To appear in ECOOP '93 Proceedings, Springer Verlag Lecture Notes on Computer Science.

Integrating Independently-Developed Components in Object-Oriented Languages

Urs Hölzle
Computer Systems Laboratory
Stanford University
urs@cs.stanford.edu

Abstract. Object-oriented programming promises to increase programmer productivity through better reuse of existing code. However, reuse is not yet pervasive in today's object-oriented programs. Why is this so? We argue that one reason is that current programming languages and environments assume that components are perfectly coordinated. Yet in a world where programs are mostly composed out of reusable components, these components are not likely to be completely integrated because the sheer number of components would make global coordination impractical. Given that seemingly minor inconsistencies between individually designed components would exist, we examine how they can lead to integration problems with current programming language mechanisms. We discuss several reuse mechanisms that can adapt a component *in place* without requiring access to the component's source code and without needing to re-typecheck it.

1 Introduction

Object-oriented programming promises to increase programmer productivity through better reuse of existing code. In the ideal scenario envisioned by the object-oriented community, future programs would be mostly composed out of preexisting components rather than rewritten from scratch: "[Programmers will] produce reusable software components by assembling components of other programmers" [Cox86]. Reusing existing, tested code simultaneously reduces the effort needed to create new applications and improves the quality of the resulting programs.

However, this ideal scenario has not yet become reality: today, many programs are still mostly written from scratch, and there are few commercially available building blocks that could be reused in new applications. Clearly, the envisioned market for software ICs [Cox86] has not materialized yet. Writing reusable components and frameworks is hard [Deu83, OO88, OO90]. Furthermore, the few successful reusable components built so far (such as Interviews [LVC89] and ET++ [WGM88]) are relatively monolithic frameworks, and components are often hard to combine with other components [Ber90]. Why is this so? Why hasn't the dream of *pervasive reuse* been realized yet?

In this study we try to answer this question and come to the conclusion that one reason is that current programming languages (and programming environments) are not very well equipped to handle the subtle integration problems likely to occur with pervasive reuse. To illustrate our point, we envision a futuristic world where reuse is indeed pervasive and where programs are mostly composed out of reusable components. We then argue that in such a world components would not be likely to be completely integrated because the sheer number of components would make global coordination impractical. Given that minor inconsistencies between individually designed components would always exist, we examine how these inconsistencies can lead to problems when combining the components, and how such problems could be overcome.

The paper is divided into two parts. In the first part, we clarify our assumptions and present a simple example of an integration problem caused by a few small inconsistencies. We then try to solve this problem using mechanisms present in many current object-oriented languages (composition, subtyping, and multiple dispatch) and show that none of these approaches completely succeeds in solving the problem.

In the second part, we argue that a broader view of reuse should be adopted, a view that includes an imperfect world where component interfaces are not completely coordinated. Programming languages and environments need to provide mechanisms that allow the programmer to make small changes to component interfaces without having access to the components' source code, and components should be delivered in a form that allows such modifications. We discuss several possible approaches that promise to solve integration problems and thus might bring us one step closer to a world of pervasive reuse.

2 Assumptions

Our study is based on four premises:

- that programs will be composed out of existing components,
- that such components were designed independently, and are sold by independent vendors,
- that a component's source code cannot or should not be changed, and
- that programs are written in a statically-typed language.

The first premise is one of the primary goals of object-oriented programming. We will assume that a large portion of new programs (say, more than 90%) consists of reused components. Thus, relatively little time is spent writing new code, and most of the programming effort lies in combining reusable components. Therefore, it is imperative that components can be combined easily so that programmer productivity is maximized.

The second premise follows from the first: if programs consist mainly of reused code, there must be many reusable components (thousands or tens of thousands), and, assuming a free market, these components must have been designed by many different organizations and vendors. Thus, it is very likely that many components were designed independently, without knowledge of each other. Even if a component vendor tried to be compatible (in some sense of the word) with other components, the sheer number of different components would make it impractical to verify compatibility for all combinations. Therefore, it may well be that a new application is the first application to combine two particular components, and that these two components are not perfectly well-integrated with each other. Many vendors will probably integrate their own components into frameworks, but this does not fundamentally change our assumption: there will be hundreds of frameworks, and as soon as an application reuses two or more independent frameworks, integration problems are likely to occur.

The third premise follows from the desire for incremental change: "Adding new code is good, changing old code is bad" [Lie88]. It is motivated by pragmatic necessity: vendors may not be willing to make their source code available to clients, and even if source code is available, it is desirable not to change the source in order to stay compatible with future versions of the component. If components become much more widely used than today, it will be even more important not to rely on any internal details of the components, or else tracking new versions of the tens or hundreds of components used by an application will become a software maintenance nightmare.

Finally, we are concerned only with statically-typed languages using subtype polymorphism [CW85], because these languages are so widely used and because static typing is often viewed as necessary for building reliable large systems (see, for example, [CC+89]). However, many of the problems we discuss also apply to dynamically-typed object-oriented languages (see section 5.1).

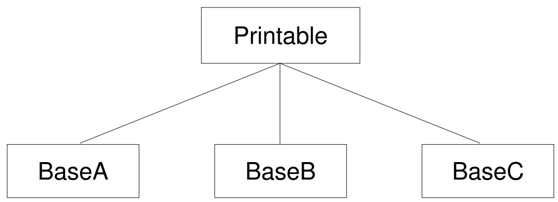
3 The Problem: Inconsistent Components

Assume that we are writing an application using the components A, B, and C, each of them bought from a different vendor. The simplified interfaces of the types offered by the components are shown below:

```
type BaseA {              "BaseA is the supertype of all objects in component A"
     method print();
     methods mbase_a1, mbase_a2,...
}

type SubA: BaseA {                         "One of many subtypes of BaseA"
     methods msub_a1, msub_a2,...
}

type BaseB {              "BaseB is the supertype of all objects in component B"
     method printPart1();          "prints one part of a BaseB object"
     method printPart2();          "prints the other part of a BaseB object"
     methods mbase_b1, mbase_b2,...
}

type SubB: BaseB {                         "One of many subtypes of BaseB"
     methods msub_b1, msub_b2,...
}

type BaseC {              "BaseC is the supertype of all objects in component C"
     method drucke();                      "'drucke' is German for 'print'"
     methods mbase_c1, mbase_c2,...
}
```

```
type SubC: BaseC {                    "One of many subtypes of BaseC"
     methods msub_c1, msub_c2,...
}
```

Now suppose that we need to integrate objects derived from BaseA, BaseB, and BaseC in our application. For example, we would like to write a method that takes any object of the application and prints it on the screen. That is, we would like to create a common supertype Printable for BaseA, BaseB, and BaseC:



The three components are only superficially incompatible because the desired functionality is already present: BaseA has a print method, BaseB has two public printPart methods that together would form a valid print method, and BaseC also has a printing method, albeit with a different name (component C was bought from a German supplier). However, since the components were developed independently of each other, they were not derived from a common supertype. Therefore, the three components cannot be integrated in a straightforward way in our application.

One might argue that the integration problem would not exist if the components were "well-designed" from the start, that is, if the component designer had created the "right" type hierarchy. However, we believe it is unlikely that such perfection would be common in the component market of the future. Even if some components achieved near-perfect status over time, users discovering flaws in the type factorization would still have to wait for the component provider to release a revised hierarchy. More importantly, *different users may well have conflicting requirements*. For example, in a type hierarchy representing cars, one user may want to have subtypes representing sports cars, off-road vehicles, vans, etc., whereas another user needs subtypes for US-built cars, Japanese cars, and European cars. If the component provider attempted to reify all possible types and type factorizations, users would be overwhelmed by a myriad of types, most of which they don't need. Finally, components cannot be viewed in isolation: even if all components are internally consistent and well-designed, their combination may not be consistent [Ber90]. Programs are likely to combine many different components or component frameworks, making perfect harmony unlikely.

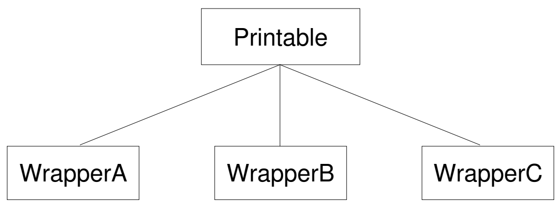
However, it should be emphasized that we do not assume complete "anarchy" or lack of any standardization. On the contrary, we assume that developers will do their very best to make their components reusable and easy to integrate with other components. That is, we assume that the components we would like to reuse are *almost* compatible—but a few details are not: a method is missing, another method takes the "wrong" type of argument, etc. We believe that such minor inconsistencies are inevitable in any large-scale component market: a component provider simply cannot foresee all possible situations in which the component could be reused.

4 Integration Using Composition

Composition is a well-known approach to solve problems similar to our example. With composition, we define three *wrappers*, each of which contains a pointer to an object of the respective base type. For example, WrapperA would contain a BaseA object:

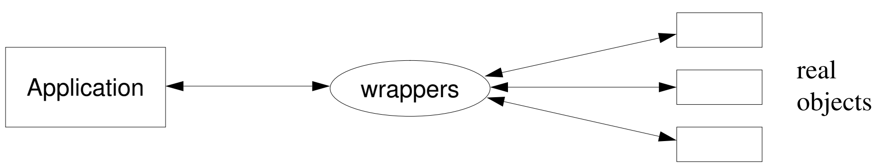
```
type WrapperA : Printable {
    a: BaseA;          "a holds the real object created by component A"
    method print();    "invokes a.print"
}
```

Instead of arranging the objects of the three components in a subtype relationship, we arrange the wrappers in the proper relationship:



Each wrapper forwards the desired messages to the actual component object. In addition, WrapperC effectively renames the drucke method by calling it from its print method. Similarly, WrapperB synthesizes a print method using printPart1 and printPart2.

In essence, wrappers isolate the core of our application from the type hierarchies of the components. Using wrappers, we have created an alternate view of the type hierarchy presented by the components—instead of dealing with BaseA objects, the application only deals with the wrappers.



4.1 Problems with Wrappers

So far, we have considered only one side of wrappers, namely the *unwrapping* process that translates wrappers into component objects (arrows going from left to right). In addition, we also have to do the *wrapping*: whenever a component function returns an object, it has to be wrapped before it is passed back to the application (arrows going from right to left). Unfortunately, as in real life, wrapping is more difficult than unwrapping.

We could create a new wrapper whenever an object needs to be wrapped, independently of whether another wrapper for that particular object already exists or not. In this "functional style" of wrapping, multiple wrapper objects can exist for a single component object, which creates a problem in languages allowing pointer equality. Pointer equality cannot be used in the presence of "functional" wrappers because two wrappers may appear to be nonequal even though they denote the same object. Therefore, wrappers are no longer transparent unless pointer equality is strictly avoided. This restriction is simple to enforce in pure object-oriented languages like SELF [US87] where equality is a user-defined operation rather than a pointer comparison, so that we can define equality of wrappers to be equality of the wrapped objects. However, pointer equality is commonly used in many other languages (e.g., C++ [ES90]). In such languages, "functional" wrappers require strict coding discipline, thereby introducing the possibility of programming errors.

Also, functional wrappers can cause performance problems because a new wrapper object is allocated for every object returned by a component. If a certain component method is called a million times, a million wrappers will be created even if the method always returns the same component object.

To avoid these problems, wrappers could be canonicalized. With canonicalized wrappers, exactly one wrapper exists per component object, and thus only the minimum number of wrappers is created. Comparing the pointers of two wrappers gives the same result as comparing the component objects themselves, so that pointer equality does not create problems. Unfortunately, canonicalization is expensive: every time an object is returned by a component method, we have to check a dictionary to see if the object already has a wrapper. If the object is new, we must create a new wrapper and enter it into the dictionary. Thus, each invocation of a component function incurs at least one dictionary lookup, an overhead that will be unacceptable for many applications. Furthermore, both kinds of wrappers could introduce additional run-time overhead because each method invocation on a component object involves at least one additional call (to invoke the wrapper method itself).

Wrappers introduce additional redundancy into the system because they duplicate part of the interfaces of the components. Therefore, if a component interface changes, additional work is required to adapt the program. While this code duplication is undesirable, it is not as bad as it seems. Much of this duplication could be automated: the programming environment could construct the wrappers automatically (with some help from the programmer in case of renaming etc.) and keep track of the relationship between the components and their wrappers. When a component interface changes, the wrapper could be updated automatically.

4.2 Even more Problems with Wrappers

Even though the performance problems caused by wrappers can be severe, wrappers could still be practical if their use could be restricted to a few small areas in the program, so that the overall performance impact would be small. Unfortunately, this is not very likely to happen: wrappers have a tendency to spread, "infecting" everything they touch. An example will illustrate this point: suppose that a method defined in component A returns a list of objects rather than a single object. In this case, we are forced to *wrap the list* in order to prevent the application from seeing the "naked" component objects contained in the list! (The list wrapper would then wrap the individual component objects whenever they are retrieved from the list.)

It would appear that we would not need a wrapper for the list if we wrapped the list elements instead. However, this is not possible in general. For example, if a returned list represents the objects in a graphical editor, removing an element from the list would have no effect on the editor if we had copied the list and created a new list of wrapped objects rather than a wrapped list. (We need to create a new list because we cannot replace the objects with their wrappers in the original list since the wrappers are not subtypes of their respective component type.)

So, even though we wanted to wrap only the objects of component A, we ended up having to create a wrapper for a generic data type, the list. Even worse, if component B also used lists we would have to create a second wrapper for lists of BaseB objects. More realistic components probably would use many other data structures, all of which may have to be wrapped as well.

In languages without delegation (i.e., most current languages), wrappers have an additional limitation: they cannot be used to override a methods of the wrapped component object. For example, suppose in our application we needed to override print in BaseA to add a few blanks before the actual printout. This will work fine if print is only called from our own code, but not if it is called indirectly (as a result of invoking some other component method). That is, any existing component method that sends print will invoke the original print method because the receiver will be a component object, not a wrapper.

By now it should have become clear that our attempt to solve the integration problem with wrappers is not entirely satisfactory. Wrappers can solve some integration problems, but using them may require a large number of wrapper types and can introduce potentially severe run-time and space overheads. Inserting wrappers into a program can be a tedious process, and programs using wrappers are often harder to understand and therefore harder to maintain.

The problems with wrappers appeared because we tried to *hide* the real objects from the application in order to solve the typing inconsistencies between the components. But introducing separate wrapper objects for the real objects forced us to maintain the new invariant that the application must never interact directly with the real objects; all interactions must go through the wrappers. Maintaining this indirection is tedious and potentially expensive in terms of execution time or space. To eliminate these problems, we somehow have to unify the wrapper with the real object. In other words, we have to change a component object's type without losing the object's identity. The next section examines several such approaches.

5 Integration Using Typing Mechanisms

Typing mechanisms can be used to view an object from different viewpoints. For example, in any language with subtyping, an object of type SubA can also be viewed as being of type BaseA. To solve the integration problem, we would like to view, for example, a SubA object as being of type Printable. This chapter discusses how typing mechanisms of existing languages can be employed to this end.

5.1 Implicit Subtyping

Implicit subtyping, as used in some programming languages (e.g., POOL-I [AL90] and Emerald [RTK91]), can be very helpful in integrating different components. With implicit subtyping, the subtyping relationships need not be declared explicitly but are inferred by the system. In our example, Printable is a valid supertype of BaseA, and the system automatically detects this relationship. Therefore, we can pass a BaseA object as the actual parameter to a method expecting a Printable object.

In essence, implicit subtyping allows us to *extend* the *explicit* type lattice (i.e., the types created and named by the programmer) after the fact, without changing the actual *implicit* lattice (formed by all possible sub- and supertypes of all types occurring in the program). BaseA and SubA are two nodes in a much larger type lattice, and implicit subtyping allows us to reify and name any node in this lattice. For example, there is a type between BaseA and SubA whose interface contains $msub_{a1}$ but not $msub_{a2}$. The original designer of component A chose not to reify this type because it did not seem useful, but for our application it might very well be needed. Similarly, Printable is an implicit supertype of BaseA which the original component designer chose not to reify. It is well known that reusable class hierarchies tend to be factored into smaller and smaller pieces with each design iteration because users discover new possibilities for reuse if certain behaviors are factored out [OO90]. Implicit subtyping allows the (re)user of a component to perform this refactoring without waiting for the component provider to do so, and without changing the component itself.

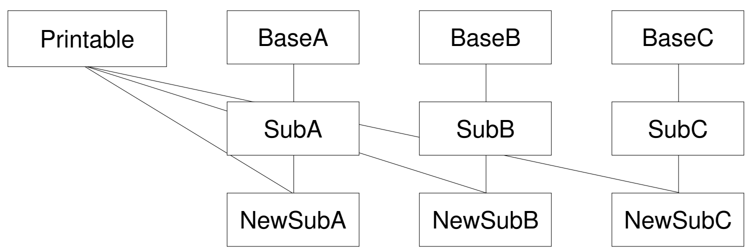
The value of not having to explicitly name every possibly useful type becomes obvious as soon as one tries to name them. For example, Johnson and Rees propose to improve reusability through fine-grain inheritance by methodically splitting up the explicit class hierarchy into small pieces so that reusers can pick the pieces from which they want to inherit [JR92]. But in the example they present, a simple List class is split up into a complex multiple-inheritance hierarchy involving 25 different classes, only one of which (List) is usually needed. Implicit subtyping allows the programmer to avoid confusing the reusers with such a myriad of types while at the same time retaining all of the flexibility.

For all its benefits, implicit subtyping is only a limited solution to the integration problem. For example, it cannot overcome problems caused by misnamed methods (e.g., drucke in BaseC) or missing methods that could be synthesized out of existing methods (e.g., print for BaseB). Furthermore, implicit subtyping is not very popular: no major object-oriented language offers it, probably because it is often perceived as weakening type checking since it may establish a subtype relationship in cases where there is no semantic relationship (for example, Cowboy isn't a subtype of Drawable even though it has a draw method [Mag91]).

Interestingly, with respect to the integration problems discussed here, dynamically-typed languages are quite similar to statically-typed languages with implicit subtyping. In dynamically-typed languages, the subtyping relationships among objects are implicit and checked lazily (and a "message not understood" error is produced at runtime if an object does not have the required type). Therefore, a type like Printable implicitly exists (though it is not reified), and the programmer can write methods that take any "printable object" as an argument. Since the types are implicit and need not be statically checkable, dynamically-typed languages have a certain advantage. For example, types can be more flexible ("if the receiver is a positive number, the argument must be printable, otherwise it can be anything, since no message is sent to it"). More importantly, it is not possible for a programmer to accidentally *overconstrain* the type of an argument (e.g., to specify Integer where Number would suffice), and thus a dynamically-typed component will probably be somewhat more reusable. However, dynamically-typed languages usually cannot solve problems caused by misnamed or missing methods.

5.2 Multiple Subtyping

Multiple subtyping can ameliorate some integration problems even if the language uses explicit rather than implicit subtyping. The idea is similar to wrappers, except that we create a new subtype for every component type instead of creating a wrapper type. The subtype *mixes in* [Moo86, BC90] the desired interface in addition to inheriting the component type's interface. In our example, the type hierarchy would look as follows:



If our application created only NewSubA objects instead of SubA objects, we would have solved the integration problem: since NewSubA is a subtype of BaseA, the objects could be passed to all component methods expecting SubA or BaseA objects, and since NewSubA…C are subtypes of Printable we could also use the objects with all application methods expecting Printable objects.

While this solution appears to work for our simplistic example, it has a serious flaw that significantly reduces its usefulness in realistic applications: it requires that we create *only* NewSubA objects, and no SubA objects. Unfortunately, we do not have control over all object creations; usually, SubA objects would be created (and returned) by functions within component A, since the component was written before NewSubA was created. These objects would not fit into our type hierarchy and therefore could not be passed to methods expecting Printable objects. In essence, we are encountering the wrapping problem again, this time using subtyping rather than composition.

If the programming language allowed explicit type coercions, we could create "type wrapper" methods that coerce the result value to NewSubA for all component methods returning SubA objects. However, we are not aware of any popular object-oriented language that would allow this. In languages offering checked type narrowing (such as Eiffel [Mey91] and BETA [KM+87]), we cannot perform the narrowing because these operations check the object's true type (the creation type). That is, the narrowing tests if the object was *created* as a NewSubA object, not whether it *conforms* to the NewSubA type. Of course, such a test would fail because the object was created with type SubA. In fact, even in a language with implicit subtyping where the narrowing mechanism checked interface conformance, it would still not be sufficient. For example, NewSubB contains the print method which is not present in SubB, and thus a coercion from SubB to NewSubB would fail. Coercion mechanisms cannot solve our problem because we need to truly *extend* both the interface and (though trivially) the implementation of an existing component object, rather than just revealing more information about an object's preexisting interface and implementation.

5.3 Factory Objects

Multiple subtyping didn't solve our problems because we had no control over the objects created by component methods. Fortunately, there is a way to solve this problem at least partially. The idea is to introduce a level of indirection at every object creation. Instead of directly creating an object ("obj = new SubA"), every object is created via a call to a factory object ("obj = factory->createSubA()"). Linton originally proposed this technique to better encapsulate the InterViews library [Lin92], but it is also very useful to solve integration problems. If the factory object is exposed to the (re)user, we can replace the standard factory with our own version creating NewSubA objects rather than SubA objects. Since the object creation types are no longer hardwired in the code, we regain control over the objects created by component methods and can substitute our slightly modified types for the original ones.

However, this solution is still far from ideal. In our example, to make components "printable" we have to introduce a large number of classes (on per concrete component class). Each of these classes must duplicate the interface of corresponding component class: although the objects now have the correct creation type, we must still write "type wrapper" methods to convert returned objects. For example, if SubA has a method get that returns a SubA object, we need to override get in NewSubA to return a NewSubA object (the method just calls the original one and coerces the result object to NewSubA). Unfortunately, changing the result type of a method is impossible in some languages (notably C++), thus crippling the factory object approach in those languages.

Besides adding a considerable amount of code, factory objects may also slow down execution. Every object creation involves an additional dynamically-dispatched call, and most method invocations on component objects also involve an additional call through the "type wrapper" method and a type test for the result coercion (assuming checked coercions). Of course, all reusable components would have to use factory objects, but we assume that component vendors would be happy to enforce this programming convention since it makes their components more reusable.

This observation leads us to a serious problem. Suppose that vendor High sells a package that adds higher-level functionality to the more basic functionality of the SubA component sold by vendor Low. To smoothly integrate the Low objects into its code, vendor High has replaced Low's factory object as suggested above. But if we also need to replace Low's factory object for our own purposes, we are stuck: how can we merge High's changes to the original factory object with our changes? The table below shows the original factory (Low's), the changed factory (High's), and the factory we'd like to use ourselves. High has changed the entry for making SubA objects to solve an internal conflict in the High library, so now it returns HighSubA objects. This creates several in-

Entry	Low's factory	High's factory	Our factory
makeBaseA	BaseA	BaseA	Printable
makeSubA	SubA	HighSubA	NewSubA

tegration problems. For example, what should the new makeSubA entry be? High expects HighSubA objects, but we need NewSubA objects so that these objects have the Printable interface. The problem is that we may not know type HighSubA since it is probably a private type that High does not wish to expose, and even if we did we may not be able to use it because it may introduce conflicts into our type hierarchy (e.g., it may define its own print method with an incompatible signature).

In general, it will be hard to use more than one component that replaces Low's factory object because there may be no factory object simultaneously satisfying all component's needs for "replacement types." Furthermore, the order of changes to factory objects must be coordinated carefully so that no component overwrites the changes made previously by another component. In other words, though factory objects can solve some simple problems, they do not appear to scale well.

5.4 Multimethods

Multimethods (also called generic methods) [DG87] are relatively new in statically-typed languages [ADL91, Cha92]. With multimethods, the message lookup may involve all arguments, not just the receiver. Since multimethods are dispatched on multiple arguments, they are not "contained" in a single class, and thus are in some sense independent of the class or type hierarchy (but see [Cha92]). We can use this independence to modify a component without changing the component itself. For example, by defining a method print specialized for arguments of type BaseB we effectively add a print method to BaseB. By defining an additional print method specialized for BaseC arguments, we have a multimethod print for all component objects, solving our problem without introducing a new Printable type.

Unfortunately, we must pay a heavy price to avoid introducing the type Printable. Because the various component objects still have no common supertype, we need to write three versions of every method that conceptually takes an argument of type Printable—one version specialized for BaseA objects, one for BaseB and one for BaseC. That is, instead of writing one method we need to write three. Even worse, if we'd like to write a method that takes *two* arguments of type Printable, we must implement *nine* methods, one for each pair in the Cartesian product of the possible actual argument types. In addition to these code duplication problems, a solution with multimethods is not as readable as we'd like because the abstraction Printable is not reified in the program, making it harder to understand the resulting code.

If it were possible to create and name types that are sets of other types or classes and to specialize multimethods on types (not classes), the combinatorial explosion of multimethods could be avoided. For example, we could create the type Printable as the set containing BaseA, BaseB, and BaseC, and specialize the print multimethod on this type. However, we are not aware of any language that allows dispatching of multimethods on types.

Finally, multimethods also could not solve our problem if the underlying language allowed the *sealing* of types [App92]. A sealed class cannot be subclassed, and a sealed type cannot be specialized further. Sealing would prevent the programmer from creating new generic methods that dispatched on a sealed class or type. Sealing has mainly been proposed for implementation reasons [App92] to allow the compiler generate better code, but also as a software engineering mechanism to prevent "internal" classes from being subclassed [KM+87].

6 Discussion

Our study has led us to a surprising result: even our very simple example creates integration problems that cannot be solved easily with the language mechanisms present in current statically-typed object-oriented languages. Both traditional reuse mechanisms, composition and subtyping, do not fare well in our scenario. Composition (the wrappers approach) needs a significant amount of extra code and can lead to potentially serious performance problems. Subtyping, whether implicit or explicit, provides only a partial solution, and multimethods introduce code duplication. In other words, current statically-typed object-oriented languages do not appear to support reuse of independently developed components well.

This result can be (mis-)interpreted in several ways:

- "Pervasive reuse is only possible if all components are standardized so no integration problems occur." We believe that this conclusion would be equivalent to concluding that pervasive reuse is impossible. The complete integration of thousands of components into a coherent framework does not appear to be practical in the near future. The standardization of software components is sometimes being compared to the standardization of components in the building industry, or to standardized interfaces between HiFi components. We believe that this analogy is flawed because such interfaces are simple, self-contained, and "flat," whereas software interfaces are much more complicated, interrelated, and "deep." For example, a software component may be unusable if it is embedded in the "wrong" type hierarchy even though it offers the correct "flat" interface. Of course, this does not mean that standardization attempts are futile or undesirable, but such attempts are likely to be time-consuming and to eliminate only some (but not all) integration problems. If pervasive reuse of software is to become reality, we must find a more practical solution to the integration problem.
- "Editing the source code of components isn't so bad; after all, it's the only practical way to achieve pervasive reuse." This "pragmatic" interpretation contradicts one of our basic assumptions, and we feel that it is not justified. While editing the source code of components can solve the problems presented in this paper, it requires access to source code and produces a major software maintenance problem (tracking new revisions of the component). One of the goals of reuse is to reduce the cost of software development, and introducing such maintenance problems does not seem compatible with that goal.

- “Programs should be compact. Integration problems appear *between* (not within) programs, and such problems should be solved with systems integration languages, not programming languages.” It is undoubtedly true that integration problems also occur on the inter-application level. However, if we cannot solve the integration problems *within* programs, overall productivity is unlikely to improve very much since most of the development effort lies in writing applications rather than in systems integration. Furthermore, if the inter-application interface is statically typed as well, application reusers will face the same problems that component reusers face—pushing the problem to a higher level doesn’t make it disappear. However, application interfaces will probably be more standardized since they are coarser and fewer. Also, wrappers become more viable since the inter-application communication bandwidth is likely to be lower than the inter-component bandwidth so that the performance problems caused by wrappers are less severe.

We believe there is no single cause for the integration difficulties encountered with current languages and systems. However, the first and foremost culprit may be that the integration problem often is not acknowledged at all. Programming language designers seem concerned mostly with possibly large but well-integrated single applications rather than with applications composed out of a multitude of preexisting components. That is, they assume that component interfaces are well-coordinated and can be adjusted if necessary.

Another problem is that the traditional view of reuse overemphasizes *compiled-code reuse* (“Look, Ma, no changes!”), leading to several problems that can compromise widespread reuse. The unwillingness to recompile code can lead to rigid, less adaptable components and precludes many approaches that could modify a component without relying on internal implementation details. Because there is only one version of the object code, programmers face a dilemma between flexibility and performance. If the object code is kept flexible, it cannot be optimized as much; if it is highly optimized, flexibility suffers because many design parameters are hardwired in the code. For example, current compiled-code-oriented systems often use impure interfaces that expose implementation information in order to obtain better performance, thereby severely compromising reusability [HU92].

We wish to emphasize reuse in a broader sense, where reuse equals saved programming effort. Programming time is much more important than compilation time because the computational power needed for recompilation becomes cheaper and cheaper every year whereas the programmer’s time does not.† Of course, reuse mechanisms must be practical in terms of the machine resources they require. But the overriding goal of reuse should be to increase programmer productivity, and we should not constrain our search for better reuse mechanisms *a priori* to mechanisms that preserve compiled-code reuse (see [Szy92] for a similar argument).

---

† This is not to say that compilation time isn’t an important factor influencing programming productivity. But even with today’s hardware, a well-engineered compiler can compile several thousand lines of code per second [Tem90]. The problem is that most compilers are not tuned for compilation speed.

To summarize, we believe that a programming language / environment combination needs to provide the following functionality to be successful in a world of pervasive reuse:

- Component types and type hierarchies should be changeable within certain bounds so that component reusers can integrate one component with another. That is, it should be possible for the reuser to change the component types directly, without first introducing new subtypes or subclasses.
- This flexibility must be provided without requiring source code access to the component's implementation, and it should not compromise the efficiency of the resulting programs.

How this functionality is best provided, and whether it is provided directly by the programming language (type system) or by the programming environment, remains an open question.

7 Towards a solution

In view of this broader concept of reuse, this section discusses several possible approaches that promise to overcome the integration problem and to increase reuse.

7.1 Type Adaptation

A straightforward approach, which we call *type adaptation*, is to allow the programmer to change given interfaces in a restricted way *after* they have been delivered as a component. The interface changes must not invalidate the component's type structure (e.g., the subtype relationships must be preserved), but this restriction still allows many useful adaptations:

- New types can be added to the type hierarchy (for example, a common supertype Printable). As with implicit subtyping, this allows the programmer to reify types that are implicitly already present in the component's type lattice.
- Types can be complemented with new methods (e.g., a print method for BaseB calling printPartial1 and printPartial2) and new instance variables. Of course, the added methods do not have access to any internal implementation details of the component, and they may not conflict with existing public methods.[†] (In a language separating types and classes, complementing a type means adding the methods to all classes that implement the type.)
- Operations can be renamed to make the component compatible with others (in our example, drucke could be renamed to print).

[†] This also requires that the public (external) and private (internal) name spaces are kept separate. Then, adding a public method print does not create a name conflict even if the component's implementation already has a private method print. The two print methods are completely separate; no method added through type adaptation could call (or even name) the private print method, and no method in the component's implementation could call the new public print method (which didn't exist when the component implementation was written).

Since type adaptation augments types "in place," it causes none of the problems associated with wrappers or subtyping. No additional objects or type hierarchies are created, and all objects created within the component automatically have the augmented type (and no object has the original unchanged type). Furthermore, since the changes do not depend on internal component details (and vice versa), the component implementation could be replaced by the vendor with a newer version without causing adaptation problems for customers using type adaptation. Similarly, because the component's implicit type hierarchy does not change, the component implementation need not be re-typechecked after the changes. A variant of type adaptation would also be useful for dynamically-typed languages, where it would adapt objects (e.g., classes or prototypes) rather than types.

Adapting a component interface does not require access to the source code of the implementation. For example, a component could be distributed in an architecture-neutral intermediate format (such as ANDF [OSF91]) that contains all type information needed by a type adaptation system (but no source code) and could be translated into optimized machine code. In this intermediate form, the component's implementation need not hardwire the exact storage layout or dispatching mechanisms like machine code would. For example, introducing additional supertypes may change the optimal message dispatching strategy relative to the original component and would therefore be hard to implement if components were delivered as object code. With an intermediate format, the optimal dispatching strategy could still be used because machine-code generation is deferred until reuse time. Thus, we could eliminate the dilemma between flexibility and efficiency because the machine code can be customized for each application.

A more detailed discussion of the implementation of such an intermediate format is beyond the scope of this paper. More research is probably needed to investigate its practicality, but it appears that the implementation could leverage off already existing intermediate formats aimed at cross-architecture portability, such as the commercial ANDF format mentioned above.

7.2 Extension Hierarchies

Ossher and Harrison have proposed a mechanism called *extension hierarchies* [OH92] that is similar to type adaptation. Even though their description emphasizes extension rather than adaptation, extension hierarchies seem well suited to solving the integration problem. The basic idea is to combine a base hierarchy (= component) with sparse extension hierarchies containing changes, additions, and deletions. Extensions are modelled as operators and can be combined to form new extensions; both sequential combination (applying one extension, then another) and parallel combination (merging two extensions) are supported. Extension hierarchies are more general than type adaptation since they allow arbitrary changes that are not necessarily interface-preserving, and because they can be used to extend implementations as well as interfaces. Also, extension hierarchies are a programming environment mechanism rather than a language mechanism.

We believe that extension hierarchies hold much promise. While we do not agree with all aspects of the particular variation proposed in [OH92] (for example, the authors strictly avoid recompiling code after extensions are applied), extension hierarchies can solve many integration problems in a straightforward way. In our example, we could define an extension for each component to insert the Printable supertype. Since the extensions are kept separate from the base component, they can be reapplied automatically to newer versions of the components (as long as the component's interfaces do not change).

In general, extension hierarchies require full source-code access to apply an extension. To integrate a component, a programmer would change it, and the changes form the extension. Later, when a newer version of the component is delivered (ideally in the form of an extension to the previous version), the two extensions would be combined to form the new integrated component. However, the two extensions may conflict, and the conflicts will have to be resolved by the programmer, creating the maintenance problem we wished to avoid with our "no source changes to reused components" policy. That is, if extension hierarchies are used in their full generality, they can create problems similar to conventional change management systems. Therefore, we believe that an extension hierarchy system would have to be used in a restricted way to solve integration problems, so that extensions would only perform the modifications allowed by type adaptation. The system would probably contain an automatic "extension checker" to verify that an extension conforms to the rules of type adaptation. Passing this check would guarantee that an extension could be combined without conflicts with "update extensions" provided by the component's manufacturer, as long as the package's public interface is not changed by the manufacturer. In essence, the extension checker would thus represent a type adaptation subsystem implemented within an extension hierarchies system.

7.3 Other Related Work

A mechanism very similar to type adaptation (called *enhancive types*) was proposed by Horn [Hor87]. However, unlike type adaptation, enhancive types do not change types in place but instead allow a base type to be coerced into another type (the enhanced type) that offers additional operations implemented in terms of the base type's public methods. Horn's paper also discusses in more detail why the original methods of an enhanced type need not be re-typechecked. Unfortunately, the paper was couched in theoretical terms as an extension to constrained genericity, and the idea has largely been overlooked in programming language design.

Sandberg's *descriptive classes* [San86] are the first example of types that could be declared after the classes that were their subtypes. The usefulness of creating new superclasses for existing classes was discussed in detail by Pedersen [Ped89] and implemented in Cecil by Chambers [Cha93a]. *Predicate classes* [Cha93b] offer yet another way to extend objects, although this isn't their intended use. Opdyke [Opd92] defines a set of program restructuring operations (refactorings) that support the design, evolution and reuse of object-oriented application frameworks. Although Opdyke discusses the refactorings in terms of source code changes, most of them could be performed on a sourceless intermediate code format. In such a system, the refactorings would be very similar to the modifications allowed by type adaptation.

Palsberg and Schwartzbach have recognized a similar problem with traditional reuse mechanisms and have proposed a type substitution mechanism aimed at maximizing code reuse [PS90]. However, for the scenario outlined here, their solution is both too general and too restricted: too general because in the presence of subtype polymorphism it may require the re-typechecking of a component's implementation, and too restricted because it does not allow adding new methods or renaming methods. Thus, a system using type substitution could only partially solve the integration problems we encountered in this paper.

The Mjølner BETA fragment system [MPN89] is a grammar-based programming environment designed to customize and assemble program fragments. A fragment is any sequence of terminal and nonterminal symbols derived from a nonterminal symbol. Fragments may define interfaces or implementations, and in principle one interface fragment could have several implementation fragments. Most importantly, fragments can be plugged into empty "slots" of other fragments, providing for a very flexible program integration system. While the mechanisms of post-facto type adaptation and the fragment system are closely related, the underlying design and reuse philosophy of BETA / Mjølner is very different from ours since it is intended for *planned* reuse only. Extending the fragment system to incorporate type adaptation would be a promising area for future research.

The prototype of the Jade programming environment for Emerald described by Raj [RL89] is similar to BETA's fragment system. A piece of code can be left partially unspecified, and the code piece's *habitat* describes the required interface of the unspecified parts. Jade is interesting because the underlying programming language, Emerald, does not provide inheritance and thus does not by itself support code reuse. That is, the Emerald / Jade combination treats reuse at the level of the programming environment rather than at the language level. Like BETA / Mjølner, Jade emphasizes planned reuse only and therefore does not directly address the integration problem.

8 Conclusions

We have examined problems likely to occur in a world where reuse is pervasive and components are designed by independent organizations. In such a world, a large part of every application would consist of reusable components, and applications would combine many different components or frameworks. The interfaces and type hierarchies of components must be expected to be slightly inconsistent in relation to each other (even if the individual components are perfectly self-consistent) because the sheer number of different components would make perfect coordination impractical.

Today's programming languages and environments rely on components to be well-integrated and do not handle the integration of independently developed components well. Even slight inconsistencies can lead to integration problems that cannot be handled satisfactorily with the reuse mechanisms available in current languages. Wrappers create redundant type hierarchies and make programs harder to understand and change, and the wrapping and unwrapping of objects can incur performance problems. Typing mechanisms usually cannot solve the problems because existing component types and type hierarchies cannot be adapted; factory objects combined with multiple subtyping can solve simple problems but do not scale well. A language with multiple dispatch on argument types (not classes) could solve most adaptation problems at the expense of language complexity, but no current object-oriented language offers the combination of these features. Even dynamically-typed languages suffer from adaptation problems, although they are usually somewhat better off than their statically-typed counterparts.

We believe that it is helpful to adopt a broader view of reuse mechanisms that does not center around perfectly coordinated components and pure compiled-code reuse. Future programming language / environment combinations should allow the reuser to change component types and type hierarchies *in place* to integrate components with each other; component interfaces should not be completely frozen as they are today. Component implementations may best be delivered in an intermediate format so that the new flexibility can be combined with efficient execution.

We have proposed a new reuse mechanism, *type adaptation*, that is based on previous ideas by Horn, Sandberg, and Pedersen. It can adapt a component in restricted ways without requiring access to the component's implementation and without retypechecking it. With mechanisms such as type adaptation or Ossher and Harrison's extension hierarchies, we may come one step closer to reaching the dream of pervasive reuse.

Acknowledgments: I would like to thank David Ungar for his continuous guidance and support; I am also very grateful for the support provided by Sun Microsystems Laboratories. Ole Agesen, Lars Bak, Bay-Wei Chang, David Cheriton, Craig Chambers, John Maloney, Jens Palsberg, Clemens Szypersky, and the anonymous reviewers provided valuable comments on earlier drafts of this paper. Many thanks also go to Peter Kessler and Alan Snyder for discussions on the subject of integration.

References

[ADL91] Rakesh Agrawal, Linda G. DeMichiel, and Bruce G. Lindsay. Static Type-Checking of Multi-Methods. In *OOPSLA '91 Conference Proceedings*, pp. 113-128, Phoenix, AZ, October 1991. Published as *SIGPLAN Notices 26(11),* November 1991.

[AL90] Pierre America and Frank van der Linden. A Parallel Object-Oriented Language with Inheritance and Subtyping. In *ECOOP/OOPSLA '90 Conference Proceedings*, pp. 161-168, Ottawa, Canada, October 1990. Published as *SIGPLAN Notices 25(10)*, October 1990.

[App92] Apple Computer, Eastern Research and Technology. *Dylan, an object-oriented dynamic language*. Apple Computer, Cupertino, CA, April 1992.

[BC90] Gilad Bracha and William Cook. Mixin-Based Inheritance. In *ECOOP/OOPSLA '90 Conference Proceedings*, pp. 303-311, Ottawa, Canada, October 1990. Published as *SIGPLAN Notices 25(10)*, October 1990.

[Ber90] Lucy Berlin. When Objects Collide: Experiences with Reusing Multiple Class Hierarchies. In *ECOOP/OOPSLA '90 Conference Proceedings*, pp. 181-193, Ottawa, Canada, October 1990. Published as *SIGPLAN Notices 25(10)*, October 1990.

[CW85] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *Computing Surveys 17(4)*, pp, 471-522, December 1985.

[Cha92] Craig Chambers. Object-Oriented Multimethods in Cecil. In *ECOOP '92 Proceedings*, pp. 33-65, Utrecht, The Netherlands, June 1992. Published as *Springer Verlag LNCS 615*, Berlin, Germany 1992.

[Cha93a] Craig Chambers. *The Cecil Language—Specification and Rationale.* Technical Report 93-03-05, Computer Science Department, University of Washington, Seattle 1993.

[Cha93b] Craig Chambers. Predicate Classes. In *ECOOP '93 Conference Proceedings*, Kaiserslautern, Germany, July 1993.

[CC+89] Peter Canning, William Cook, Walter Hill, and Walter Olthoff. Interfaces in strongly-typed object-oriented programming. In *OOPSLA '89 Conference Proceedings*, pp. 457-468, New Orleans, LA, October 1989. Published as *SIGPLAN Notices 24(10)*, October 1989.

[Cox86] Brad Cox. *Object-Oriented Programming: An Evolutionary Approach*. Addison-Wesley, Reading, MA 1986.

[DG87] Linda G. DeMichiel and Richard P. Gabriel. The Common Lisp Object System: An Overview. In *ECOOP '87 Conference Proceedings,* pp. 223-233, Paris, France, June 1987. Published as *Springer Verlag LNCS 276*, Berlin, Germany 1987.

[Deu83] L. Peter Deutsch. Reusability in the Smalltalk-80 Programming System. Proceedings of the *Workshop on Reusability in Programming*, p. 72-76. Newport, RI, September 1983.

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison Wesley, Reading, Ma 1990.

[Hor87] Chris Horn. Conformance, Genericity, Inheritance and Enhancement. In *ECOOP '87 Conference Proceedings,* pp. 223-233, Paris, France, June 1987. Published as *Springer Verlag LNCS 276*, Berlin, Germany 1987.

[HU92] Urs Hölzle and David Ungar. The Case for Pure Object-Oriented Languages. In *Proceedings of the OOPSLA '92 Workshop on Object-Oriented Languages: The Next Generation.* Vancouver, Canada, October 1992.

[JR92] Paul Johnson and Ceri Rees. Reusability through Fine-grain Inheritance. *Software—Practice and Experience 22(12)*, pp. 1049-1068, December 1992.

[KM+87] B. B. Kristensen, O. L. Madsen, B. Møller-Pedersen and K. Nygaard. The BETA Programming Language. In B. Shriver and P. Wegner (eds.), *Research Directions in Object-Oriented Programming*, pp. 7-48. MIT Press, Cambridge, MA 1987.

[Lie88] Henry Lieberman. Position Statement in the Panel on Varieties of Inheritance. In *Addendum to the OOPSLA '87 Proceedings*, p. 35. Published as *SIGPLAN Notices 23(5)*, May 1988.

[LVC89] Mark A. Linton, John Vlissides, and Paul Calder. Composing user interfaces with Interviews. *IEEE Computer Magazine*, February 1989.

[Lin92] Mark A. Linton. Encapsulating a C++ Library. *Proceedings of the 1992 Usenix C++ Conference*, pp. 57-66, Portland, OR, August 1992.

[Mag91] Boris Magnusson. Position statement during the ECOOP '91 Workshop on Types, Geneva, Switzerland, July 1991.

[MPN89] Ole Lehrmann-Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *The BETA Programming Language—A Scandinavian Approach to Object-Oriented Programming.* OOPSLA '89 Tutorial Notes, New Orleans, LA, October 1989.

[Moo86] David A. Moon. Object-Oriented programming with Flavors. In *OOPSLA '86 Conference Proceedings*, pp. 1-8, Portland, OR, October 1986. Published as *SIGPLAN Notices 21(11)*, November 1986.

[Mey91] Bertrand Meyer. *Eiffel—The Language*. Prentice Hall, New York 1991.

[OH92] Harold Ossher and William Harrison. Combination of Inheritance Hierarchies. In *OOPSLA '92 Conference Proceedings*, pp. 25-43, Vancouver, Canada, October 1992. Published as *SIGPLAN Notices 27(10)*, October 1992.

[OO88] Panel: Experiences with reusability. In *OOPSLA '88 Conference Proceedings*, pp. 371-376, San Diego, CA, September 1988. Published as *SIGPLAN Notices 23(11)*, November 1988.

[OO90] Panel: Designing Reusable Designs: Experiences Designing Object-Oriented Frameworks. In *Addendum to the OOPSLA/ECOOP '90 Conference Proceedings*, pp. 19-24, Ottawa, Canada, October 1990.

[Opd92] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. Ph. D. Thesis, Department of Computer Science, University of Illinois, Urbana-Champaign 1992. Published as Technical Report UIUCDCS-R-92-53097.

[OSF91] Open Systems Foundation. *OSF Architecture-Neutral Distribution Format Rationale*. Open Systems Foundation, June 1991.

[Par72] David Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM, 15(12)*, December 1972.

[Ped89] Claus H. Pedersen. Extending ordinary inheritance schemes to include generalization. In *OOPSLA '89 Conference Proceedings*, pp. 407-417, New Orleans, LA, October 1989. Published as *SIGPLAN Notices 24(10)*, October 1989.

[PS90] Jens Palsberg and Michael Schwartzbach. *Type substitution for object-oriented programming*. In *ECOOP/OOPSLA '90 Conference Proceedings*, pp. 151-160, Ottawa, Canada, October 1990. Published as *SIGPLAN Notices 25(10)*, October 1990.

[RL89] Rajendra K. Raj and Henry K. Levy. A Compositional Model for Software Reuse. *Computer Journal 32(4),* pp. 312-322, 1989.

[RTK91] Rajendra K. Raj, Ewan Tempero, and Henry K. Levy. Emerald: A General-Purpose Programming Language. *Software—Practice and Experience 21(1)*, pp. 91-118, January 1991.

[San86] David Sandberg. An Alternative to Subclassing. In *OOPSLA ‘86 Conference Proceedings*, pp. 424-428, Portland, OR, October 1986. Published as *SIGPLAN Notices 21(11)*, November 1986.

[Szy92] Clemens Szypersky. Extensible Object-Orientation. In *Proceedings of the OOPSLA ‘92 Workshop on Object-Oriented Languages: The Next Generation.* Vancouver, Canada, October 18, 1992.

[Tem90] Josef Templ. Compilation Speed of the SPARC Oberon Compiler. Personal communication, April 1990.

[US87] David Ungar and Randall B. Smith. SELF—The Power of Simplicity. In *OOPSLA ‘87 Conference Proceedings*, pp. 227-242, Orlando, FL, October 1987. Published as *SIGPLAN Notices 22(12)*, December 1987.

[WGM88] André Weinand, Erich Gamma, and Robert Marty. ET++—An Object-Oriented Application Framework in C++. In *OOPSLA ‘88 Conference Proceedings*, pp. 168-182, San Diego, CA, October 1988. Published as *SIGPLAN Notices 23(11)*, November 1988.