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

Understanding OOP Language Support for Reusability

Robert Biddle and Ewan Tempero

Technical Report CS-TR-95/19
August 1995

Abstract

Object-oriented programming (OOP) has been widely acclaimed as a technology that will support the creation of *reusable software*. However, the practical impact has so far been limited. We believe success has been limited because of widespread misunderstanding of the way the technology supports reusability. In this position paper, we introduce our analysis of the connection between OOP and reusability. In particular, we show that inheritance *does* support reusability, but not in the way commonly supposed. More generally, we claim better understanding about language support for reusable software is necessary.

Keywords: Reusability, Object-Orientation, Programming Languages

Publishing Information

This paper appeared in the proceedings to the Seventh Workshop on Institutionalizing Software Reuse. August 1995, St. Charles, Illinois, USA.

1 Background

Object-oriented programming (OOP) has been widely acclaimed as the technology that will support the creation of *reusable software* [10]. However, it is also 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?

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" [5]. This definition encompasses many diverse areas, from application generators, to domain analysis, to design patterns, to classification and search, to management [12, 4]. 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" [9]. We have been looking at this issue from the point of view of specific languages and specific features [1, 3, 2] .

2 Position

We believe the lack of success of OOP is due not any particular shortcoming in the technology, but to widespread misunderstanding of the way the technology supports reusability. In this position paper, we briefly outline our analysis of the connection between object-orientation and reusability. 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. Finally we discuss the key issue of the role of inheritance in software 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.

2.1 Reusability and Reuse

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.

It is our broad aim to better understand how to support reusability. Towards this, we wish to analyse language structures, and also develop a language independent model for reusability support.

We refer to the code that we want to reuse as the *component*, and this is used by other code that we call 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.

A context is relevant if it can use functionality in the component. However, the functionality may not be accessible by the context. Typical reasons for this are: unhelpful encapsulation, namespace conflicts, and dependence on irrelevant or inaccessible components. A reusable component therefore must be designed to avoid such limitations. We refer to this property as *generality*.

Where a context invokes a component to make use of some functionality, nothing but the functionality should matter. In fact, there may be no requirement that it be provided the same way every time. The weaker the binding between the context and the component, the more components can be used in that context. The strength of the binding required between the context and the component represents the *flexibility* of the binding.

Finally, a context invoking a component connects two different sets of assumptions and behaviour. Care must be taken to prevent connections where assumptions and behaviour are sufficiently misaligned that the results are not well defined. Prevention of this kind of mistake requires concern for *safety*.

2.2 Object-Orientation 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.

2.3 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 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.

2.4 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 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.

2.5 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 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 main idea is that object of a more specialised class inherits the operations and behaviour and operations 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* involves 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, a manager might be regarded as having a component 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, an 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.

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.

Inheritance and polymorphism are used to represent classification hierarchies in the application domain, but 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 specifies an interface to which inheriting classes conform. The advantage is that context code may then be written in terms of the abstract class, and then used with any inheriting 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. Abstract classes are also the basis of Object-Oriented Frameworks [7]. 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. In this way the advantages of reusability can be extended to the program design level, as well as the program implementation level.

2.6 OOP and Reusability Support

Earlier we outlined our working model for reusability support. In these terms, the key features of OOP can be described as follows:

- 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.
- Inheritance: enables flexibility of coupling context and component, so that the context may be reusable with components of conforming interfaces.

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.

In particular, there are significant implications of our observations about inheritance and reusability. Because inheritance supports reusability of context code, programmers should take care to design context code to take advantage of this support, and thus facilitate later reuse. We believe this is the major implication of inheritance for program designers, but also suspect it is not widely understood.

Throughout the design process, understanding how OOP can deliver reusability will can illuminate attractive options, and help lead to success.

3 Comparison

While there have been a number of discussions on reusability, very few consider the impact of language features on the reusability of software. Those that do tend to blur the distinction between reuse of software and reusability of software, often using both concepts in the same sentence as if they are synonyms [10, 11]. As a consequence, any comments that do pertain to reusability are overlooked.

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[7] or higher-level class organizations that have been found to have been useful in the past [6]. Other efforts such as the ”adaptive software” approach [8] support reusability with strategies and tools that use structures within the code, but that work at a level above the code. While all these approaches are important, and offer significant assistance in the creation of reusable code, in our own work we have been primarily concerned with support at the programming language level.

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] Robert Biddle and Ewan Tempero. Reusability and inheritance. Technical Report CS-TR-95/8, Victoria University of Wellington, 1995.

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

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

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

[6] 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.

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

[8] Karl J. Lieberherr, Ignacio Silva-Lepe, and Cun Xiao. Adaptive object-oriented programming using graph-based customization. *Communications of the ACM*, pages 94–101, May 1994.

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

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

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