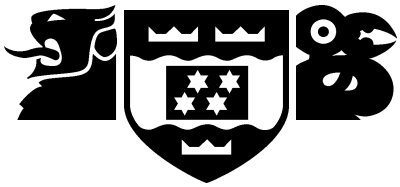
VICTORIA UNIVERSITY OF WELLINGTON



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

Teaching Design by Teaching Reusability

Robert Biddle and Ewan Tempero

Technical Report CS-TR-96/17
1 November 1996

Abstract

Initial teaching of programming typically stresses language and program implementation issues, rather than design. We suggest a focus on reusability offers a way to introduce design at an early stage. Our approach is based on a general model for understanding reusability. It allows students to understand some basic design issues that assist programming language learning and program implementation, by establishing principles at a higher level. This paper outlines our model, describes how we use it to structure our teaching, and explains how it supports early teaching and learning of design.

Publishing Information

This paper appeared at the *Teaching and Learning Object Design in the First Academic Year* Workshop, OOPSLA'96

1 Background

Our recent research work has involved developing a conceptual model of programming language support for reusability [3]. We initially created the model to facilitate discussion and understanding of reusability in the context of programming language design, with the goal of improving software development. However, we increasingly believe the model is also useful in our teaching. It provides an effective way to explain both design concepts and language features in a way that emphasises reusability, and also provides practical guidance for developing reusable code. We have been exploring using this approach in teaching both university computer science students and professionals in industry, and have found the experience encouraging.

2 Position

Initial teaching in computer science is usually focussed on students learning a programming language and implementing programs. This focus has the advantage of providing concrete experience for students that quickly reinforces their learning and empowers them to tackle real tasks. However, this focus has the disadvantage of misleading students into thinking that language learning and program implementation are the only issues in software construction.

In reality, software construction involves many issues concerning *design*. In initial courses, however, we typically teach design only implicitly, with examples, or with vague guidelines. The limited time and small scale that are typical of initial courses also compound misunderstanding.

Our position is that it *is* possible to address design issues explicitly at an early stage, and this can be done while students are learning a programming language and starting to implement small scale programs. The design issue we stress is *reusability*. We explain the importance of reusability, we explain language features in terms of reusability, and we explain how to make design decisions by considering the impact on reusability.

The next section of this paper briefly describes our model for understanding the reusability. We then outline how we use the model in our teaching, and explain our view of the connection between reusability and design. Finally we describe our experiences in teaching with this method, and offer some conclusions.

2.1 A Model for Understanding Reusability

In our approach, we explicitly distinguish the concepts of "reuse" and "reusability". While there is clearly a relationship between the two — the proof of the reusability is in the reuse — there is also very definitely a difference. Code reuse is an activity that takes place after code has been initially created, whereas reusable code is the result of activity that takes place when code is created.

Code has been reused since the dawn of programming through the use of some form of cut-and-paste operation. While this is reusing code, it is ad hoc in nature. This means it cannot be systematically applied in new situations and so cannot provide many benefits. The benefits we expect from *reusable* code include:

- new applications can be built faster, because code does not have to be reinvented (and rewritten and redebugged),
- applications are cheaper to maintain, because components can be improved and the benefits recouped by all applications that use them, and
- applications are easier to understand, because they are built using well-known components.

To get these benefits, we need the kind of code arrangements depicted in figure 1. On the left of the figure, two segments are related in that at some place or places one uses or *invokes* the other.

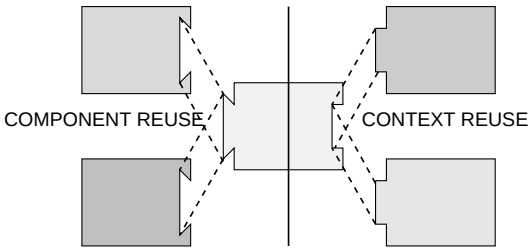


Figure 1: *Component Reuse, as shown on the left of the diagram involves several different contexts invoking the same component. Context Reuse, as shown on the right of the diagram, involves one context invoking different components using the same interface.*

The effect is as if the second segment of code has somehow been inserted into the first segment. In such situations, we call the invoking code the *context* and the invoked code the *component*. Each different context that invokes a component represents a use or reuse of the component. 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 applies to more recent concepts, such as the definition and instantiation of classes in object-oriented programming.

An important aspect of this structure is that it also works in reverse. The common case is *component reuse* as described on the left of figure 1. Sometimes however, we work the other way around: we have a context, and use it with different components, as shown on the right of the figure. For example, we might have context code 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*.

Context and component are really *roles* played by segments of code. As the figure shows, a segment 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. This observation, together with the notion of context reuse, shows that any discussion of reusability must encompass both component and context. We believe this is a common failing in previous discussions of this subject.

When a context invokes a component, it has expectations as to what the component does. Similarly, the component has expectations as to how it will be invoked. We call these expectations *dependencies*, and they are the fundamental low-level concept of our model. Dependencies can adversely affect the reusability of a segment of code, because they limit the number of contexts than can invoke it, or the number of segments it can invoke. However, there are dependencies that are beneficial to reusability, by allowing customisability and checkability.

We are especially interested in how language features affect software reusability, and we have also found it useful to classify dependencies according to the support provided by the software development environment.

We define reusability in terms of the number of contexts or components that can *usefully* invoke other components or contexts. This definition reflects the fact that it doesn't matter how reusable a segment of code is, it is not useful if it doesn't do what is required. Thus, determining whether a segment of code is useful is important. Research in human-computer interaction has explored the concept of "usability" and has resulted in improved understanding of many issues in user interface design [7].

Table 1 summarises the main features of the model; further details are described elsewhere [3].

2.2 Teaching with the Reusability Model

In this section we briefly explain how our model can be used to focus on reusability while teaching how to program. It is our intention that this approach not only emphasises the importance of reusability,

- segment roles:
  context and component
- programmer intentions:
  contract and non-contract
- language support:
  explicit, implicit, and informal
- benefits:
  generality (customisability or flexibility)
  checkability
  usability

Table 1: Model summary

but also motivates a deeper understanding of the program design process, and how programming languages support this process.

Our approach assumes that students understand the primitive elements of programming such as variables, expressions, statements, sequences, selection and iteration. Accordingly, we begin by teaching these elements in a conventional way (however, we are considering introducing our approach even earlier.) At this point, any non-trivial program built using only these elements will quickly involve repeated programming, debugging and so on. It is then very easy to motivate reusability as an important issue.

We begin with a simple version of our model, one that explains programs as consisting of segments which can be invoked as necessary, thereby avoiding repeated work. This leads immediately to the introduction of procedures, and we can discuss component reuse (calling the same procedure from different places) and context reuse (multiple implementations of a procedure).

We make the point then that we seldom need to reuse things the exactly the same way every time, and so discuss generality. We introduce the use of scope as a means of making segments general, and then introduce parameters and compare the two, showing how parameters allow explicit dependence on values, instead of implicit dependence on names.

We introduce classes initially as segments of data declaration, and can show both component reuse (using the same class for different declarations) and context reuse (multiple implementations for the same class). Supporting context reuse leads then to encapsulation, because encapsulation prevents contexts from depending on class implementations, so ensuring that multiple implementations can be used with a given context.

Throughout, we recall that context and component are roles, and so dependence is transitive. For procedures, this means procedures calling procedures that call procedures, etc. For classes, this means classes composed of other classes, etc. We are especially careful to stress the importance of composition prior to inheritance, because we find many students use inheritance too enthusiastically.

Our explanation of inheritance is that it supports context reuse, by allowing a new class to conform to the interface of an existing class. Because of the interface conformance, the new class can thus be used with context code created for the existing class. Figure 2 illustrates the "jigsaw" diagrams we use to make clear the difference between inheritance and composition.

We have detailed how our reusability model can be used to develop and organise a teaching sequence, as summarised in the table below. We feel this teaching sequence is valuable in many ways, most importantly by giving early implicit and explicit emphasis to reusability, and by providing practical justifiable guidance that helps students learn to program well.

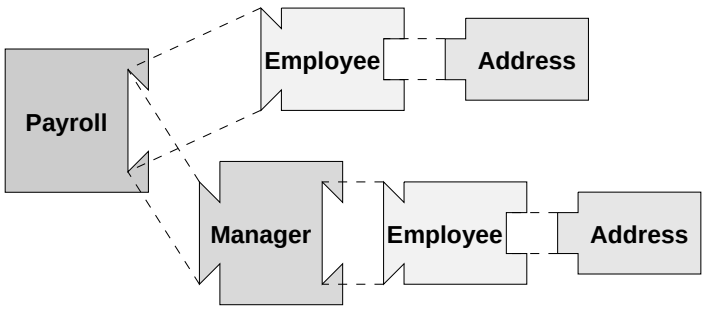


Figure 2: *This jigsaw diagram illustrates how interface conformance makes inheritance different than composition. An "employee" class can reuse an existing "address" class by composition (top), but composition produces a different interface for employee than for address (see tabs on top part of diagram). However, a "manager" class can reuse the existing employee class via inheritance, and have an interface that conforms to the employee class interface (see tabs on bottom part of diagram).*

We follow up this presentation sequence with hand-on laboratory exercises structured in a very similar way. We always begin by having the students construct software by reusing provided code, then have them create code in place of that provided, then generalise the code, then reuse their own code in different contexts. All our exercises are in a single application domain. We then assign more lengthy programming projects involving the same principles, but in a different application domain.

2.3 Design and Reusability

When we create software, we use the potential of some technology to address the requirements of some application domain. Almost always there are many possibilities, and determining which possibilities to choose is what design is all about. The technology and the application domain both affect design themselves.

Typical introductions to programming stress the technology: the programming language and environment. Much real-world software development must focus on the application domain: what the software is *really* supposed to do. There are many ways to consider the quality of software, and accordingly many approaches to design. We believe that our approach, stressing reusability, offers advantages because it forms a bridge from the technology of the programming language to higher level design issues. Moreover, it is a very concrete and practical approach, and so allows early experience and consequent empowerment.

One of the strengths of the object-oriented paradigm is that it allows an easy route to design involving the application domain. OOP is "programming by analogy", and so the classes and objects in the application domain can be used as models for classes and objects in the program. Unfortunately, our experience is that the real world is a complicated place, and program design still requires experience that students lack. However, we have found that considering reusability offers a useful beginning for object-oriented design, and makes it simpler for students to check the sanity of their design.

We have found the approach especially useful in introducing design using inheritance. Many introductions to OO attempt to explain the role of inheritance by showing examples of it being used in application domains that feature well-understood classification hierarchies (for example, the "zoo animal" examples of Lippman [6]). Students find this approach appealing, but quickly discover that building one's own hierarchy in realistic programming situations is much more difficult than these presentations suggest.

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

- Model:
  component reuse
  context reuse
  generality
  checkability
- Procedures:
  procedure reuse
  procedure call reuse
  global variables; parameters
  name and type checking
- Classes:
  class reuse
  class declaration reuse
  global types and constants; parameters
  name and type checking; inheritance

Table 2: Teaching sequence summary

between some very disparate classes! An 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. Moreover, it replaces the philosophical nature of "is a" with the very practical nature of "usefully conforms to", so making software reuse the important criterion. We discuss our approach to teaching inheritance via reusability in detail elsewhere [2].

2.4 Experience

Our experience in applying this approach is now almost four years old, and includes four times teaching a university computer science programming course to second year students, and about fifteen short courses on object-oriented programming for professionals in industry. However, the approach has developed gradually, and we have introduced new elements of the approach over time.

The programming language we usually use is C++, and although we are fairly committed to the object-oriented paradigm, our exact choice of language has less to do with philosophy than practical issues of software availability and local employment demands. (We have discussed this issue specifically elsewhere [1].)

In both our teaching to university students and in industry, we regard our experience using this new approach as highly successful. We feel that we are able to explain design issues more clearly, and based on clear principles. Moreover, student feedback has been positive, and several students with previous experience have reported that gaining better understanding than they had from previous courses.

Some students have reported confusion because we don't echo the books and magazines they have read. For example, we still see some students with wildly enthusiastic inheritance hierarchies complicating programs that never needed them. Even these students benefit from our focus, however, because the focus provides an objective and principled way to discuss and evaluate their programs so that they can see both why and how they might do better.

The interplay of principles and reality is important at all levels. For beginners, it is important to remember that learning practical programming involves learning a programming language. However, this can result in beginners confusing general principles with language peculiarities. Our approach

addresses this by making it clear that the language features are not themselves the principles, but represent attempts to support the principles. Accordingly, language details are just superficial details, but they serve a deeper purpose. This keeps the students aware of the role of the language, and of its strengths and its limitations. We hope this will mean students will be better able to understand other languages later on, rather than being "locked in" to one way of programming. In fact, while explaining features of the language we are using, we often mention how other languages work differently.

3 Comparison

Our work stems from our model for understanding reusability. The previous work most similar in character to ours is the "3Cs" model [4, 5, 8]. Like ours, this is a conceptual model designed to foster better articulation and to promote deeper insight about reuse and reusability. However, the 3Cs model directly concerns the design of reusable software components in general, whereas our model concerns programming language support for reusable software components.

Our model was developed independently, but of course both models involve some similar distinctions. Our model is the result of a more "bottom-up" approach, and we believe it offers a new and worthwhile perspective. While the "3Cs" model has been used to group together a number of guidelines for design for reusability, we believe our model provides a smaller and more powerful set of principles for better understanding the key issues.

References

[1] Robert Biddle and Ewan Tempero. Teaching c++: Experience at victoria university of wellington. In *Proceedings of the Software Education Conference*, Dunedin, New Zealand, November 1994. IEEE Computer Society Press. Also available as Technical Report CS-TR-94/18.

[2] Robert Biddle and Ewan Tempero. Explaining inheritance: A code reusability perspective. In *Proceedings of the Twenty-Seventh SIGCSE Technical Symposium*, February 1996. Also available as Technical Report CS-TR-95/18.

[3] Robert Biddle and Ewan Tempero. Understanding the impact of language features on reusability. In Murali Sitaraman, editor, *Proceedings of the Fourth International Conference on Software Reuse*. IEEE Computer Society Press, April 1996. Also available as Technical Report CS-TR-95/17.

[4] Stephen Edwards. The 3C model of reusable software components. In *Proceedings of the Third Annual Workshop: Methods and Tools for Reuse*, June 1990.

[5] Larry Latour, Tom Wheeler, and Bill Frakes. Descriptive and prescriptive aspects of the 3Cs model: SETA1 working group summary. In *Proceedings of the Third Annual Workshop: Methods and Tools for Reuse*, June 1990.

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

[7] Jakob Nielsen. *Usability Engineering*. Academic Press, New York, 1992.

[8] Will Tracz. The three cons of software reuse. In *Proceedings of the Third Annual Workshop: Methods and Tools for Reuse*, June 1990.

4 Biography

Robert Biddle and Ewan Tempero are on the faculty of the department of Computer Science at Victoria University of Wellington, New Zealand.

Robert Biddle's research is in software engineering, particularly in software reusability and in aspects of human-computer interaction. He is also interested in Computer Science Education. Robert has experience in development of commercial computing applications, and in research and development in industry. He studied at the University of Waterloo in Canada, and at the University of Canterbury in New Zealand.

Ewan Tempero teaches courses in algorithm design and concepts in object technology. His research primarily involves understanding how programming language features affect the reusability of code, particularly features associated with Object-Oriented languages. He has also carried out research in distributed systems, investigating language support for distributed applications and formal analysis of networking protocols. He graduated from the University of Otago, New Zealand, with a BSc (Honours) in Mathematics in 1983 and received his PhD in computer science from the University of Washington, USA, in 1990.