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

Explaining Inheritance: A Code Reusability Perspective

Robert Biddle, Ewan Tempero

Technical Report CS-TR-95/18
September 1995

Abstract

Programmers new to the object-oriented paradigm often have difficulty learning how to use inheritance properly. In this paper we introduce an approach to explaining inheritance that is based on understanding the nature of reusability. We show how the important aspect of inheritance is *interface conformance*, and explain the role this plays in supporting reusability. We then outline a method for determining when and how to use both single inheritance and multiple inheritance, and discuss the implications of our approach.

Publishing Information

This report will be presented at the *Twenty-Seventh SIGCSE Technical Symposium*. Philadelphia, Pennsylvania. February, 1996

1 INTRODUCTION

Programmers learning the object-oriented paradigm (OOP) quickly become familiar with two principal claims: that OOP is a "natural" way to design programs, and that OOP provides good support for code reuse. However, while appreciating these claims in theory, many learners have difficulty in designing programs in practice. In particular, learners often have difficulty in understanding when to use the key OOP technique of *inheritance*, and even more difficulty with *multiple inheritance*.

Most writers on OOP attempt to explain the role of inheritance by showing examples of it being used in program design, usually in application domains that feature well-understood classification hierarchies (for example, the "zoo animal" examples of Lippman [4]). While this approach is appealing, the novice quickly finds that building one's own hierarchy is much more difficult than these presentations suggest. Other writers take a higher level approach, and introduce particular ways in which inheritance is useful, such the "facility inheritance" of Meyer [5]. While such distinctions can be interesting, introducing them early only heaps complications on top of a subtle subject.

The usual advice about inheritance is to use it to represent the "is a" relationship. While this advice is intuitively attractive, imaginative students have no difficulty envisaging "is a" relationships between some very disparate classes! This paper introduces our alternative approach to explaining when and how to use inheritance. It is based on the goal of producing *reusable software*, which has been the focus of our software engineering research [1].

Inheritance has long been associated with software reuse, but it is usually the case that inheritance is presented first, and then explained as a way of reusing code. Our approach is to reverse this: we discuss the creation of reusable software first, and then use that to explain how to use inheritance correctly. This approach is not in disagreement with standard examples or classifications, but offers both learners and teachers a different perspective. We have found the approach very useful in teaching OOP to both university students and programmers in industry; we have also found the approach very useful in our own design work.

In this presentation, we assume familiarity with the main concepts of OOP. However, understanding reusability is the foundation of our approach, so in section 2 we give a detailed review of the aspects of reusability we consider important. Section 3 explains the link between reusability and inheritance, and in section 4 we describe our approach to explaining the use of inheritance, specifically single inheritance, multiple inheritance, and abstract inheritance. We then present our conclusions.

2 REUSE AND REUSABILITY

Before discussing reusability, it is helpful to review the benefits it can provide. Reusing software has the potential for significantly reducing the cost of software in many ways. Reusing software reduces the time it takes to build new applications, because that software does not have to be reinvented (and rewritten and redebugged). Reusing software means that applications are easier to understand because there is less new software to learn, and, when the same thing happens in different parts of the application, it is the same software that is used. Finally, reusing software reduces maintenance costs, because if it turns out to be broken, it only has to be fixed in one place.

Actually reusing software involves many important factors, and the programming language level is not the only important concern (for further discussion, see [2, 6]). In practice, much software reuse begins simply with text editor cut and paste operations. However, such simplicity places a heavy burden on the programmer: understanding, checking, and customizing. Moreover, this approach results in multiple divergent copies of the original software, and

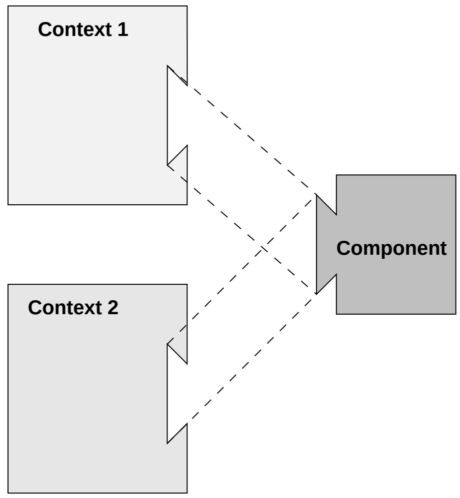

Figure 1: *A simple model for understanding reuse and reusability: the common case is where several contexts reuse a component.*

so voids many of the promises described above. To fully benefit from software reuse, software should be designed so that it is easy to use without change in new situations, that is, software *reuse* is best supported by *reusable* software. This means that to get the benefits of software reuse, programmers must understand how to use programming language features to create reusable software.

To explain the issues of reuse and reusability, we use the following simple model (see the "jigsaw" diagram in figure 1). We are interested in situations where two sections of code are related in that at some place or places one uses or *invokes* the other. The effect is as if code from the second section has been inserted into the first section at those places. In such situations, we call the invoking code the *context* and the invoked code the *component*. This distinction mirrors the distinction between reuse and reusability. Each invocation of a component by a context represents a use or reuse of the component, and the reusability of a component is related to the number of contexts in which it can sensibly be used.

This model is consistent with the programming language mechanisms that have supported reusability for many years: macro definition and expansion, and procedure definition and call. However, it also applies to concepts of OOP, such as the definition and instantiation of classes. Macros, procedures, and classes are all kinds of component, and are all designed for reusability in various ways. Macro expansion, procedure call, and class instantiation are all mechanisms that allow contexts to reuse these components. All these mechanisms work first at the level of a single program, where reuse and reusability are important because programs are developed over time. However, these mechanisms also work at the library level — macro libraries, procedure libraries, class libraries — where the benefits are more far-reaching.

Other facilities have been developed to provide support to make components more reusable. Some mechanisms, such as parameters, support customization. Other mechanisms are more subtle in their support of reusability. In particular, *encapsulation* constrains interaction between context and component to a specified *interface*: this supports reusability by ensuring independence of context and component, so maintaining their applicability to other situations.

An important aspect of our model is that it also works in reverse. The common case is as described in figure 1, where we have a reusable component, and reuse it from different contexts. However, sometimes we work the other way around: we have a context, and use it with different components, as shown in figure 2. For example, we might have context code

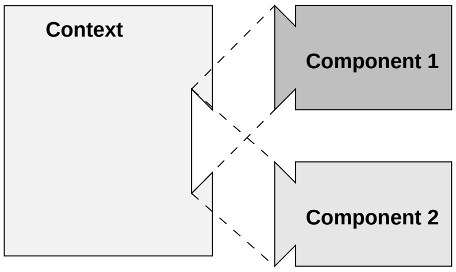

Figure 2: *Context Reuse: in this case one context invokes different components using the same interface.*

that calls a procedure, and in a different setting we may need the same context code, but want a different procedure to be called. We call this *context reuse*. In general, mechanisms that support reusability also allow context reuse, but some mechanisms specifically support *context reusability* — we discuss this further in the next section.

In our model, context and component are really *roles* played by sections of code. Sometimes a section of code can play both roles: the component role because it is invoked by other contexts, and the context role because it invokes other components. For example, a procedure may be called from elsewhere, but also itself call other procedures. In OOP, the equivalent is the organizational technique known as *composition* (some writers prefer the term aggregation). Use of composition allows the definition of a new class by reusing existing classes. The new class, as context, provides functionality in part by using the services of the existing classes as components. This shows that composition is a reuse mechanism.

3 INHERITANCE AND REUSABILITY

The discussion above shows how several key ideas of OOP (classes, encapsulation, and composition) are related to reuse and reusability. Our main observation regarding inheritance is that *inheritance supports context reusability*. This observation has important consequences: it provides guidance about *when* to use inheritance, and guidance about *how* to use it. To fully appreciate our approach, it is necessary to understand the development of this observation; we present it in this section. We then describe how we use this observation to explain the proper use of inheritance in the next section.

There are many similarities between composition and inheritance. As with composition, inheritance also allows definition of the new class (the child) by reusing an existing class (the parent). As with composition, the new class provides functionality in part by using the services of the existing class. And, as with composition, inheritance is a reuse mechanism — it provides a way to create a new class by reusing an existing one.

What is different about inheritance is the effect the parent class interface has on the interface of the new class. When a new class is created from an existing class using composition, there is no relationship between the interface of the existing class and the interface of the new class. With inheritance, the interface to the child class can *include* the interface to the parent class. As a consequence, the child's interface *conforms* to the parent's.

The child's interface conforms to the parent's interface if it includes all the parts of the parent's interface. This implies that instances of the child class may be used anywhere instances of the parent class may be used, since any operation that may be called on the parent must also be available for the child. This is why inheritance is often described as "is a" — a child class instance "is a" parent class instance. More importantly, interface conformance has significant implications for reusability.

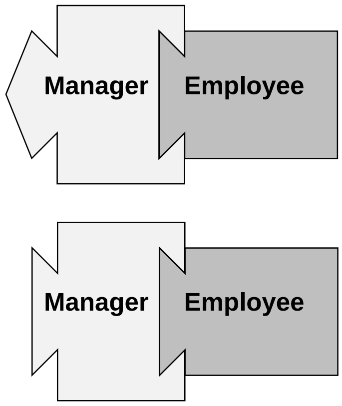

Figure 3: *The manager class can reuse the existing employee class by composition (top) or inheritance (bottom). However, composition produces a different interface (as shown by the shapes of the tabs on the top diagram), whereas inheritance can produce an interface that conforms to the employee class interface (as shown by the shapes of the tabs on the bottom diagram).*

In discussing our model, we explained that a class could be regarded as a component that could be used or reused by some context. Where context code uses a class, encapsulation means that the context is constrained to only use the class according to its interface. This has the often-stated advantage that the class implementation can be changed without having to change the context. It also means that if inheritance is used to create a new class whose interface conforms to the original class, then the new class can be used by the same context without modification. This is an example of *context reuse*, which we discussed in the previous section. Thus inheritance, because of interface conformance, improves the reusability of the context, since it allows the context code to be easily used in new situations.

Note that context reuse is possible using the class mechanism alone: with an existing context, we can implement the class it uses differently. With inheritance, however, we can use a context with several different classes, even in the same program.

Most languages with inheritance allow control of how the parent interface is inherited by the child. If such mechanisms are used in such a way that the child's interface does not conform to the parent's interface, then the main benefit of inheritance is lost. If the child's interface does conform, but has extra features, this is fine: any context code will still work with the child class. If the child's interface conforms to the parent's interface, but with different behavior, this also fine, because the context code will also work with the child class. This offers an alternative explanation of *polymorphism*: where context code is used with different classes that conform to one interface, but where each has different behavior.

4 WHEN AND HOW TO USE INHERITANCE

We believe that advice given to learners about inheritance is often too vague. As a result they do not reap the benefits of inheritance, even though they feel they understand it. Our approach is to give more objective advice based on the reasoning in the previous section. We outline our advice for single inheritance below, and in later sections address multiple and abstract inheritance.

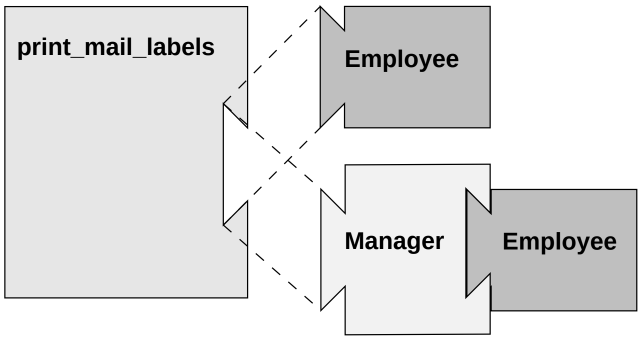

Figure 4: *Because the interface of the manager class conforms to the employee class, it can be used in place of the employee class in the context code.*

4.1 SINGLE INHERITANCE

Inheritance allows reuse of context code that uses the parent's interface. Inheritance is therefore worthwhile if reusing that context code will be desirable. Late in program development, the context code may already have been written, and reuse may be an attractive way to reuse old code and avoid writing new code. Early in program development, there may be the opportunity to design the context code in order to ensure its reusability with inheritance.

For example, consider the standard example of the programmer creating a personnel program. Late in development, there may be a need to create a manager class. It makes sense to base this new class on an existing employee class. As well as the code for the employee class, there is also likely to be context code written for the employee class. Such context code might take an employee, and print a set of mailing labels with the employee's name and address.

Both composition and inheritance enable the programmer to reuse the employee class (see figure 3). However, only inheritance enables the programmer to directly reuse existing context code written for the employee class. Creating the manager class by inheritance will allow the mailing label code to be used with both ordinary employees and with managers (see figure 4).

Early in development, the programmer will still be organizing the employee class and the context code that uses it. As we have observed above, this context code should be designed to facilitate later reuse by children of the employee class. At the early stage it may not be possible to know what such classes might be. However, to enable later reuse the context code should always be written in such a way that anything to do with the employee class should be handled by the class, and not by the context code. Thus the mailing-label code should not make decisions that depend on the fact that it is dealing with ordinary employees (for example, whether to use a home or business address), but should instead let the employee instance make the decision. This has the effect of improving the design of the employee class as well as creating context code that is more reusable. The same principles also apply to using and creating class libraries.

4.2 MULTIPLE INHERITANCE

The discussion above concerns ordinary or "single" inheritance. Single inheritance allows a new class to be created by reusing one existing class, and results in the new class conforming to the interface of its parent class. Multiple inheritance generalizes this concept: it allows a new class to be created by reusing several existing classes, with the resulting class interface conforming to the interfaces of *all* its parent classes.

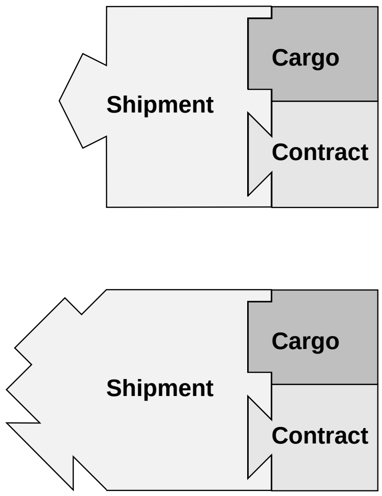

Figure 5: *The shipment class can reuse the exiting cargo and contract classes by composition (top) or inheritance (bottom). However only with multiple inheritance will the new class still be usable in any contexts expecting the parent classes.*

By creating a new class with multiple inheritance, a programmer can reuse the context code of all the parent classes. Code that used any one of the parent classes can now also use the new class, because the interface to the new class conforms to the interface of each of the parent classes. As is the case with single inheritance, multiple inheritance is therefore worthwhile if reusing the context code will be desirable.

For example, consider a transportation program. Late in development, several classes may already have been created, such as a contract class, for various legal details, and a cargo class, for various dimensional and care details. At this point, suppose there is a wish to create a shipment class, which must involve both legal and dimensional details. There could also be some context code that takes a contract and determines whether it is valid in a given jurisdiction, and other context code that takes a cargo, and determines what kind of insurance will be required.

Composition will enable the contract and cargo classes to be reused, as will multiple inheritance. However, only multiple inheritance will allow the programmer to reuse the context code of both contract and cargo (see figure 5). Creating a shipment by multiple inheritance of both contract and cargo will allow the jurisdiction code to be used with shipments as well as other contracts, and allow the insurance code to be used with shipments as well as other cargo.

4.3 ABSTRACT INHERITANCE

Another form of inheritance involves *abstract* classes, classes that cannot be instantiated, but just specify an interface. Rather than representing classification in the application domain, abstract classes indicate required functionality, which is then supplied by actual or *concrete* classes. In this way, abstract classes can be used to organize programs.

The advantage of abstract classes is that context code can be written in terms of the abstract class, and then used with any inheriting concrete class. In this way the context code will be reusable with any implementation of the class, even if several implementations are used within one program. In a sense, abstract classes represent a pure expression of our approach; because they are only interface, their only use is with classes that have conforming interfaces.

Abstract classes are also the basis of object-oriented frameworks [3]. In this approach, a high-level design is written as a program that consists only of abstract classes, and the design is applied to particular situations by providing implementations of the abstract classes. Frameworks can be seen as providing reusable context code.

5 CONCLUSION

We have described an approach for explaining inheritance based on creating reusable code. This approach relies on the fact that inheritance can be used to create new classes with interfaces that conform to existing classes, that is, the new classes can be used anywhere the existing classes might be used. Care must be taken to ensure that not only does the new class interface conform to its parent's, but that there is some utility gained by the conformance — there must be situations where it makes sense to use the new class in place of the parent.

While there are more exotic uses of inheritance, we have our doubts as to their usefulness; they often provide a reuse mechanism that could equally well be provided by composition and they do not contribute much to reusability. Whether or not they are in fact important, we believe their use is sufficiently complex that they should not be taught to beginners.

The main advantage of our approach is that it replaces the subjective "is a" test with the objective "usefully conforms to" test. Students find this test easier to apply as their own biases are less likely to affect their decision. This is not to say the "is a" test is wrong. The problem with it is that it takes experience to apply it correctly — experience that students lack.

The "usefully conforms to" test involves a more concrete explanation as to what is going on, and students tend to respond better to concrete explanations. It also gives teachers a specific criterion for evaluating student designs. Whereas some students may strenuously defend their decision that (for example) their phonebook class "is a" string class, the "usefully conforms to" test provides a convincing argument as to what the correct relationship really is.

Another advantage to our approach is that it gets students thinking about creating reusable code early. It has been observed that most software engineering programs teach software production from scratch, with little taught in the way of reusing code [7]. We believe that principles associated with creating reusable code should be taught along with general programming principles.

Finally, once the conformance relation between classes is understood, creating classroom examples to demonstrate inheritance becomes much easier.

What we have said about inheritance is not necessarily new, but we believe our contribution is in providing an explanation for how to use inheritance that is based on solid engineering principles, rather than subjective advice.

In summary, we recommend:

- teach software reusability first,
- explain clearly the similarities and differences between composition and inheritance, and
- advise using inheritance when it makes sense to have conforming interfaces.

ACKNOWLEDGMENT

We thank our colleague Peter Andreae for his care in proofreading, and for his always helpful comments and suggestions.

References

[1] Robert Biddle and Ewan Tempero. Understanding the impact of language features on reusability. Technical Report CS-TR-95/17, Victoria University of Wellington, 1995.

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

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

[4] Stanley B. Lippman. *C++ Primer*. Addison-Wesley, 2nd edition, 1991.

[5] Bertrand Meyer. *Reusable Software: The Base Object-Oriented Component Libraries*. Prentice Hall, 1994.

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

[7] Edward Yourdon. *Decline & Fall of the American Programmer*. Yourdon Press, 1993.