unflattened, composed subject.

2.4 Unflattening and Transforming References

Once we have completed our integration, it may be the case that we do not need to keep all the methods in a class whose superclass also contains these methods. We may therefore *unflatten* the hierarchy to remove such cases.

Figure 5 shows the results of unflattening the composed subject in Figure 4, although the protected methods that are delegated to have been hidden. The unflattened methods from Figure 3 are as follows:

- `B.super_print()`, since it is identical to `A.S1_print()`;
- `B.S1_toString()`, since it is identical to `A.toString()`;
- `B.S2_print()`, since it is identical to `A.S2_print()`; and
- `B.print()`,[4] since it delegates to `S1_print()` and `S2_print()` for both `A` and `B`; `S1_print()` is simply overridden by `B`.

Once we have unflattened the class hierarchy in the composed subject, we need to ensure that all referenced names in signatures and implementations correspond to the correct names within the composed subject. One simple cause of a lack of agreement between referenced and actual names is the fact that names may need to be

[3] Ossher *et al.*'s earlier work on subject-oriented compositors also utilized summary functions, but this does not appear to have been documented in their publications, e.g., [9].

[4] This equivalence can be hard to detect, so this method might not be unflattened (as is the case in Figure 5).

changed to avoid conflicts with existing names. Also, an input entity may end up forwarding to a differently named entity in the composed subject, possibly due to its separate correspondence with multiple other entities. In fact, such multiple correspondences are the chief reason that forwarding is required: without it, there would be no way to resolve an input entity reference if that entity were in multiple correspondences.

2.5 Cycles

OO programming languages such as C++ [8] and Java™ [5] use class hierarchies to define both typing hierarchies and implementation inheritance hierarchies. The flattening/unflattening process affects which implementations are declared in which class, but since this is a post-source-level transformation of the code, it does not alter the burden of the programmer attempting to subclass within any particular input class hierarchy. But the flattening/unflattening process preserves the subtyping constraints defined in the input class hierarchies. Hence, in Figure 5 we see that class `B` remains a subclass of class `A`, and furthermore, that all the operations declared on both classes in each of the input hierarchies remain declared on the composed classes, although the implementations have altered (in a way that is semantically correct, or else the composition specification was invalid).

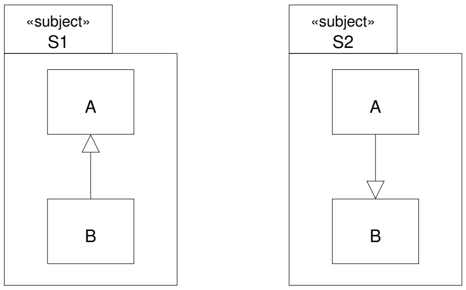

Figure 6: Two conflicting, input hierarchies.

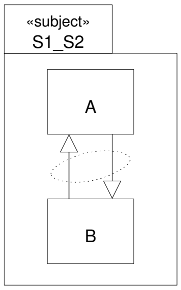

Figure 7: The cyclic hierarchy that results.

However, Ossher *et al.*'s point 2 (see Section 2.1) is not solved by flattening alone; yes, composed classes can be formed, but the subtyping relationships cannot be resolved if the class hierarchy is to define both the typing and implementation hierarchies. For example, consider the simple hierarchies in Figure 6; specifying that identically named classes in each subject correspond results in the cyclic class graph shown in Figure 7. We need a means to break such cycles that maintains the subtyping constraints required by each input subject.

3 The Approach

Taking our cue from OO programming languages, such as Modula-3 [1], that explicitly separate the typing and implementation hierarchies, we can eliminate cycles in the generalization hierarchy while maintaining the necessary subtyping relationships that are required for the composed system to be compilable and type-safe. There are two separate, but closely related, approaches, one for each of two categories of OO programming language: (category 1) those that permit multiple inheritance (e.g., C++), and (category 2) those that permit specification of separate interfaces, with individual classes able to implement multiple such interfaces (e.g., Java). We assume subtyping as required by the conditions of Castagna [2], but we currently limit our discussion to singly-dispatched languages.

We begin by looking at the simple case of a composed class graph consisting of a single cycle (Section 3.1), then extend this to slightly larger graphs (Section 3.2), as the implications of the two approaches for the different language categories becomes more apparent and non-trivial for more topologically interesting graphs. We then consider the effects of the approach on parameter-, variable-, and return-types (Section 3.3), which we will ignore in Sections 3.1 and 3.2.

3.1 Isolated Cycles

As a first step, we will consider simple class graphs consisting solely of a single cycle, such as occurs in Figure 7. Consider the composed classes `A` and `B` prior to attempting to fit them into an acyclic generalization graph. If we look closely at the generalization relationships required by the class graphs of Figure 6, we see that the requirements are weaker than those implied by the composed graph. If we speak loosely, `S1_S2.B` is only required to be a specialization of `S1.A` and not of `S2.A`, and `S1_S2.A` is only required to be a specialization of `S2.B` and not of `S1.B`. So let us enforce the required generalization relationships indirectly.

For category 1 languages, we can introduce two abstract classes to each of our input subjects: `AIntf` and `BIntf`. Each of these declares operations (i.e., no implementations) identical to those declared in the analogous class from its input class hierarchy. The added abstract classes have generalization relationships defined between them that are analogous to those for the concrete classes with which they are associated. Now we make each of our original classes specialize its corresponding abstract class, as shown in Figure 8. This process does not add or remove any attribute declarations, method declarations, or method implementations as originally specified within the input classes; it has simply made the interfaces and types explicit.

We may then proceed to flatten these hierarchies and compose them as before—with two important differences:

1. we will ignore the generalization relationships between the concrete classes in each input class hierarchy, and

2. the abstract classes in the two input class hierarchies will be considered *non-corresponding*, and hence, will not be integrated.

Figure 9 shows the result of flattening and the correspondences that are specified between the classes.

Finally, we compose the flattened, input class hierarchies, integrate them according to the correspondences defined, and unflatten them, resulting in the composed class hierarchy shown in Figure 10. The subtyping relationships required by the input class hierarchies are maintained; for example, `S1_S2.A` is a valid `S1.A`, `S2.A`, and `S2.B`, in terms of type, once the appropriate renaming is performed.

The process for category 2 languages is a straightforward analogy with that for category 1 languages, save that interfaces are used in place of abstract classes. The composed hierarchy that eliminates the cycle of Figure 7 for category 2 languages is shown in Figure 11.

Larger cycles are handled as a straightforward extension to this process. The point being that we cannot select any particular class within the cycle as being a reasonable supertype for the others, so no generalization relationships are to be maintained directly between any of the classes within the cycle.

3.2 Cycles Embedded in a Larger Graph

Before dealing with a composed graph of arbitrary topology, we consider the special case where every class in a cycle has both a particular superclass and subclass. Such a case arises when we have input class hierarchies such as those illustrated in Figure 12, which are equivalent to those of Figure 6 but each with a class `Top` and a class `Bot` added.

For a category 1 language, we proceed as in the case of an isolated cycle, first explicitly separating the inter-

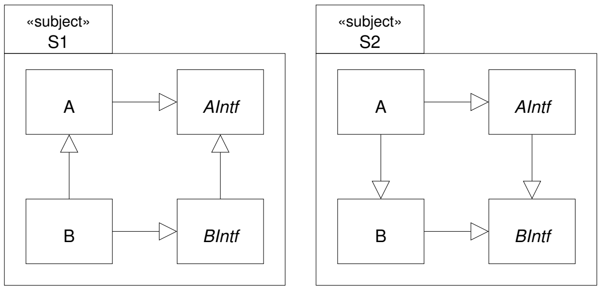

Figure 8: The input hierarchies of Figure 6, having had their interfaces explicitly separated.

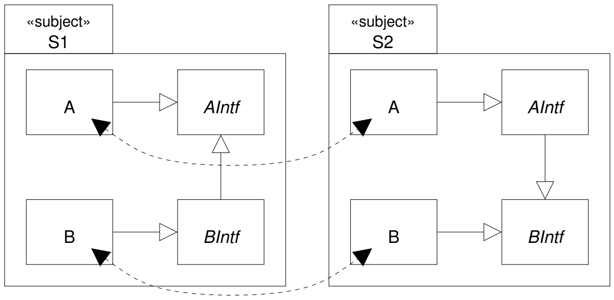

Figure 9: The input hierarchies of Figure 8, having been flattened, with the desired correspondences depicted.

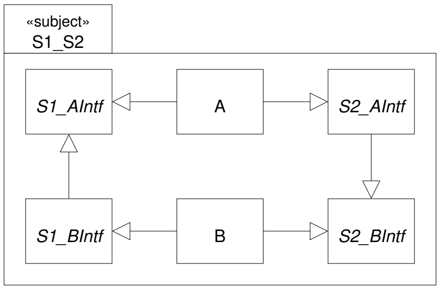

Figure 10: The composed hierarchy resulting from Figure 9.

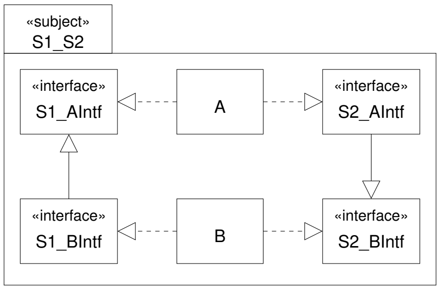

Figure 11: The composed hierarchy for category 2 languages that is equivalent to Figure 10.

faces of each class as an abstract class in a parallel hierarchy, and having each concrete class extend its analogue abstract class; Figure 13 is the result.

Again, we flatten the hierarchies, make only the concrete classes correspond, and eliminate the generalization relationships between the concrete classes, allowing the type hierarchy to be maintained by the abstract classes. The composed hierarchy is shown in Figure 14.

But we can do better than this for a category 1 language if we look at the class graph that would have resulted from a straightforward composition of the hierarchies in Figure 12, as shown in Figure 15. Here, we note that only classes `A` and `B` are involved in a cycle, `Top` and `Bot` are not; it would be better if the acyclic result only contained the abstract classes where it really needed to do so. With this in mind, we can eliminate the `TopIntf` and `BotIntf` abstract classes from both input hierarchies, and have the remaining abstract classes be inserted into the chain from `Top` to `Bot` as shown in Figure 16. Finally, the composed hierarchy appears as in Figure 17.

For a category 2 language, the optimization shown in Figure 17 is not available to us, because an interface cannot generalize a class nor vice versa. However, we can still do better than the interface analogue of Figure 14, because the interfaces from `S1` and `S2` for `Top` and `Bot` do not need to be kept separate; thus, Figure 18 is the result for a category 2 language.

In both approaches, we may unflatten the implementation hierarchies more if we wish; we can see from Figure 15 that only the generalization relationships resulting in the cycle are objectionable. Thus, we can have `S1_S2.A` and `S1_S2.B` extend `S1_S2.Top`, and `S1_S2.Bot` extend `S1_S2.Top` for both approaches; furthermore, for category 1 languages, `S1_S2.Bot` could extend both `S1_S2.A` and `S1_S2.B` instead of `S1_S2.Top`. This would largely be an exercise in conserving space, however.

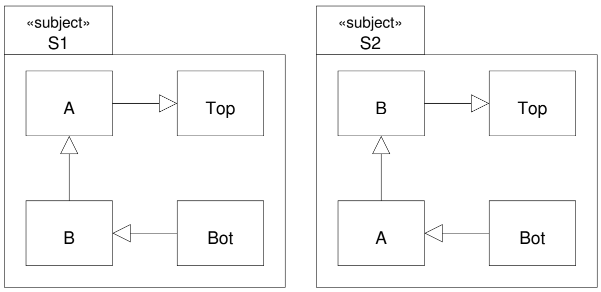

Figure 12: Two conflicting, input hierarchies.

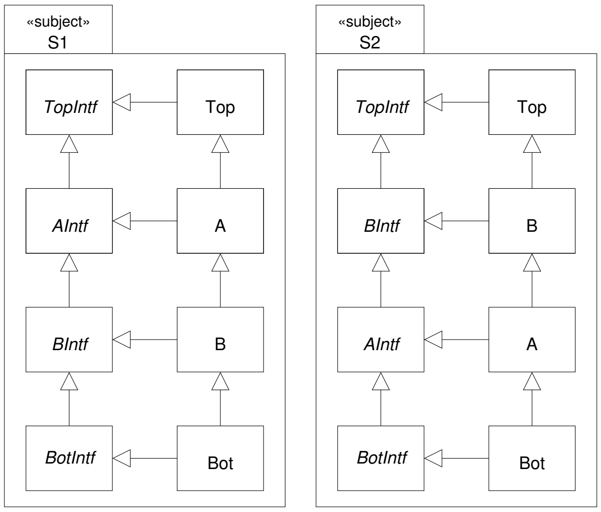

Figure 13: The input hierarchies, having had their interfaces explicitly separated.

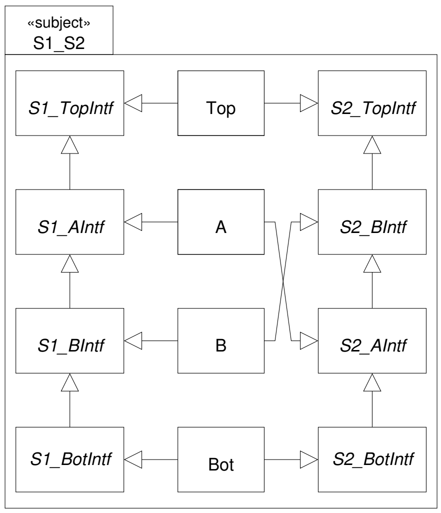

Figure 14: The composed hierarchy.

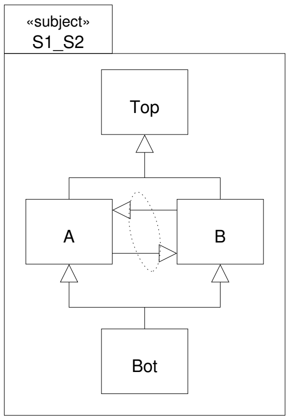

Figure 15: The composed hierarchy, without applying cycle elimination.

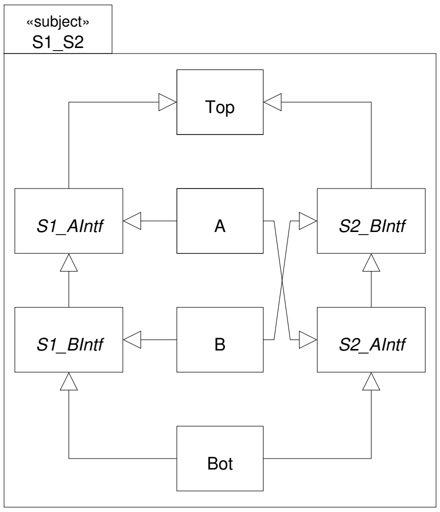

Figure 17: The optimized, composed hierarchy.

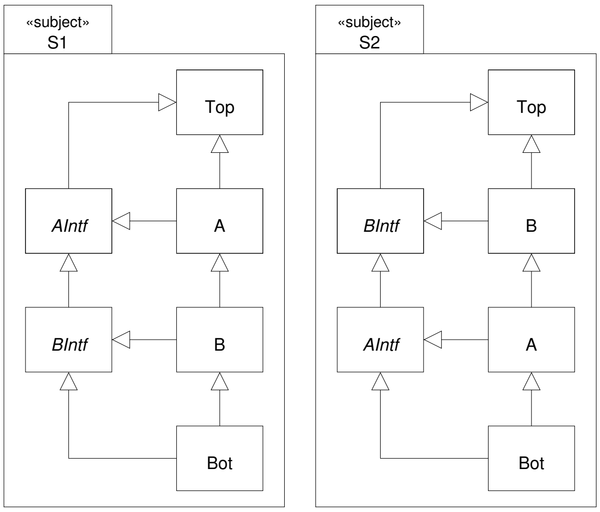

Figure 16: The input hierarchies, having had unneeded abstract classes removed.

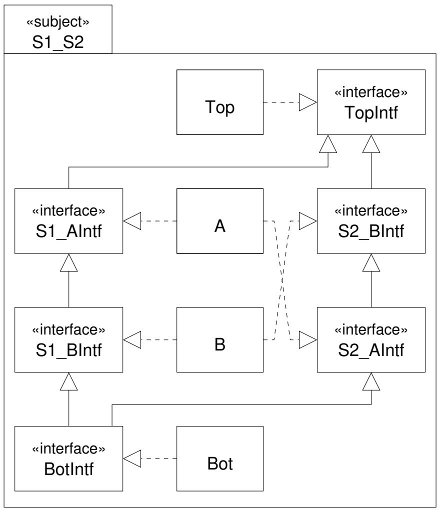

Figure 18: The optimized, composed hierarchy for a category 2 language.

3.3 Dealing with Parameter and Variable Types

We have thus far swept an important issue aside: the abstract classes and interfaces that we add to the input class hierarchies to explicitly separate typing cannot actually have the identical interfaces to the classes with which they are associated!

Consider a class B that declares an operation process() returning a value of type B. Let such a B reside in two input subjects S1 and S2, as in Figure 19. In S1, B is generalized by A, while in S2, B generalizes A. Furthermore, S2.A overrides process(), but returns an object of class A as type B. If we define a correspondence between S1 and S2 such that like-named elements correspond, our composed graph will contain a cycle, just as in Figure 7. To eliminate the cycle, our technique will introduce two abstract classes or interfaces (depending on the language with which we are dealing) AIntf and BIntf to each of the input subjects, for A and B respectively. According to the description of the technique given so far, S2.AIntf, S1.BIntf and S2.BIntf should each declare an operation process() returning a value of type B; after composition, we would then have the composed graph shown in Figure 20 where B.process() delegates to the protected methods of B (namely, _S1_process() and _S2_process()), which were the implementations declared in the input subjects. But there is a problem with this picture: the implementation of A.process() was copied from S2.A, which returns an object of class A. Since the generalization relationship between A and B has been broken, the composed hierarchy will not be type-safe.

The key point that has been missed here is that the class information (as opposed to type information) has not been completely removed from the interfaces and implementations; in short, AIntf and BIntf should be referred to, and not A and B, everywhere except in constructor invocations. Thus, S1_BIntf should declare process() to have return a return type of S1_BIntf while S2_AIntf and S2_BIntf should declare process() to have return a return type of S2_BIntf.

This yields a problem of its own, of course—namely that S1_S2.B needs to implement two operations that are overloaded solely on the basis of return type, something that is illegal in most languages. To get around this, we rename process() to S1_process() for the operation from S1 and to S2_process() for the other. Figure 21 shows the resulting situation. The implementation for each of these methods must delegate to the two implementations being composed from the input subjects. The protected methods of B (namely, _S1_process() and _S2_process()) contain these original implementations from the input subjects. The public methods

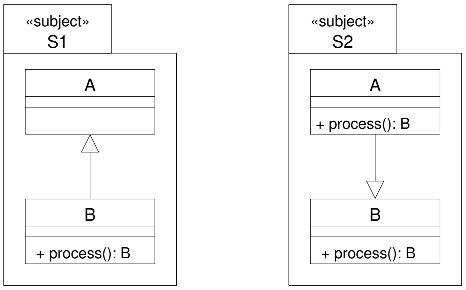

Figure 19: Input class hierarchies leading to difficulties with return value types.

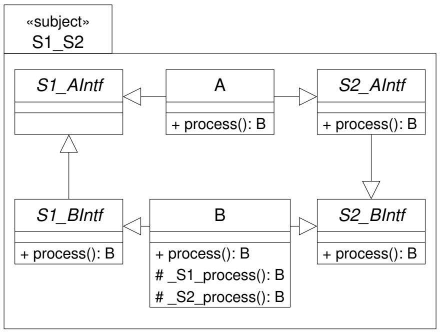

Figure 20: The composed graph with a problematic return type in A.process().

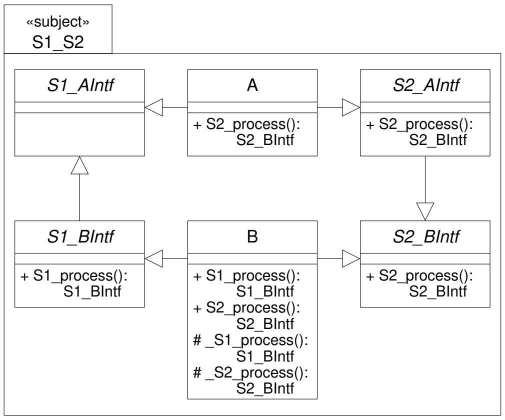

Figure 21: The composed graph after repairing type references.

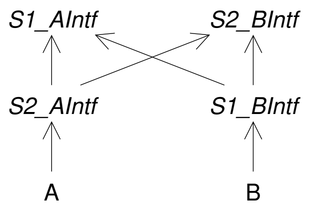

Figure 22: The effective subtype graph for `S1_S2`.

cast the return values of the protected methods to obtain a value of the appropriate return type. For example, `B.S2_process()` is implemented as (using C++ syntax):

```
S2_BIntf B::S2_process() {
  S2_BIntf rv1, rv2;
  rv1 = (S2_BIntf)_S1_process();
  rv2 = _S2_process();
  return summaryFunc(rv1, rv2);
}
```

where `summaryFunc` is a summary function that has been specified to combine the return values in some desired fashion. Now, `B.S1_process()` should seemingly be implemented as:

```
S1_BIntf B::S1_process() {
  S1_BIntf rv1, rv2;
  rv1 = _S1_process();
  rv2 = (S1_BIntf)_S2_process();
  return summaryFunc(rv1, rv2);
}
```

but notice that there is a problem here: the return type of `B._S2_process()` is `S2_BIntf`, and an instance of class `A` could legally be passed as an `S2_BIntf`. In other words, the type cast in `B.S1_process()` is not safe. On the other hand, the type cast in `B.S2_process()` *is* safe, since any object legally passed as an `S1_BIntf` (i.e., an instance of `B`) is also legally representable as an `S2_BIntf`.

This situation can be seen more clearly if we look at the *effective subtype* graph for `S1_S2`, shown in Figure 22. Each arrow represents the relation "can be safely cast to", or "is effectively a subtype of".[5] Since `S1_BIntf` is effectively a subtype of `S2_BIntf` but not vice versa, the implementation of `B.S2_process()` given above is type safe but that of `B.S1_process()` is not.

The result of the cycle elimination process has been to introduce an asymmetry, or one-way correspondence, in the types that have made the seemingly identical signatures for `process()` in the input subjects actually conflict. To make the given implementation of `B.S1_process()` type safe, we would need to introduce an algorithm for type conversion from an instance of `A` to an instance of `B`, something that is not semantically trivial, and so, not a simple candidate for automation. The question remains whether we should still allow `B.S2_process()` to be implemented as given, but whether a one-way correspondence is acceptable depends on the semantics of composition needed in the given situation—at least type safety is assured in the one direction. Regardless, as in any situation involving conflicts in correspondence, strict automation is not an option in the foreseeable future.

This situation will exist for any types involved in cycles, and will affect both return types and parameter types. However, due to the contravariant typing we have assumed, the situation is backwards for parameters. Consider having a method that accepts a parameter of type `B` but returning no value: `process(B)`. We alter the input subjects to replace `process()` with `process(B)` and go through a process analogous to that for the return type, obtaining the composed subject in Figure 23. Note that, since we are overloading on the basis of parameter types, we do not need to use different names for the two versions of `B.process`.

Since the effective subtype graph remains that of Figure 22, we proceed to implement `B.process(S2_BIntf)` as follows:

```
void B::process(S2_BIntf inVar) {
  _process((S1_BIntf)inVar);
  _process(inVar);
}
```

but again, this is not a safe type cast since an instance

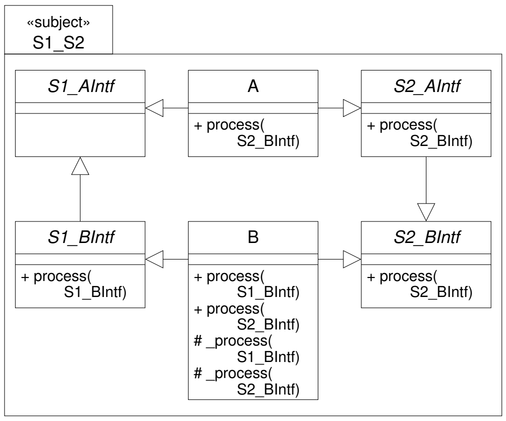

Figure 23: The composed graph with parameters, after repairing type references.

[5] We say "effectively" since the interfaces that are effective subtypes fail to declare the operations declared by their effective supertypes, and they are not implicitly declared since no inheritance relationship exists between them.

of class A could have been passed in `inVar` and A cannot fulfill the role of an `S1_BIntf`. Now, we see that the one-way correspondence could be in the opposite direction. In a similar situation involving both parameters and return types, the methods would be in strict conflict.

Local variables, while needing their types modified, will not otherwise be affected directly, since the methods in which they are embedded will be operating from a single perspective, namely the input subject in which they were defined.

While this affects the legal ways in which correspondence can be defined, it does not directly affect the approach, given a non-conflicting correspondence, so we press forward and present the general solution.

4 The General Algorithms

The general problem of a composed graph with arbitrary topology utilizes the same details as in the last section; every class involved in any cycle has its input classes explicitly separate their type interfaces, and the type interfaces do not correspond, remaining separate in the composed subject.

We begin with a formal description of correspondence and composition in Section 4.1, continue with the algorithm for category 1 languages in Section 4.2, and end with the algorithm for category 2 languages in Section 4.3.

4.1 Correspondence and Composition

A class graph G is an ordered pair (C, H) where C is a set of classes (and interfaces), and H is a set of ordered pairs (c_i, c_j), $i \neq j$, where c_i is a generalization of (or realization by) c_j and both c_i and c_j are elements of C. For a set of input class graphs $\{G_1, G_2, \ldots, G_n\}$, a restricted correspondence κ is a mapping from their classes to $\{0, 1\}$, where 0 is considered *non-corresponding* and 1 is considered *corresponding*; in other words,

$$\kappa : \mathcal{P}\left(\bigcup_i C_i\right) \setminus \emptyset \longrightarrow \{0,1\},$$

subject to the conditions:

1. each input class must correspond with something (even if only itself), i.e.,

$$\forall c \in \bigcup_i C_i \;\; \exists S \mid c \in S \wedge \kappa(S) = 1; \text{ and,}$$

2. each input class must correspond only with one set of classes, i.e.,

$$\forall c \in \bigcup_i C_i \;\; \exists S_i, S_j \mid \\ c \in S_i \wedge c \in S_j \Leftrightarrow S_i = S_j.$$

Let $G^* = (C^*, H^*)$ be a composed class graph that is not an input class graph, i.e., for all i, $G^* \neq G_i$. A composition $\mathfrak{K}_\kappa$ is a mapping from a set of input class graphs to an output class graph that is induced by a restricted correspondence. The output class graph is defined by the following conditions:

1. there is a unique class in the output class graph for every input set that corresponds, i.e.,

$$\exists \text{ bijection } \mathfrak{K}_\kappa^C : \mathbf{S} \rightarrow C^* \;\; \forall S \in \mathbf{S} \mid \\ \kappa(S) = 1; \text{ and,}$$

2. each generalization in the input subjects cause there to be a generalization in the output subject between the resulting composed classes, except that no self-loops are induced, i.e.,

$$\forall (c_j^*, c_k^*) \in H^* \; \exists (C_i, H_i), S, T \mid \\ (c_l, c_m) \in H_i \wedge c_l, c_m \in C_i \wedge c_l \in S \wedge \\ c_m \in T \wedge \kappa(S) = 1 \wedge \kappa(T) = 1 \wedge j \neq k,$$

and

$$\forall (c_j, c_k) \in \bigcup_i H_i \; \exists S, T, c_j^*, c_k^* \mid \\ (c_j^*, c_k^*) \in H^* \wedge c_j \in S \wedge c_k \in T \\ \wedge \mathfrak{K}_\kappa^C(S) = c_j^* \wedge \mathfrak{K}_\kappa^C(T) = c_k^*.$$

Let $\mathcal{C}^*$ be the set of all classes in the composed graph G^* involved in one or more cycles. For every composed class $c_i^* \in \mathcal{C}^*$, there exists a set X_i of classes of which it is composed, i.e., $\mathfrak{K}_\kappa^C(X_i) = c_i^*$. We define $\mathcal{C} \equiv \bigcup_i X_i$ to be the set of classes that reside in the input class graphs that, after composition, have their composed analogues involved in one or more cycles. For every class $c_i \in \mathcal{C}$, c_i resides in some input class graph G_i. Each c_i generalizes a set of classes ∇_i in G_i and is generalized by a set of classes Δ_i in G_i. We define $\nabla_i^{\mathcal{C}}$ to be those classes generalized by c_i that are themselves elements of $\mathcal{C}$, and $\Delta_i^{\mathcal{C}}$ to be those classes generalizing c_i that are themselves elements of $\mathcal{C}$.

4.2 Category 1 Languages

For category 1 languages, we proceed as follows. For every class $c_i \in \mathcal{C}$, we add an abstract class a_i to input class hierarchy G_i, and have a_i generalize c_i. For every class $\check{c}_i \in \nabla_i^{\mathcal{C}}$, we have a_i generalize the abstract class $\check{a}_i$ that has been added to G_i for the sake of $\check{c}_i$. Likewise, for every class $\hat{c}_i \in \Delta_i^{\mathcal{C}}$, we have a_i be generalized by the abstract class $\hat{a}_i$ that has been added to G_i

for the sake of $\hat{c}_i$. We use our optimization step of Section 3.2 for category 1 languages as follows. For every class $\check{c}_i \in \nabla_i \setminus \nabla_i^{\mathcal{C}}$, we have a_i generalize $\check{c}_i$ directly, and for every class $\hat{c}_i \in \Delta_i \setminus \Delta_i^{\mathcal{C}}$, we have a_i be generalized by $\hat{c}_i$ directly. We now have the general equivalent of Figure 16.

For each $c_i \in \mathcal{C}$, we remove all generalizations $(c_i, \check{c}_i)$ and $(\hat{c}_i, c_i)$ for all $\check{c}_i \in \nabla_i^{\mathcal{C}}$ and $\hat{c}_i \in \Delta_i^{\mathcal{C}}$. We now have a transformed set of input class graphs, ready for composition. We define κ' to be the extension of κ where the mapping is identically 0 for the portion of the extended domain that lies outside the original domain $\mathcal{P}\left(\bigcup_i C_i\right) \setminus \emptyset$. As a result, $\mathfrak{K}_{\kappa'}^{C}$ is the extension of $\mathfrak{K}_{\kappa}^{C}$, where the mapping is undefined for the portion of the domain that lies outside $\mathcal{P}\left(\bigcup_i C_i\right) \setminus \emptyset$. In other words, for every class $c_i^* \in \mathcal{C}^*$, we still specify that all the elements of X_i correspond, but none of the added abstract classes correspond to any other classes, and we insert the analogous generalizations. We integrate according to the specified correspondences, unflatten the hierarchies as much as possible, and we obtain the general equivalent of Figure 17, a composed graph $G^{*\prime}$ with no cycles.

4.3 Category 2 Languages

For category 2 languages, we proceed as follows. For every class $c_i \in \bigcup_i C_i$, we add an interface ι_i to input class graph G_i, and have c_i realize ι_i. For every class $\check{c}_i \in \nabla_i$, we have ι_i generalize the interface $\check{\iota}_i$ that has been added to G_i for the sake of $\check{c}_i$. Likewise, for every class $\hat{c}_i \in \Delta_i$, we have ι_i be generalized by the interface $\hat{\iota}_i$ that has been added to G_i for the sake of $\hat{c}_i$.

We flatten the hierarchies and, for each $c_i^* \in \mathcal{C}^*$, we remove all generalizations $(c_i^*, \check{c}_i^*)$ and $(\hat{c}_i^*, c_i^*)$ for all $\check{c}_i^* \in \nabla^*$ and $\hat{c}_i^* \in \Delta^*$. We use our optimization step of Section 3.2 for category 2 languages as follows. For every class $c_i^* \in \bigcup_i C_i \setminus \mathcal{C}^*$, we define κ' such that, for the set of classes X_i composed into c_i^*, $\kappa'(I_i) = 1$ for the set of interfaces I_i added for the sake of X_i, $\kappa'(X_i) = 1$, and κ' is identically 0 everywhere else. As a result, $\mathfrak{K}_{\kappa'}^{C}$ is the extension of $\mathfrak{K}_{\kappa}^{C}$, where the mapping is defined for the portion of the domain which lies outside $\mathcal{P}\left(\bigcup_i C_i\right) \setminus \emptyset$ only for each set of interfaces I_i as described above. We integrate according to the specified correspondences, unflatten the hierarchies as much as possible, and we obtain the general equivalent of Figure 18, a composed graph $G^{*\prime}$ with no cycles.

Note that any interfaces in the input class graphs for a category 2 language can themselves form cycles upon composition. These may be dealt with analogously to category 1 languages, since an interface may be generalized by multiple interfaces.

5 Conclusion

This paper has presented an approach for the elimination of cycles from class hierarchies that result from composition. The approach is based upon the principle of separating the type hierarchy from the implementation hierarchy. One consequence to the approach is that seemingly identical method signatures can actually end up being partially conflicting, making their integration non-trivial in any automated fashion. There is also the fact that, in Java, invoking a method via an interface is typically slower than invoking a method via a class in current implementations of the Java virtual machine.

These consequences would seem to be a necessary evil, however, if one insists upon specifying cycles in the composed class graph. While one might consider the given cycle elimination approach to be unsatisfactory given the consequences, we had before a situation where cycles caused a dead-end in the composition process; now, although we need to resort to a certain amount of manual intervention, we may at least define a solution.

Acknowledgements

Many thanks to Siobhán Clarke and Gail Murphy for their comments on early drafts of this work, and to Harold Ossher for his comments on the approach. “Java” is a trademark of Sun Microsystems. “UML” is a trademark of Object Management Group, Inc.

References

[1] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 report (revised). *ACM SIGPLAN Notices*, 27(8):15–42, August 1992.

[2] Giuseppe Castagna. Covariance and contravariance: Conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431–447, May 1995.

[3] Siobhán Clarke. *Composition of Object-Oriented Software Design Models*. PhD thesis, School of Computer Applications, Dublin City University, Dublin, Ireland, 2000. To appear.

[4] Siobhán Clarke, William Harrison, Harold Ossher, and Peri Tarr. Subject-oriented design: Towards improved alignment of requirements, design and code. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 325–339, Denver, CO, USA, 1–5 November 1999. Published as ACM SIGPLAN Notices 34(10), October 1999.

[5] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java™ Language Specification*. The Java Series. Addison-Wesley Publishing Company, second edition, June 2000.

[6] William Harrison and Harold Ossher. Subject-oriented programming (a critique of pure objects). In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 411–428, Washington, DC, USA, 26 September–1 October 1993. Published as ACM SIGPLAN Notices 28(10), 1 October 1993.

[7] William Harrison, Harold Ossher, and Peri Tarr. Using delegation for software and subject composition. Research Report RC 20946, IBM T. J. Watson Research Center, Yorktown Heights, NY, USA, 5 August 1997.

[8] International Standards Organization and International Electrotechnical Commission. *Programming languages—C++*, 1 September 1998. International standard ISO/IEC 14882.

[9] Harold Ossher, Matthew Kaplan, Alexander Katz, William Harrison, and Vincent Kruskal. Specifying subject-oriented composition. *Theory and Practice of Object Systems*, 2(3):179–202, 1996.

[10] Céline Rouveirol. Flattening and saturation: Two representation changes for generalization. *Machine Learning*, 14:219–232, 1994.

[11] Peri Tarr and Harold Ossher. *Hyper/J™ User and Installation Manual*. IBM T. J. Watson Research Center, Yorktown Heights, NY, USA, 2000. `http://www.research.ibm.com/hyperspace`.

[12] Robert J. Walker and Gail C. Murphy. Implicit context: Easing software evolution and reuse. In *8th International Symposium on the Foundations of Software Engineering*, San Diego, CA, USA, 6–10 November 2000. To appear.