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

Object-Oriented Programming and Reusability

Robert Biddle, Ewan Tempero, and Peter Andreae

Technical Report CS-TR-95/6
July 25, 1995

Abstract

Some recent commentaries have suggested that Object-Oriented Programming has not been successful in increasing the productivity of software engineers. The main argument is that the object structure is not sufficiently reusable in different contexts. Alternative strategies involving "software components" have been proposed. In this paper, we explore the connection between object-oriented programming and reusability. We show that object-oriented programming does support design for reusability, but does so in ways that are not well understood. The designer must make careful decisions with reusability in mind, and in particular must include consideration of reusability in the process of class identification and class organisation.

1 Introduction

Proponents of object-oriented programming (OOP) claim it as the solution to the creation of *reusable software*, the holy grail of software engineering. Now however, some commentaries suggest it has not lived up to expectations ("Object technology failed to deliver on the promise of reuse." [14]). It is true that the amount of reusable software is not as high as the marketing hype associated with the object-oriented bandwagon has led us to expect. So what has gone wrong?

We believe this lack of success is due not to the inability of OOP to create reusable software, but to lack of understanding on the part of those trying to use the technology. The blame for this lack of understanding is mainly due to many of OOP's proponents, who often resort to platitudes such as "it's natural" to explain why OOP should be successful. This has given the impression that no effort is needed — reusable software will just happen if OOP is used. While many practitioners have realized that this cannot be the case, the lack of an alternate view has meant they have no clear guidance as to the proper use of OOP for reusability.

Another common exhortation to programmers is "design for reuse!" Here at least there are clearer guidelines as to what approaches to take (see for example [5]). However, such guidelines, while they may be presented in the context of OOP, do not explain how OOP helps (or hinders) in the implementation of such designs. Even good expositions of object technology that discuss design for reuse ([3]) seldom discuss how to use OOP effectively in this regard. Some go so far as to say reusability is not directly connected to OOP at all [12], but that it is solely due to good design.

Adding to the confusion is the fact that the area of "software reuse" is very broad. Most definitions of software reuse involve some variation of "the use of engineering knowledge or artifacts from existing systems to build new ones" [4]. This definition encompasses many diverse areas, from application generators, to domain analysis, to design patterns, to classification and search, to management [11, 2]. However there has been almost no discussion in any of these areas on how to use OOP effectively for the creation of reusable code.

While there are many important parts to the process of reusing software, in the end there has to be code to reuse [13] and it must be in a form amenable to reuse — the fact that the design is reusable does not mean the the code that results from it will be reusable. 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" [9]. This means that whatever programming language is used must have features that allow the creation of reusable code, and those features must be properly used.

We believe that object-oriented programming languages do provide appropriate features for creating reusable code, but that the correct use of these features is not well understood. In this paper, we explore the connection between OOP and reusability. We begin with a discussion of what it means for software to be reusable. This is fundamental to (and usually lacking from) any discussion of reusable software, for, if we do not know what it is, how can we know when we have created it? In section 3, we discuss the important features of OOP. We present OOP as "design by analogy", which we believe offers a concise, consistent, and hype-free description of the object-oriented paradigm. In particular, it finally settles the status of *genericity*. Genericity is not normally considered part of the object model but always seems to appear in such discussions. In section 4, we discuss the connection between OOP and reusability. We clarify the contribution of *inheritance*, a feature usually cited as evidence that OOP promotes reusable software. We show that current explanations of how inheritance promotes reusable software are incomplete. Finally, in section 5, we present our conclusions.

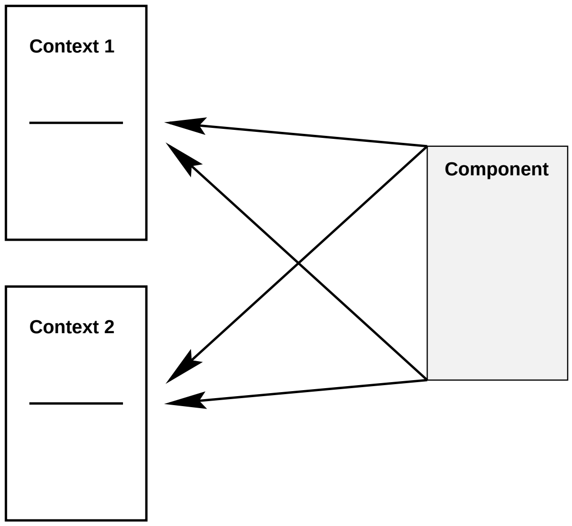

Figure 1: *Several contexts may invoke a reusable component. The key issues are the generality of the component, the flexibility of the interface, and the safety of the connection.*

2 Reusability

The first thing to understand is the distinction between *software reuse* and *reusable software*. Software reuse is an activity that takes place after software has been initially created. Software has been reused since the dawn of programming. Its most common form involves an editor and the cut-and-paste operation. On the other hand, reusable software is software that has been created in such a way that it is easy to use without change (that is, designed for reuse). This is a distinction that is frequently confused, probably because the proof of the reusability of software is its reuse, so it is very difficult to discuss one without the either (indeed, as we have done in the introduction). However, it is a distinction that we believe is important to understand in order to reap the full benefits of OOP. Much of the current misunderstanding about OOP is due to the features it offers that allow code to be reused (such as inheritance), rather than the features it has that helps in the creation of reusable code.

To understand how to make code more reusable, we must first consider how it is going to be used. 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*. A good model to use is that of procedure call; the component is the procedure and the place from which it is called is the context.

Reusable software is about creating components that can be used in all relevant contexts with the minimum of effort. The key terms here are “relevant contexts” and “minimum of effort”. A complete description of the effort required to use a component must include the effort required to determine what components are available, the effort required to confirm that a component is relevant to the intended use, and the effort required to use the component properly. Although the first two steps are important [10, 7], it is the last that has the biggest impact on the reusability of software — a component that requires an editor to make it useful is not a very reusable one.

Three important properties of code that make it more reusable [1] are *generality*, *flexibility*, and *safety*. Generality is about structuring the component so that details of its behaviour can be customised (for example parameters to procedures). Flexibility is about minimising the requirements of the context on what component is appropriate (for example, overloading of procedure names). Safety is making sure that the features for generality and flexibility do not result in the wrong components being used (for example, typechecking of parameters).

3 Object-Oriented Programming

To best understand how OOP involves reusability, it is important to understand just what OOP is all about. Unfortunately, the marketing hype about OOP obscures the truth, so this step is especially necessary. 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.

3.1 Technical Foundation: Types and Modules

The more technical foundation of OOP concerns an evolutionary process in program design and programming language support. In this process, beginning in the 1950s, programming has involved two elements: control and data. This split between the dynamic and the static permeates our history: instructions and memory, statements and variables, programs and files. Although both elements are always involved, much of the time program design has been more concerned with one than the other. For a long time, the design of control structure was the primary concern; some later techniques such as "Jackson Structured Programming" [8] shifted the emphasis to data structure design. OOP can be seen as a synthesis of the two approaches.

In organising control, we started with only the branching, structure and developed some very complicated programs. Development of the subroutine or procedure simplified this enormously, because it directly supported control flow reusability. The "structured programming" revolution further simplified control design, and showed that below the level of a procedure all we usually needed was a combination of sequence, selection, and iteration. Later, attention turned to the level above the procedure, where several procedures could work together packaged up as a module.

In organising data, we started with only indexable cells, and used them however we felt like using them. Simple types were introduced, and initially were thought of as allowing storage efficiency. Compound types, arrays and records, had a documentary effect, and introduced the idea of reusing data descriptions. We began to better understand the wider implications of the "type" concept, and support for user-defined types supported data description reusability more directly.

The technical view of object-oriented programming is that it is the confluence of these two threads of development. Take the encapsulation of a module, and apply it to operations on a user defined type, and the result is a class. An object is an instance of a class, and the rest is history. OOP involves both the organisational and reusability structures of both control design and data design together.

3.2 Philosophical Foundation: Design by Analogy

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 structured as *entities* and *actions* on entities.

The real world is complicated place, and so are many application domains. We do not always understand what the *real* structures are. However, the entity/action model is one that we apply very widely, and one with which we have great experience. We see our world as a world of things and doing.

Design by analogy can lead to easier program design, easier program understanding, and can also benefit program maintenance. The design is easier because there is a guiding philosophy: design the program like the domain itself. So if you understand the domain, you can proceed with the design. (If you don't understand the domain, you shouldn't be doing the design anyway.) And if the program has been written by someone else according to the "design by analogy" principle, anyone else who understands the domain has a head start in understanding the program. The question of program maintenance is trickier — the approach doesn't *necessarily* make maintenance easy. However, because the program is structured like the domain, changes in the domain will lead to similar changes in the program, and they will be of a similar complexity. This last aspect can greatly improve relations between programmer and client!

Moreover, design by analogy leads directly to reusability. In the world of entities and actions, it is typical that both entities and actions are involved in many different contexts. The same 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 be needed in the future.

4 OOP and Reusability

4.1 Objects and Classes

OOP begins with objects, which are the program abstraction for entities. Each object has a set of operations, which represent the actions that may be performed on the entity. The behaviour of these operations, and resultant states of the object, are analogous to the real world model.

OOP involves design by analogy with the entity/action model, but neither the model nor the analogy stops with entities and actions. In the real world, entities and actions are connected in various ways, and these ways of connecting are also important in OOP. We typically group objects into *classes*, and build upon them, typically using *composition*, *genericity*, and *inheritance*.

It is a big organisational step from seeing the 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 a larger class. In the real world, however, there are many ways to classify things, and the ways are not always consistent. Some OOP languages (such as Self [15]) allow dynamic organisation of objects; others (such as C++) 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. Both approaches foster reusability, because code can be written for any conforming, rather than for one particular object.

4.2 Encapsulation and Reusability

One of the key ideas from the technical foundation of OOP was the idea of encapsulation. This means that the object or class is a capsule, whereby the behaviour and state are bound together as a unit, with the only access being the designated operations. There are strong reasons of reusability for this approach.

For the object or class to be reusable in different (perhaps as yet unknown) contexts, it is necessary to make certain that the behaviour cannot be interfered with. For example, a queue

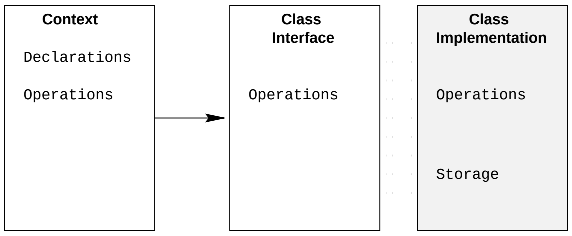

Figure 2: *A context is connected to an object or class component via an interface. The actual object or class implementation is encapsulated, and has no direct connection with the context at all.*

class cannot permit any particular context code to directly manipulate its implementation, because it may comprise integrity of the queue when used in other contexts. Moreover, the reusability issue here can work both ways. If it might be useful to use the context code together with a different implementation of the object or class, 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 class or object at all. For example, context code cannot be permitted to even access a queue implementation, because then it becomes dependent on that particular implementation, and so limits its reusability.

Encapsulation therefore supports reusability by ensuring that neither modification nor any access be permitted from context to object or class implementation. In fact, the reverse case is also important — the context must be protected from the object — for the same reasons, and encapsulation can also provide this protection. We regard the role of encapsulation here as supporting reusability by providing safety: ensuring code will be usable in other contexts.

4.3 Composition: Analogy and Reusability

Perhaps the most common way that two different entities can be statically structured together is by *composition*: where one object is used as a part of the other. This is common in the real world, 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.

4.4 Genericity: Analogy and Reusability

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. A key distinguishing characteristic of container objects is their operational interface, with facilities for inserting and extracting the contained objects, subject to the detailed behaviour of the container. Aside from this aspect, containers have few common features, and seldom have any particular programming language support.

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

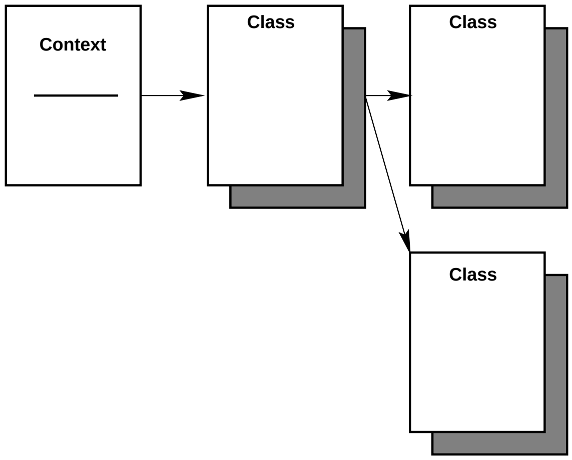

Figure 3: *Objects or classes may be implemented by using other objects or classes via composition. Not only does this approach allow component objects or classes to be reusable, but it also allows the composite class implementation to be general by using components defined elsewhere.*

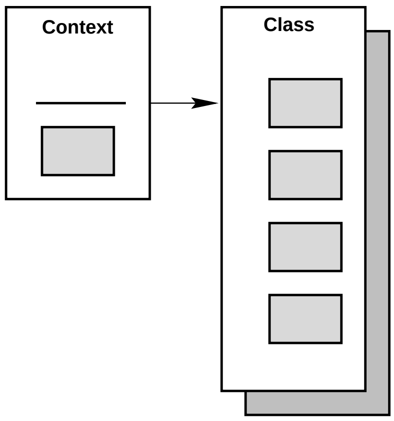

Figure 4: *Container objects or classes store other objects for later retrieval according to the specific behaviour of the container. This behaviour is often independent of the kind of objects contained.*

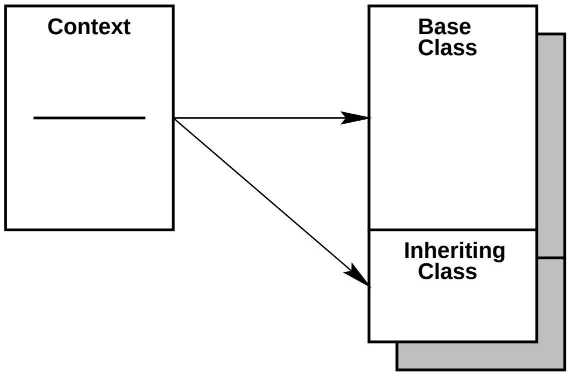

Figure 5: *Objects and classes may be created as specialisations of other objects or classes. This does this involve reusing the base of which they are specialisations. More importantly, this makes context code reusable.*

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. Without genericity, we would have to construct different containers that differ only in the type of the objects they may contain. 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 allowing the types of the contained elements to be customisable.

4.5 Inheritance: Analogy and Reusability

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 there are objects that do seem to belong in a class, except for some specialised operations and behaviour. 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.

In OOP, this kind of classification hierarchy is usually represented directly with *inheritance*. This is undoubtably the most celebrated mechanism in OOP languages, but also the most misunderstood.

The basic idea is that an object of a more specialised class inherits the operations and behaviour of a more basic class. The usual claim is that this is a big advantage because it involves reusing the operations and behaviour of the base class. It *does* involve reusing the base class, but it is *not* a big advantage. After all, the base class can be reused perfectly well by using composition. For example, consider a personnel system where an employee class is already developed: a manager class might later be developed by having a component part that is an employee.

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, any 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. For example, if a manager is regarded as a specialised form of employee, any program that uses employees will also work with managers. This is the real advantage of inheritance: it makes the *context* code reusable.

For example, consider again the personnel program with the existing employee class. Both composition and inheritance enable the programmer to reuse the employee c lass. However, only inheritance enables the programmer to reuse the context code. Some context code might take an employee, and print a set of mailing labels with the employee name and address. Creating the manager class by inheritance will allow the mailing label code to be used with both ordinary employees and with managers.

Some OOP discussion distinguishes inheritance and *polymorphism*. Inheritance can mean that objects of the specialised class conform to the base class interface in a primitive way by ignoring any specialised behaviour. Polymorphism then implies the extra step, where specialised objects conform to the base class interface, but retain their specialised behaviour. The nature of the advantage is the same flexibility in both cases, simply allowing lesser or greater reusability of the context code.

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.

Containers

In discussing containers and genericity, we mentioned the importance of consistency in the type of objects that container hold. For some containers this is straightforward, and all contained objects must be of exactly the same class. For others, however, some variability is perfectly reasonable. For example, a list of employees might reasonably contain managers. The real issue concerns how the objects are to be used. Because inheritance allows programmers to construct specialised objects that still conform to simpler interfaces, it is also useful in describing the variability that can be allowed for objects held in containers. If a container is constructed using genericity, there is no difficulty in using it to hold objects described with an inheritance hierarchy. In this way, the reusability of the container may be combined with the reusability of context code that uses objects held by the container.

Some OOP languages do not provide genericity (Smalltalk [6], for example), 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 might hold. For example, if queues of employees and queues of events 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 type checking. Because genericity allows individual containers to have their item types declared statically, it also permits more efficient compile-time type checking.

5 Conclusions

Object-Orientation enables reusability in several ways. The more philosophical foundation of OOP is design by analogy with the entity/action model of the world. And because entities and categories are frequently involved in different contexts, so objects and classes also span contexts. The more technical foundation of OOP involves encapsulation, composition, genericity, and inheritance. Each of these can be used to enable reusability in important ways:

- Encapsulation: ensures safety of connecting context and component, so ensuring both are reusable elsewhere.
- Composition: allows generality in object implementation contexts, so that the context is reusable with other component implementations, and equivalently that components may be reusable with various contexts.
- Genericity: allows the extra generality of customisability so that containers can be made reusable with contexts requiring different contained elements.
- Inheritance: enables flexibility of coupling context and component, so that the context may be reusable with components of conforming interfaces.

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 that OOP has great potential to foster reusability, but we believe many designers do not understand the issues well enough to turn potential into results. All the talk about OOP being "natural" seems to suggest that programmers do not need to understand it to make it work; that's just not true. Moreover, the literature is full of unhelpful explanations. Nowhere is this more true than in discussion of inheritance, where misplaced emphasis on the reusability of base classes is commonplace.

These relationships between OOP and reusability have some important implications for object-oriented design. Most importantly, program designers hoping to achieve reusability must understand how OOP can help deliver it. The process of object-oriented design is largely one of class identification and class organisation. At any point in the design, it is reasonable to consider whether, and how well, the reusability caters for future reuse. Throughout the design process, understanding how OOP can deliver reusability will can illuminate attractive options, and help lead to success.

References

[1] 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.

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

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

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

[5] 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.

[6] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its implementation*. Addison-Wesley, 1983. Reprinted with corrections, July 1985.

[7] Scott Henninger. Using iterative refinement to find reusable software. *IEEE Software*, pages 48–60, September 1994.

[8] Michael Jackson. *System Development*. Series in Computer Science. Prentice Hall, 1983.

[9] 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.

[10] Bertrand Meyer. Lessons from the design of the Eiffel libraries. *Communications of the ACM*, 33(9):69–88, September 1990.

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

[12] Bjarne Stroustrup. The design of C++. The Distinguished Lecture Series VIII (video), University Video Communications, March 1994.

[13] Will Tracz. Software reuse maxims. *ACM Software Engineering Notes*, 13(4):28–31, October 1988.

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

[15] David Ungar and Randall B. Smith. Self: The power of simplicity. In *1987 Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, pages 227–242, 1987. Published as ACM SIGPLAN Notices 22(12), December 1987.