VICTORIA UNIVERSITY OF WELLINGTON

Department of Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 471 5328
Fax: +64 4 495 5232
Internet: Tech.Reports@comp.vuw.ac.nz

Inheritance and Reusability

Robert Biddle and Ewan Tempero

Technical Report CS-TR-95/8
May 9, 1995

Abstract

Object-oriented programming has been widely acclaimed as the technology that will support the creation of *reusable software*, particularly because of the "inheritance" facility. However, the limited practical impact has not justified the acclaim. We believe this lack of success is due not any particular shortcoming in inheritance, but to widespread misunderstanding of the way the technology supports reusability. In this paper, we explore the connection between inheritance and reusability. We show that inheritance *does* support reusability, but does so in ways that are not well understood.

1 Introduction

Object-oriented programming (OOP) has been widely acclaimed as the technology that will support the creation of *reusable software* [Fis87, Mey87]. Recently, however, some commentaries suggest OOP has not lived up to expectations [Ude94]. It is true that the amount of reusable software is not as vast as the marketing hyperbole associated with the object-oriented bandwagon led us to expect. So what has gone wrong? We believe this lack of success is due not any particular shortcoming in OOP, but to widespread misunderstanding of the way the technology supports reusability.

There are many definitions of what software reuse means, but most involve some variation of "the use of engineering knowledge or artifacts from existing systems to build new ones" [FI94]. This definition encompasses many diverse areas, from application generators, to domain analysis, to design patterns, to classification and search, to management [Sof94, BP89]. Our interest is in the use of existing code. In particular, we want to establish precisely how different language features impact the reusability of code. As Meyer has said: "...any acceptable solution must in the end be expressible in terms of programs, and programming languages fundamentally shape the software designers' way of thinking" [Mey86]. We have been looking at this issue from the point of view of specific languages and specific features [ABT94, BTA95].

The frequently cited reason for the acclaim given to OOP is the facility generally known as "inheritance". If OOP has not succeeded, then inheritance deserves to be examined. In this paper, we explore the connection between inheritance and reusability. We show that inheritance *does* support reusability, but does so in ways that are not well understood, and not in the ways usually claimed.

We begin with a discussion of what it means for software to be reusable. While reusability is an understood aim in programming language design, it is seldom discussed directly; we have found that discussing it directly helps our analysis. In the next section, we discuss the important features of OOP. We regard OOP as "design by analogy", which we believe offers a concise, consistent, and hype-free description of the object-oriented paradigm, and provides a foundation for understanding the support of reusability. Inheritance appears in many forms. While our discussion applies to most forms, to avoid confusion we briefly outline in section 4 the form we use here. We then look at class reusability *without* using inheritance. This provides a benchmark with which we can establish the benefits of inheritance in section 4. The remaining sections discuss how various uses of inheritance impact the reusability of code. Finally, in section 9 we present our conclusions.

We use Eiffel as our notation for presenting examples [Mey91], however our discussion readily transposes to other languages.

2 Reusability

We believe the first step to a better understanding of this topic is recognising the distinction between *software reuse* and *reusable software*. Software reuse is an activity that takes place *afterward*, when software was initially created in the past. To best support this, reusable software must be created *beforehand* in such a way that it is later easy to reuse.

There is an analogous situation in the wider world. In recent years people have become concerned about waste and disposal of manufactured goods. The concern is primarily with *recycling*, and when good have been manufactured and then used, it is the activity that takes place afterward. However, recycling is not always easy or cheap: many goods are difficult to recycle. For example, components are sometimes difficult to separate, and materials are often hard to identify. As a result, it has been realised that in order to best facilitate recycling, goods must be created to be *recyclable*, which is something that must be planned beforehand. For

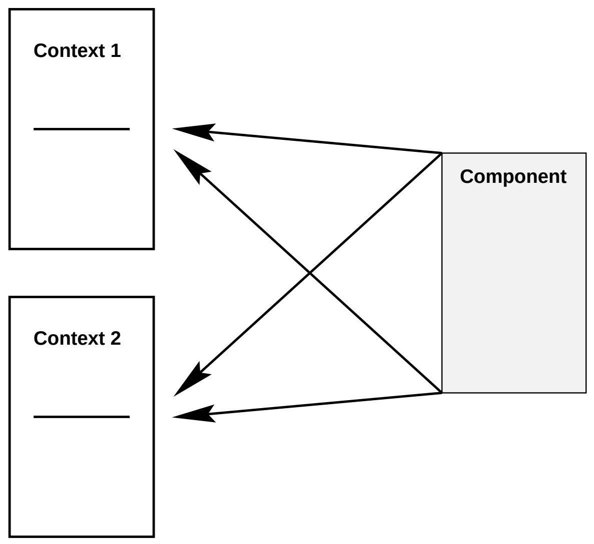

Figure 1: *Several contexts may invoke a reusable component. The key issues are the generality of the component, the flexibility of the interface, and the safety of the connection.*

example, components may be designed to facilitate separation, and materials can be clearly labeled.

Many of the discussions on reuse and reusability tend to blur the distinction, often using both terms in the same sentence as if they were synonymous. Although such discussions do include comments on how code can be made more reusable, they tend to focus on how language features help reuse code (for example, [Mey87, Mic88]). As a consequence, how the features impact the reusability of code tends to be overlooked.

There are two reasons why this confusion between reuse and reusability exists. The first lies in the fact that, no matter how reusable the software is, there has to be some mechanism that allows it to be reused. Thus the tendency is to think that it is the reuse mechanism that is making the code more reusable. It is true that without that mechanism, the code could not be reused. However, even with that mechanism, code may not be reusable because of other factors. The second reason for the confusion is that the proof of the reusability of software involves reuse. Thus any discussion of reusability (and ours is no exception) must necessarily talk about code reuse.

Although the distinction between software reuse and reusable software is often confused, it is generally accepted that effort must be made in the creation of code if it is to be made easier to reuse. This is generally referred to as "design for reuse". Such discussions tend to focus on general advice and heuristics for making classes more reusable[JF88] or higher-level class organizations that have been found to have been useful in the past [GHJV95]. While these issues are important for the creation of reusable code, in our work we are concerned with how programming languages support the creation of reusable code.

Figure 1 shows the kind of situation we want to create. In the figure, a shaded box represents the code that we want to reuse, the *component*, and an unshaded box represents the use of that code, the *context*. For our purpose, reusable software consists of components that can be successfully used by all relevant contexts without change and with the minimum of effort. We are not concerned with components that require modification with a text editor, nor with multi-purpose components that provide many services of which only a few are ever used at one time. There are also other reuse issues not directly associated with the creation of reusable software, such as finding the components, understanding what the components do, and so on. While these issues are very important — indeed they ultimately may be what dictates the success of reusable software — they are not our concern in this paper.

3 Object Oriented Programming and Reusability

To best understand how OOP involves reusability, it is important to understand just what OOP is all about. In fact, we believe there are two elements in the foundations of OOP, one technical and the other more philosophical. Both have significant practical implications.

The more technical foundation of OOP concerns an evolutionary process in program design and programming language support. From a historical perspective, OOP may be seen as the confluence of various threads of development in program design: some involving the design of control, others involving the organisation of data.

The more philosophical foundation of OOP goes deeper, and directly addresses one of the most fundamental issues in programming: organising complexity. There are two key ideas involved. The first idea is that we should design programs *by analogy*. That is, we should organise our programs using structures like the structures in the application domain. The second idea is that most application domains, like the real world, are commonly regarded as structured as *entities* and *actions* on entities.

In the world of entities and actions, it is typical that both entities and actions are involved in many different contexts. Entities are used for various purposes, and the same action may be applied to various entities. Accordingly, when we design by analogy, we are likely to use structures that either have already been needed and designed before, or that will needed in the future. Either way, there are obvious advantages of reusability.

The more technical foundations of OOP also support reusability, particularly via encapsulation. Encapsulation means that the object is a capsule, whereby the behaviour and state are bound together as a unit, with the only access being the designated operations. For the object to be reusable in different (perhaps as yet unknown) contexts, it is necessary to make certain that the behaviour cannot be interfered with. Moreover, it might be useful to use the context code together with a different implementation of the object, then the reusability of the context code is of interest. Accordingly it is important that the context code cannot depend upon the implementation of the object at all, and this can be enforced by encapsulation.

3.1 Classes

It is a big organisational step from seeing world as consisting of entities to seeing the world as classes of entities; it can also be a problematic step. Every thing is distinct, but we consider many things to have such similar operations and behaviour we regard them as mere instances of the the larger class. In the real world, however, there are many ways to classify things, and the ways are not always consistent. Some OOP languages allow dynamic organisation of objects into classes; others require the programmer to deal will classification conflicts at the design stage, and make classes explicit in the program — this typically allows more efficient implementation. Whichever approach is taken, there are strong organisational reasons to structure objects into classes: it fosters reusability, because code can be written for any instance of a class, rather then for one particular object.

3.2 Composition

Perhaps the most common way that two different entities can be statically structured together is by *composition*: where one object is used as a component part of the other. (This is called "aggregation" by some writers, for example Booch [Boo94].) This is common in the real world,

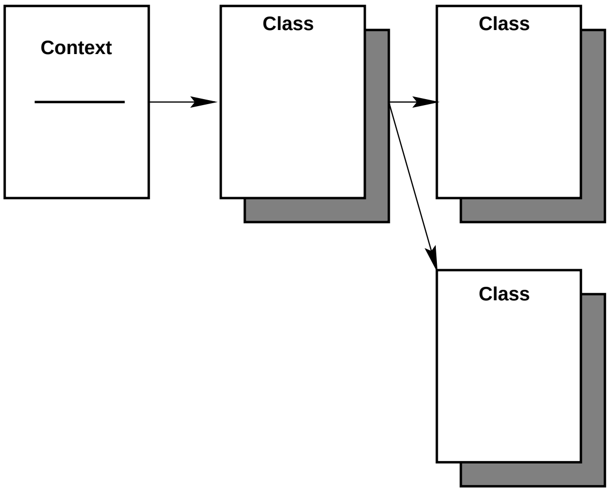

Figure 2: *Objects or classes may be implemented by using other objects or classes via composition. Not only does this approach allow component objects or classes to be reusable, but also allows the composite class implementation to be general by using the components. (The interface/implementation distinction is shown here by the interface box overlapping the grey implementation box.)*

where technology and nature build larger things using smaller things. We model this in programs, and so OOP languages typically allow the construction of objects by implementing them using several other objects. This facility thus makes further use of the reusability of objects and classes, especially because a wide variety of larger kinds of objects can be constructed using other classes in various ways. Moreover, the end result is the creation of more objects and classes, allowing further reusability still.

4 Inheritance

One of the ways we deal with the real world is by organising objects with similar operations and behaviour into classes. As discussed above, simple classification cannot always adequately represent the necessary complexities. One such situation occurs when objects represent different specializations of some general concept: at the general level they belong to the same class, however at the specialized level they are different. In the real world this leads to taxonomical hierarchies, with a general category at the top, which is then divided and sub-divided into increasingly specialised cases. It is these taxonomical hierarchies that are typically represented using inheritance.

The main idea is that an object of a more specialised class inherits the characteristics of a more basic class. For example, consider a business that has a number of different kinds of customers, such as those that pay cash and subscribers that have an account. Both kinds of customers have names and delivery addresses, but subscribers also have a billing address. The Eiffel implementation of this is below:[1]

[1] In Eiffel, classes are made up of *features*, some which may be data or *attributes*, and some which may be code or *routines*. Features with the {NONE} annotation are private to the class.

```
class Customer
feature {NONE} -- representation
  name: string;
  address: string;
feature -- interface
  delivery_label is do
    io.putstring(name);
    io.newline;
    io.putstring(address);
  end
end

class Subscriber -- inheritance version
inherit Customer
feature {NONE} -- representation
 mail_address: string;
feature -- interface
 billing_label is do
  io.putstring(name);
  io.newline;
  io.putstring(mail_address);
 end
end
```

In this example, the features of the parent class, `Customer`, are inherited by the child class, `Subscriber`.

There are a number of issues associated with providing inheritance in a programming language. These include how the child class overrides a feature from its parent, how many parents a child may have, or how multiple features with the same name are resolved. While how these issues are dealt with may impact the reusability of code, we will restrict our discussion to just the overall impact of inheritance on reusability.

In the example above, `Customer` is reused by `Subscriber`. It is this reuse that is usually offered as evidence that inheritance makes code reusable. The argument is that, while the parent class has been augmented, it has been done so without being affected itself and without code being replicated. However, while it is clear that code is being reused, it is not clear whether the code is more reusable, or whether the increase in reusability is solely due to the fact that inheritance is a reuse mechanism.

5 Reuse without Inheritance

In order to understand what the benefits of inheritance really are, it is instructive to determine what can be done without it. Given the prominence that inheritance has with regard to code reuse, there is a tendency to think that classes cannot be reused without it. However this is not the case. The example below demonstrates the reuse of the `Customer` class via composition.

```
class Transaction
feature {NONE} -- representation
  buyer: Customer;
  item: Article;
feature -- interface
  tax is do
  ...
  end
end
```

None of the code associated with Customer has to be rewritten in creating the Transaction class.

This same technique can be used in situations such as the Customer/Subscriber example used above. Consider:

```
class Subscriber -- composition version
feature {NONE} -- representation
  parent: Customer;
  mail_address: string;
feature -- interface
  billing_label is do
    io.putstring(parent.name);
    io.newline;
    io.putstring(parent.mail_address);
  end
  delivery_label is do -- wrapper function
    parent.delivery_label;
  end
end
```

Here the "parent" is a component of the "child", with the interface to the parent reflected in the child through the use of "wrapper" functions. If the Customer class represented a substantial effort in software production, then this effort is clearly not being repeated in the creation of Subscriber. Notice that this technique also allows the redefinition of a Customer operation for the Subscriber class, it easily allows the suppression of an operation, and it will even do the equivalent of multiple inheritance.

We can see the benefits provided by inheritance by examining the composition version of Subscriber. The main advantage is not having to write the wrapper functions, with a secondary advantage that only one subroutine call is needed (although modern compiler technology should be able to deal with this). There is a further maintenance advantage: if a new routine is added to Customer, then a new wrapper function must explicitly be added to the composition version of Subscriber. Notice that if the change to Customer does not change its interface, Subscriber will still benefit from the change. However notice also that if Subscriber does not need to track the Customer interface, then no change is needed to Subscriber.

The reason why Subscriber may want to track the Customer interface is for polymorphic substitution, that is, to maintain the *isa* relationship between the two classes (sometimes referred to as "inheritance of interface"). This relationship allows the following assignment to be legal.

```
Customer c;
Subscriber s; -- composition version
c := s;
```

However, there is no such relation between the two types, and consequently such assignments are considered illegal.

Inheritance is often advocated because it can be used to provide polymorphism. In fact, polymorphism is not inherently part of inheritance. The illegal assignment above is due to the tie between types and classes in Eiffel, which restricts a *subtype* of a class to only those classes that are descendents of the class. The same restriction holds in other static type systems that use classes as types (such as C++[Str91]). In a dynamically typed language, such as Smalltalk, if `Subscriber` was created using composition, then, using wrapper methods, instances of it could be made to respond to the same messages (have the same protocol) as instances of `Customer`, and so could be substituted wherever `Customer` instances are expected. Nor is this restriction one that is due to static type systems. Emerald [RTL+91] is a statically-typed object-based language but the relationship between types is not static. Thus, the composition version of `Subscriber` written in Emerald could be substituted where `Customer` is expected, and this substitution could be verified as valid at compile time. We are not saying that associating types with classes is wrong, however the consequences of such a relationship must be understood if the full reusability benefits of inheritance are to be exploited.

A final comment about providing inheritance through composition is that, through this technique, the "child" does not have access to private features of the "parent". Most forms of inheritance allow this kind of access, however some regard the violation of encapsulation problematic [Sny87].

We conclude that, from the reusability point of view, inheritance removes the need for wrapper functions, and is one way to provide polymorphism. We will discuss the relationship between polymorphism and reusability in section 7.

6 Implementation Inheritance

Inheritance is usually described as a method for classification, and as a way to represent the specialisation relationship between classes. However a common use (some would say mis-use) by novices of inheritance is as an alternative to composition. Returning to our customer example, the `Transaction` class could be implemented as:

```
class Transaction
inherit Customer;
...
end
```

Given that there is already a language mechanism (composition) that provides this functionality, it is not clear what benefits arise from unrestricted use of this technique. In languages with sophisticated encapsulation mechanisms, such as C++, this use of inheritance allows more flexibility. For example, `private` inheritance in C++ allows access to `protected` members. Meyer [Mey94] discusses a more disciplined use of this technique. He observes that there are situations where a new class must inherit from one class (such as `Stack`) for the interface, and another (such as `Array`) for its implementation. (Snyder [Sny87] and Booch and Vilot [BV90] have made similar observations). Below is an array implementation of a stack of integers using composition. (Some details are left out for brevity):

```
class Arrayed_Stack -- version 1
inherit Stack
    redefine
      remove, put
    end
feature {NONE} -- representation
  rep: array[integer];
  size: integer;

feature -- interface
  remove is do
    size := size - 1;
    return rep.remove[size+1];
  end

  put(i: integer) is do
    size := size + 1;
    rep.put(i, size);
  end
end
```

The same example using Meyer's technique is:

```
class Arrayed_Stack -- version 2
inherit Stack, Array
feature {NONE} -- representation
  size: integer;

feature -- interface
  remove is do
    return remove[size];
  end

  put(i: integer) is do
    size := size + 1;
    put(i, size);
  end
end
```

Meyer observes that, without implementation inheritance, there are repeated qualified calls (to `rep`) which degrades performance and maintainability.

Our interest is what affect this technique has on the reusability of code. Unlike inheritance-by-composition, there are no benefits gained by not needing wrapper functions. Furthermore, the fact that a stack is implemented with an array does not mean we expect to use it wherever arrays are expected, so there are no benefits due to providing polymorphism. These observations suggest that implementation inheritance does not affect the reusability of code.

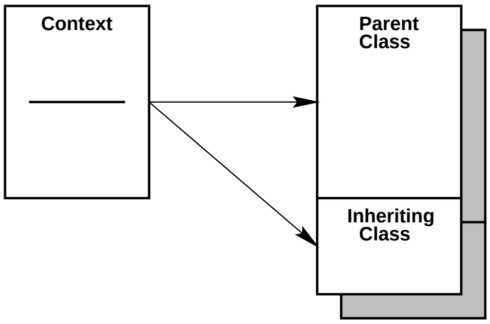

Figure 3: *Objects and classes may be created as specialisations of other objects or classes. This does this involve reusing the parent which they are specialisations. More importantly, this makes context code reusable.*

7 Polymorphism and Reusability

Polymorphism is the ability of an entity to take on different forms at run-time. This means that the code that actually gets executed can be decided upon at run-time. This is useful in situations where entities are of the same general form, but differ in specific detail. Because the entities are of the same general form, they can all be treated the same way, but when it comes to actually doing anything, they can do their own thing.

Consider again our customer example. Here we have a number of entities that are generally the same, however they differ in some details, such as whether an account has to be sent. Another way they may differ is in the price they pay. For example, subscribers may receive a discount. The standard argument for the benefits of polymorphism, is that without it, constructs such as

```
compute_price is do
   if customer.kind = Cash then
   ...
   else if customer.kind = Subscriber then
   ...
```

must be used, increasing the complexity of the code and increasing the cost of enhancements. Polymorphism allows the compiler to take responsibility for managing these constructs.

However, as well as reducing maintenance costs, polymorphism has other benefits. Assume that the customer classes have a `compute_price` feature and consider the following routine (which might be a feature of the `Point_of_sale` class).

```
print_receipt(a: Article, c: Customer) is do
  c.delivery_label;
  io.putstring(a.name);
  io.newline;
  c.compute_price(a);
end
```

This routine may be used as follows:

```
c: Customer;
s: Subscriber;
a: Article
print_receipt(a, c);
print_receipt(a, s);
```

Notice that the second call to the `print_receipt` routine may have been required long after it was written, but polymorphism allows it to be used without change. Thus we see that polymorphism affects the reusability of the *context*.

Thus, the real advantage of inheritance is gained because the new class conforms to the interface of the base class. That is: any place an object of the base class can be used, and object of the specialised class can too. Accordingly, any context code that takes objects of the base class can also be used with objects of the specialised class. This is the real advantage of inheritance: it makes the context code reusable.

8 Further Implications

In their introduction to *Design Patterns* [GHJV95], Gamma et al. observe:

> Many objects in a design come from the analysis model. But object-oriented designs often end up with classes that have no counterpart in the real world. Some of these are low-level patterns like arrays. Others are much higher level.

The patterns they go on to discuss are at the design level, and do not involve any particular implementation technique. However, the concept of inheritance is important in many of the designs. Generally, inheritance has been used in various ways for program organisation. Two of the more common are "abstract classes" and the use of inheritance to implement genericity. We discuss these briefly below to show that the nature of the reusability advantage is the same as for other uses of inheritance. The principle that inheritance facilitates context reuse is in fact very general, and underlies many more sophisticated methods of organisation, including for those involving multiple inheritance.

8.1 Abstract Classes

Inheritance is used to represent hierarchies of specialised classes in the application domain, but it can also play a role in program organisation. Where a class of objects may have several different implementations, they can be regarded as specialised cases of an *abstract class*.

The abstract class itself has no behaviour, but just specifies an interface or operations, with which the inheriting specialised classes conform. The advantage of this approach is that context code may then be written in terms of the abstract class, and then may be used with any of the specialised classes. In this way the context code will be reusable with any implementation of the class, even if several implementations are used within one program.

More generally, the concept of "frameworks" [JF88] involves sets of abstract classes as the basis for reusable design encoded as a kind of high-level implementation. In this technique, the set of communicating abstract classes represent the high-level design, and concrete classes that inherit from them implement the actual behaviour for specific programs. Again the reusability advantage of inheritance is the same, because it is the context level of the abstract structure that is made reusable with inheritance.

8.2 Genericity and Inheritance

Some entities are *containers*, and although they can hold other objects, these other objects are not component parts of the implementation. For example, the behaviour of a queue does not involve the behaviour of items in the queue. We model containers in programs too, where container objects often play important roles in the dynamic behaviour.

However, both in the real world and in programs it is common for some containers to have the same structure and behaviour as others, except for the kind of object they contain. For example, queues of people and queues of events have the same structure, and enqueue and dequeue operations work the same way. Note, however, that these two queues are hardly interchangeable, and that the type and consistency of the contained objects is obviously critical in programming. Some OOP languages provide explicit support for this via *genericity*, and allow the container to be described in a general way, but with parameters for the type of contained elements. With genericity, we can construct the container once, and leave the contained type as a parameter that can be provided as necessary. In this way genericity allows us to make the container more reusable, by being customisable.

Some OOP languages do not provide genericity, and inheritance is used by itself to address the issue of container reusability. In this approach, a container class is constructed to hold objects described by an inheritance hierarchy that encompasses all classes of objects that the container could hold. For example, if queues of employees and queues of integers were to be possible, the queue would have to be described as holding objects from an inheritance hierarchy that encompassed both employees and integers. There are several difficulties with this approach. Firstly, it means that large, complicated, and sometimes overlapping hierarchies must be described encompassing objects that have little in common. Secondly, because the reusability of the container depends on these hierarchies, it can be difficult to reuse the container classes in a new context with different hierarchies. And finally, where it *is* important that the contained objects be constrained to a particular class or sub-hierarchy, this requires inefficient run-time checking.

9 Conclusions

Object oriented programming has always promised better reuse, by implication through better reusability. Yet despite great enthusiasm, many programmers still find object-oriented design mysterious, and there is growing skepticism about the reusability. We believe this is due to the fact that literature has generally been unclear as to the precise link between OOP and reusability. Nowhere is this more true than in discussion of inheritance.

In this paper, we have performed an in-depth examination as to exactly how inheritance impacts the reusability of software. We observe that any discussion like must make the distinction between software reuse, and software reusability. This has been a problem with previous discussions. When inheritance is mentioned in discussions on reusability, it is quite often its code reuse mechanism role that is discussed. In this case, the claim is that the parent class has been made more reusable because now it can be used in a context where before it could not. In fact, as we have shown, the parent class can be reused using composition. What inheritance actually does it remove the need to write wrapper functions.

The second main role that inheritance plays is as mechanism for providing polymorphism, and in fact it is not necessary that it play that role. While it is generally understood that polymorphism is important, its role in increasing reusability has not been examined closely. Most discussions go no further than noting the “flexibility” it provides. Our contribution is to point out that it is *context* that is being reused, and hence it is the context that is being made more reusable. An important implication for software designers and programmers is that this is a valuable opportunity for creating reusable context code. Moreover, where this opportunity is of interest, care must be taken to ensure that reusabilty by inheritance is not compromised.

Finally, inheritance is a feature whose use has been subject to much debate. In particular, there have been arguments as to what its “proper” use should be. Our attitude, an attitude we take to all language debates, is to ask whether the proposed use of inheritance improves the reusability of code. We believe this is a useful way to evaluate language features. We plan to examine how specific implementations of inheritance in different languages impact the reusability of code.

References

[ABT94] Peter Andreae, Robert Biddle, and Ewan Tempero. Understanding code reusability: Experience with C and C++. *New Zealand Journal of Computing*, 5(2):23–38, December 1994.

[Boo94] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings, second edition, 1994.

[BP89] T. J Biggerstaff and A. J Perlis, editors. *Software Reusability*, volume 1. ACM Press, New York, 1989.

[BTA95] Robert Biddle, Ewan Tempero, and Peter Andreae. Object oriented programming and reusability. Technical Report CS-TR-95/6, Victoria University of Wellington, 1995.

[BV90] Grady Booch and Michael Vilot. The design of the C++ booch components. In *Proceedings OOPSLA/ECOOP '90, ACM SIGPLAN Notices*, pages 1–11, October 1990. Published as Proceedings OOPSLA/ECOOP '90, ACM SIGPLAN Notices, volume 25, number 10.

[FI94] William Frakes and Sadahiro Isoda. Success factors for systematic reuse. *IEEE Software*, pages 14–22, September 1994.

[Fis87] Gerhard Fischer. Cognitive view of reuse and redesign. *IEEE Software*, 4(4):60–72, 1987.

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

[JF88] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, June/July 1988.

[Mey86] Bertrand Meyer. Genericity versus inheritance. In Norman Meyrowitz, editor, *1986 Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, pages 291–405, October 1986. Published as ACM SIGPLAN Notices, 21(11), November 1986.

[Mey87] Bertrand Meyer. Reusability: the case for object-oriented design. *IEEE Software*, pages 50–64, March 1987.

[Mey91] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall Object-oriented series. Prentice Hall, 1991.

[Mey94] Bertrand Meyer. *Reusable software: The Base object-oriented component libraries*. Prentice Hall, 1994.

[Mic88] Josephine Micallef. Encapsulation, reusability and extensibility in object-oriented programming languages. *Journal of Object-Oriented Programming*, pages 12–36, April/May 1988.

[RTL+91] Rajendra K. Raj, Ewan Tempero, Henry M. Levy, Andrew P. Black, Norman C. Hutchinson, and Eric Jul. Emerald: A general-purpose programming language. *Software—Practice & Experience*, 21(1):91–118, January 1991.

[Sny87] Alan Snyder. Inheritance and the development of encapsulated software components. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object Oriented Programming*, pages 165–188. The MIT Press, 1987.

[Sof94] IEEE Software. Special issue on systematic reuse, September 1994.

[Str91] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 2nd edition, 1991.

[Ude94] Jon Udell. Componentware. *BYTE*, pages 47–56, May 1994.