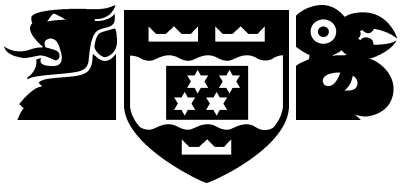
VICTORIA UNIVERSITY OF WELLINGTON



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

Modelling Reusable and Adaptable Software

Robert Biddle and Ewan Tempero

Technical Report CS-TR-95/20
October 1995

Abstract

This paper discusses issues in reusable and adaptable software. It presents a model for explaining how programming language features impact the reusability of code.

Publishing Information

This paper appeared at the *Adaptable and Adaptive Software Workshop, OOPSLA'95*

Introduction

We would like to better understand how to support the creation of reusable code and are concentrating on issues involved at the programming language level [1]. Towards this, we wish to analyse language structures, and develop a language independent *model* for reusability support. The history of science shows that understanding phenomena is often facilitated by constructing a model to help discuss and explore the nature of the phenomena. It is our position that constructing a model of software reusability will help our discussion and exploration, and ultimately assist us in removing barriers to the creation of more reusable software.

Software *reuse* can reduce the time to build new applications, because software does not have to be reinvented (and rewritten and redebugged). Reuse can mean applications are easier to understand, because they are built using well-known components. Reuse can also reduce maintenance costs, because components can be improved and the benefits recouped by all applications that use them.

Actually reusing software involves many important factors, and pragmatic management considerations often outweigh theory (for further discussion, see [2, 5]). In practice, much software reuse simply involves text editor cut and paste, resulting in multiple divergent copies that void many of the promises described above. To fully benefit from software reuse requires more sophistication, and involves the software itself. In order to best enable later reuse, software should be designed in such a way that it is easy to use in new situations. Software *reuse* is best supported by *reusable* software.

Creating reusable software involves many factors. At the design and implementation level, it involves programmers who make many practical decisions that make software more or less reusable. It also involves the design of programming languages, where many structures and mechanisms support reusability in various ways. Operating systems and programming environments also have important roles in support of the language features. We consider the work on "adaptive software" [4] as also making important contributions in supporting reusability. In particular, this work has shown that reusability can be improved if code can be adapted to new requirements in ways that can be supported automatically from a level at or above the code.

Our reasons for building a model are essentially practical. We find the variety of approaches to reusability support so diverse that we have difficulty in making sense of it all. As programmers, we want to understand better how to create reusable software, and what technology will help us in our efforts. As teachers, we want to be able to explain clearly both the issues and attempts to address them. As researchers, we want both a structure and terminology to facilitate our efforts to understand the past, and to discuss the future.

Our approach in modelling has been to start by considering practical experience, and to build a model that encompasses this experience as simply as possible. We began by describing what we felt to be the simpler and more obvious aspects of reusability. This involved considering mechanisms that we felt had been clearly successful in supporting reusability, and so essential for any model to reflect. We then considered more mechanisms, and more about the ways the mechanisms were used, and attempted to assemble all the pertinent details.

The Model

Our model consists firstly of a simple structure reflecting straightforward reuse and reusability (see 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 the second section of code has been inserted into the first section. In such situations, we call the invoking code the *context* and the invoked code the *component*. 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 be sensibly be used.

This structure reflects the programming language mechanisms that have supported reusability for many years: macro definition and expansion, and procedure definition and call. However, it also

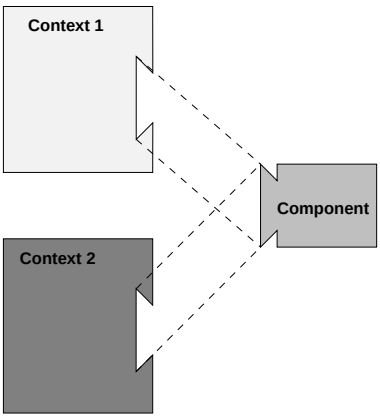


Figure 1: *A simple structure for understanding reuse and reusability: the common case is where several contexts reuse a component.*

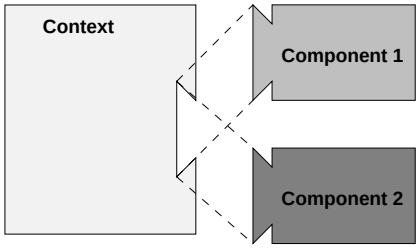


Figure 2: *Context Reuse: in this case one context invokes different components using the same interface.*

applies to concepts of OOP, such as the definition of classes, and the instantiation and use of classes as objects.

An important aspect of the structure is that it also works in reverse. The common case is as described in figure 1. 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 that calls a procedure, but 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*.

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. In OOP, this is the organisational technique known as *composition* (some writers prefer the term aggregation). Use of composition allows the definition of the 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.

An important aspect of the two segments of code is where they meet, that is, their *interface* (using the English meaning of "interface"). The context and component code can affect each other's behaviour only through the interface. We call the various ways the two segments can affect each other *dependencies*, representing the fact that each segment depends on the other doing the right thing. We describe the interface between the component and context by enumerating all the dependencies between them. Note that not only can the context depend on the component, but the component can also depend on the context. Finally, the interface of a single segment can be described as the dependencies that must be satisfied.

We classify dependencies in two ways: *contract* or *non-contract*, and *explicit*, *implicit*, or *informal*. Contract dependencies are those that have been intentionally introduced by the programmer, whereas

	Contract	**Non-contract**
Explicit	Public interface of a class.	Modula-2 interface (exposes type declarations).
Implicit	Use of `extern` functions in C++.	Non-`static` global variables that should be `static`.
Informal	A list that keeps the items in order.	The order that an iterator produces items from a set.

Figure 3: *Categories of dependencies*

non-contract dependencies exist accidently (either by mistake, or because the language does not provide support for removing them). Code that relies on non-contract dependencies is less likely to be reusable. *Encapsulation* can be seen as reducing such dependencies. Explicit dependencies are those that are described directly in the language. Implicit dependencies are those for which there is no language support for describing them, but which can nevertheless be checked in some way. Informal dependencies cannot be described in the language, nor can they be checked. Informal dependencies are not as helpful as implicit dependencies because there is no way to ensure they have been met. Implicit dependencies are not as helpful as explicit dependencies because because it is not obvious what must be done to meet them. We give examples for each of the resulting categories in figure 3.

We can describe how language features affect the reusability of code by focusing on dependencies. For example, in most languages, the use of a global variable in a component represents a dependency by the component on the name from the context. This is an informal dependency. By introducing a parameter to replace the use of the global variable, we replace the implicit dependency by an explicit dependency — now the component depends on the context to supply a value to the parameter. As another example, pass-by-value can be seen as a dependency by the component on the values supplied by the context, but note that the context in no way depends on the formal parameter used by the component: the dependency is one-way. On the other hand, pass-by-reference also introduces a dependency by the context on the component — the context now relies on the component changing the value of the formal parameter in the "expected" way.

Removing non-contract dependencies and making the rest explicit does not necessarily mean we make the code more reusable. We need to know which dependencies should be made explicit. For this we need the concepts of *safety* and *generality* of code. Safety represents how and when the obligations introduced by the existence of dependencies are met. Generality consists of *flexibility*, how to relax any checking while considering safety, and *customisability*, how to introduce useful dependencies. The creation of reusable code can then be described as increasing generality while maintaining safety.

Using the Model

Although our model is not fully developed, it has already been useful to us in improving our understanding of reusability. In particular, we have gained a new perspective on mechanisms involved in OOP. The discussion above shows how the model encompasses several key concepts of OOP: classes, encapsulation, and composition. Our next step was to consider the role of inheritance and related mechanisms.

As with composition, inheritance allows definition of the new (child) class by reusing an existing (parent) class. What is different about inheritance is that it can affect the interface of the new class: the interface to the child class can *include* the interface to the parent class. For reusability, this is the important aspect of inheritance: *interface conformance*. 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.

This is an example of context reuse, which we discussed in the previous section. Of course, context reuse is possible using the class mechanism alone: we can take an existing context and

implement the class it uses differently. With inheritance, however, we can use a context with several different classes, even in the same program. This is the primary connection between inheritance and software reusability: *inheritance supports context reusability*. This observation has important consequences: it provides guidance about when to use inheritance, and guidance about how to use it.

If the child's interface 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 behaviour, this also fine, because the context code will also work with the child class. This is an alternative explanation of *polymorphism*: where context code is used with different classes that conform to one interface, but where each has different behaviour.

Polymorphism reduces the strictness of the type checking, and so makes type dependencies more flexible. In a similar way, propagation patterns [4] can be seen as making dependencies more flexible.

An important form of inheritance involves *abstract* classes. 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.

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. Just as reusable macros enable macro libraries, and reusable procedures enable procedure libraries, we speculate that in a similar way abstract classes and frameworks could lead to "context libraries".

Conclusions

We have proposed developing a model to help understand the nature of software reusability, and presented a preliminary sketch for such a model. Our model consists of two mains roles for code: *context* and *component*, where reuse each way is of interest. Of key importance is the *interface* consisting of *dependencies*. Our model has assisted us in better understanding the connection between OOP and reusability by clarifying the effects of inheritance and some related mechanisms.

We continue to use our model to help analyse various strategies that support reusability. We hope to develop our model further, and are interested in perspectives gained from other exploration in this general area, especially including adaptive software. We feel that the key concept is how dependencies govern reusability, and believe better understanding of this is important to developing more reusable and adaptable software.

References

[1] Robert Biddle and Ewan Tempero. Understanding OOP language support for reusability. In *Seventh Annual Workshop on Institutionalizing Software Reuse (WISR7)*, August 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] 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.

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