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 programming by teaching reusabilty

Robert Biddle and Ewan Tempero

Technical Report CS-TR-96/2
January 1996

Abstract

Understanding software reuse and reusability is an important factor in effective and efficient software development. We believe that reuse and reusability should be regarded as key issues right from the start, and propose teaching how to program with an early and explicit focus on reusability. Our approach is based on a conceptual model of reusability which explains how features of programming languages can be used to create reusable software. This paper outlines our approach, and discusses our experience.

1 Introduction

Creating reusable software and achieving significant software reuse are both difficult, and there are many barriers that have impeded progress. Much work has been devoted to understanding the issues and exploring strategies to improve the situation. In fact, a wide range of strategies seems to be indicated, spanning low-level technical issues of program design to high-level managerial issues of project analysis and planning. (See [9] for a broad discussion of these issues.)

Increasingly it seems software reuse and reusability should take a central place in the study and practice of software engineering. Our suggestion is that reuse and reusability should be a central theme when we teach programming in the first place, both to provide a coherent practical explanation of programming technology, and to foster a positive attitude to software reuse for the future. As educators, we feel this is a contribution we can make at an early stage toward better integrating software reuse and reusability into the practice of software engineering.

In the past, there were opportunities to discuss the importance of reuse, for example in discussing procedure libraries or class libraries. However, there were several problems with this approach. To start with, students found it difficult to focus on reuse when they are still coming to grips with the concepts of design and the features of a programming language. Moreover, the advice we gave students about reuse is often in the form of vague guidelines that really require experience to effectively apply.

Both reuse and reusability are important, but in this paper we address this issue of reusability specifically. 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 so as to improve software development. However, we 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.

This paper is an explanation and discussion of our proposal. Section 2 explains our model, which is both the conceptual foundation for our views, and the basis of our exposition to students. Section 3 shows how we use the model to explain the pivotal concepts of modern programming in terms of software reusability. In section 4 we discuss our experience in using this approach so far, and outline possibilities for further exploration. Finally, we present our conclusions.

2 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 [5].

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),

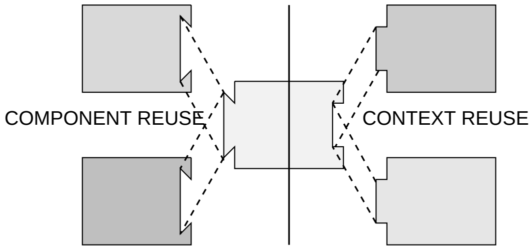

Figure 1: *Component Reuse, as shown on the left of the diagram involves several different contexts invoking the same component. Context Reuse, as shown on the right of the diagram, involves one context invoking different components using the same interface.*

- 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. 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 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.

customisability: The customisability of a component is the amount of control a context has over how the context dependencies are met. Similarly for the customisability of a context.

For example, parameters of different kinds (values, types, functions) provide different ways for the caller of a function to dictate how the function behaves to meet the caller's requirements.

checkability: The checkability of a dependency is the degree to which the fact that the dependency has been met can be checked.

The requirement that the parameters to a procedure be a certain type is a dependency. Some languages do not check that the types of the actual parameters are correct at all, some check only at run-time, while others check at compile time.

Since some dependencies are beneficial to reusability, it follows that programmers will deliberately introduce such dependencies into the code they write. We therefore find it useful to classify dependencies according to *intention*.

contract: Contract dependencies are those that are deliberately introduced to the segment.

non-contract: Non-contract dependencies are those that exist but are not intentional, either by error or because the language provides no way to exclude them.

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.

explicit: An explicit dependency is one that can be described by constructs in the programming language.

A contract example is the *import* clause in a Modula-2 module. This indicates that the context has dependencies on the module.

A non-contract example is a type definition in the *interface* section of modern versions of Pascal, since representation details that the programmer may want hidden are in fact exposed to any user of the module. Because these details are exposed, the user can directly access them, resulting in a context dependency.

implicit: An implicit dependency is one that cannot be described by constructs in the language, but whether it has been met can be automatically checked by the environment (such as compiler, linker, loader).

A contract example is any assumption that may be made on the argument types of a C++ template. For example, the template implementation may depend on one of the argument types having a specific function (such as `print`). There is no language support for making this dependency explicit. However, it will be determined at link time if this dependency is not met.

Before `static` external data was introduced to C, variables declared global to a compilation unit had to be global to all compilation units, even if access was only needed from within the compilation unit. This is an example of an implicit non-contract dependency.

informal: An informal dependency is one that cannot be described by constructs in the language, and nor can it be easily checked.

A contract example is a list implementation that is supposed to maintain items in order. Any client of the implementation will depend on these semantics being met, however in most traditional languages this cannot be checked. This is changing as more languages include constructs for specifications.

dependency between	context component
beneficial effect on reusability	checkability customisability usability
intention of dependency	contract non-contract
language support for dependency	explicit implicit informal

Table 1: Model summary

An iterator for a set implementation may always produce elements of the set in a specific order (such as order of insertion). Should any context assume this order from the iterator, then that context would have an informal non-contract dependency.

Earlier we defined reusability in terms of the number of contexts or components that could *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 [8].

Table 1 summarises our model.

3 Teaching with the Reusability Model

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

In this section we discuss common aspects of program design and features of common programming languages. Although we believe the principles we discuss are valid in general, we acknowledge that many of the points we make arise from consideration especially of the imperative programming paradigm, and those languages we have used in our teaching: Pascal, Modula-2, C, and C++.

3.1 Procedures

Our approach assumes that students understand the primitive elements of programming such as variables, expressions, statements, sequences, selection and iteration. (However we are considering beginning even earlier. See section 4.) 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 then to other key concepts in program design as outlined below.

Procedures: Component and Context Reuse

To explain procedures, we can introduce them as a way to allow a segment of executable statements to be written in one place, and invoked from many others. Interestingly, this most basic reality of procedures has in recent years sometimes been missed by beginners because of the emphasis on the role of procedures in top-down design. It is also a good idea to point out explicitly that procedures can call other procedures.

In terms of our model, this use of procedures emphasises component reusability. This suggests that we should be careful to avoid dependencies the component has on the context, so that the component can be invoked by different contexts. In practical terms, this motivates discussion of procedures using global variables from the context, and how this limits component reusability.

On the other hand, the symmetry of our model means we should also consider context reuse. We can explain that with regard to procedures, context reuse involves new procedures being created to match existing calls. We can introduce the concept of separate compilation at this point, and also the concept of procedure libraries, both of which support context reuse.

Consideration of context reuse should also include the importance of avoiding dependencies the context has on the component, so that the context can invoke different components. In practical terms, this motivates discussion of scope, which disallows contexts from using names from within procedures, thus supporting context reuse.

In our model, we suggest that in general dependencies are detrimental to reusability, but describe two cases where they are beneficial: for customisability, and for checkability.

Procedures: Customisability

In explaining the role of procedures in both component and context reuse, the issue of customisation quickly becomes evident. While procedures appear to solve the problem of avoiding repeated work, beginners quickly learn that they seldom want to reuse *exactly* the same procedures in different contexts, but that procedures need to be somehow tailored to fit new situations.

We demonstrate how global variables, despite earlier warnings, can be used to customise procedures and then show how they are not ideally suited to the task. Firstly, uses of global variables are *implicit* dependencies, and this is a drawback for reusability because programmers will find it difficult to determine what a context must provide in order to reuse the procedure. Secondly, uses of global variables are dependencies on *names* which are in fact irrelevant to the procedure's behaviour. This is drawback for reusability because reuse is only safe where names inside and outside the procedure do not clash — something that is difficult to plan for.

At this point we can introduce parameters for procedures as a beneficial form of dependency specifically for customisation, and supported by most languages. We can explain how parameters address both the weaknesses involved in using global variables. Where a context uses parameters instead of global variables, the dependencies are explicit, instead of implicit. Accordingly, a programmer can easily see what a context must provide in order to reuse the procedure. Moreover, now the component depends on the context to supply a *value* to the parameter, and external names are properly irrelevant. Finally, we can distinguish between pass-by-value and pass-by-reference as deciding whether the value dependency is only by the procedure on its caller, or both ways.

Procedures: Checkability

Our model emphasises reusing components in different contexts, and contexts with different components, but acknowledges that there is a need to ensure the integrity of any invocation. Especially with experience of parameters, students should at this point appreciate that the whole approach only works properly when contexts invoke the components they intend to, and where dependencies are indeed met. This motivates the need for dependencies to better provide checkability.

We can start by noting that checkability is supported by matching the name from a call to a procedure heading. The really important dependence of a context on a component is the informal dependence on semantic matters that are difficult to check. However, the introduced dependency on the name is a simplification that improves checkability.

In practice checkability becomes more of an issue where procedures have parameters, since this requires the matching of actual and formal parameters. We can use this opportunity to introduce the role of type systems. We explained parameters as a dependency involving values, but checkability of values is likely to be informal and involve run-time checking at best. We can address this by introducing dependencies on the types of parameters: this is less detrimental to reusability than dependence on names, but more checkable than dependence on values.

3.2 User Defined Types: Component and Context Reuse

So far we have discussed segments of executable statements — procedures — how they foster reusability, and how they can be made to better support reusability. At this point in teaching programming for reusability, we can introduce segments of data declarations — user defined data types. Teaching experience suggests that the issues in understanding data declaration are more subtle than those in understanding executable statements, and so we hope to gain by having introduced reusability of executable statements first.

We begin by presenting only simple user defined types like those in original Pascal. These data types allow a programmer to describe a data structure once, but use it to declare data as necessary. User defined types are similar to procedures, in that they enable reuse of data declaration segments in a way similar to how procedures enable reuse of executable statement segments. As was the case with procedures, user defined types can also use other user defined types — this is sometimes called composition. We emphasise composition because we have noticed some beginners fail to appreciate its importance.

In terms of our model, this presentation is again emphasising component reuse. Accordingly, we must again be careful of dependencies of the component on the context, because these will limit the contexts where the component can be reused. In the case of data types, a practical implication of this is that data types should avoid using constants or other types from the context.

As with procedures, the symmetry of our model again suggests we consider context reuse. We can explain that with regard to data type declarations, context reuse involves new data type declaration segments being used with existing code that declares data to be of that type.

For context reusability, we must be careful to avoid dependencies of the context on the component, because these will limit the components which can be used with the context. In the case of data types, a practical implication of this is that contexts should not rely on the internal details of the type implementation. With this motivation, the time is right for us to introduce encapsulated data types, usually called classes in object-oriented programming.

Classes typically allow their internals to only be accessed by a set of designated procedures, and so context code is forced to use these procedures to work with objects of the class. With such programming language support, we make it clear how dependencies of the context on the component can be avoided or controlled, thus better promoting context reusability.

User Defined Types: Customisability

When becoming familiar with procedures, even beginners rapidly see the need for customisability. When becoming familiar with user defined types, the need does not seem so obvious so quickly. However, there are two common situations that do motivate understanding of the need.

The first motivating situation concerns types that involve some size, but where different contexts require a different size to be used. In some cases, the size is used at run-time and can be provided by using parameters of interface procedures, thereby accomplishing type customisability via procedure customisability. However, in many compiled languages, some sizes need to be known at compile-time, and that strategy does not work. Another approach is to use global constants, but that has the drawbacks discussed with regard to customising procedures using global variables. Moreover, it typically restricts types to being customised in one way in one program.

The second motivating situation concerns "container" types, where the containing structure is significant, but the type of the contained items is not. This need can again be addressed by using global type names. For example, a `list` type may depend on a `list_item` type for the items in the list, but the `list_item` type may be left to be defined by the context. However, the drawbacks are the same as above: implicit dependence, dependence on names, and typically a limitation to one customisation per program.

At this point we are ready to parallel our presentation of procedures, and introduce parameterised types. By using parameterised types (such as C++ template classes) we can make it clear that the dependencies can be made explicit, and the dependencies can avoid involving type names from the context. Moreover, we explain, because there is no dependence on context type names, this approach can be used to allow a number of different customisations of a data type, all in the same program.

User Defined Types: Checkability

In our presentation of procedures, we explained how dependencies were beneficial to reusability if they assisted in making sure that calls correctly matched procedures. Especially with regard to matching actual and formal parameters, we showed that the type of parameters was beneficial in exactly this way.

At this point we should address the issue of checkability with regard to user defined types. In fact, the points to make are initially the same. With a simple user defined data type, the name is an introduced dependency that stands in for informal dependencies and promotes checkability. And with parameterised types, the need for matching context and component implies that dependencies on the type of the parameters would be beneficial in the same way as is the case for parameterised procedures.

So far, our discussion of checkability has concerned matching names (for procedure or type), and matching types (for parameters). A user defined type is the mechanism that introduces a type name into a program, and students may feel the picture is complete.

However, we should now point that our introduction of encapsulated data types means that matching types need not mean matching names. We can explain that because encapsulated data types are used via the procedures that are associated with them, this set of procedures (and their parameters) can be an alternative way of type checking.

We should now address the issue of what type checking should mean. While strict name matching between context and component does support checkability, this checkability may

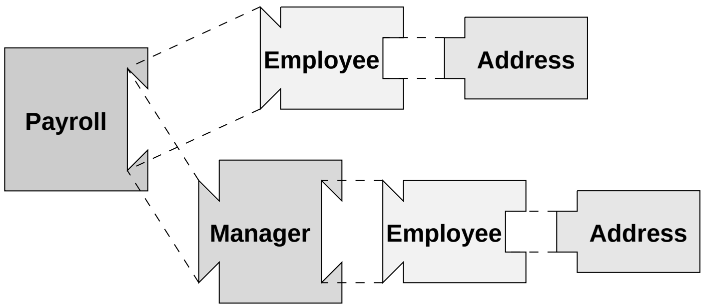

Figure 2: *This jigsaw diagram illustrates how interface conformance makes inheritance different than composition. An "employee" class can reuse an existing "address" class by composition (top), but composition produces a different interface for employee than for address (see tabs on top part of diagram).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

be stricter than necessary. In particular, we can introduce an alternative way of supporting checkability, where context and user defined type must agree not on the name, but on the way the access procedures are to be used. Because this would allow different user defined types to be used with a context, this would better support context reuse.

This is the way we suggest introducing inheritance and polymorphism.

User Defined Types: Inheritance and Polymorphism

We advocate care in introducing inheritance and polymorphism, because widespread publicity about object-oriented programming has resulted in many students being keen to use these techniques before they really understand them. We do believe that inheritance and polymorphism are best explained in terms of reusability. However, we feel the subject is best left late in the teaching sequence, when students already have a good understanding of both composition and type checking.

Following from the discussion of checkability above, we can introduce inheritance by emphasising the role of *interface conformance*. The technique allows a user defined type to have a set of interface procedures that *conform* to the set of interface procedures of another type by having at least the same set of procedures that are used in the same way.

We can then explain how inheritance does also work like composition, in that it allows the type to reuse the data declarations and procedures of the base type. So we do acknowledge that inheritance and composition are similar in this way. However, we continue to emphasise that it is interface conformance that is the distinguishing characteristic of inheritance. Figure 2 illustrates the "jigsaw" diagrams we use to make clear the difference between inheritance and composition.

We introduced inheritance after looking closely at checkability. Checkability is a beneficial use of dependencies, because it ensures the integrity of invocation. Simple type name checking does ensure this, but can be too strict. We can now explain that interface conformance can thus increase reusability. It does this by allowing context code to invoke either the base class or a derived class, because it implies that instances of the derived class may be used anywhere instances of the base class may be used. Thus with inheritance, we can use a context with several different classes, even in the same program. This is the primary connection between inheritance and reusability: *inheritance supports context reusability.* Figure 3 shows how we illustrate this key relationship.

We can then go on to discuss how sometimes context code depends on the interface of

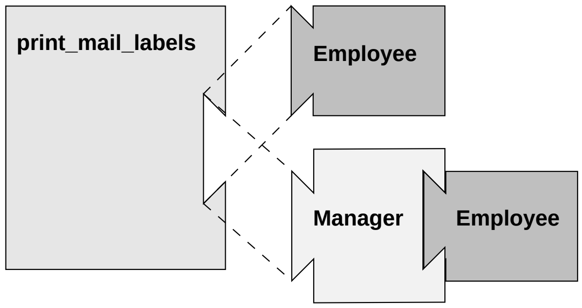

Figure 3: *This jigsaw diagram illustrates reusability of context code through interface conformance. Because the interface of the manager class conforms to that of the employee class, objects of the manager class can be used in place of objects of the employee class in the context code. Accordingly, context code, such as code to print mailing labels, may have been created to work with objects of the employee class, but may be reused without modification for objects of the manager class.*

a base class, but not on its exact behaviour. In this case the behaviour of the derived class can differ from the base class, and yet still be used by the same context code. This is our explanation of polymorphism: where context code is used with different classes that conform to one interface, but where each has differing behaviour.

We can also outline the role of abstract classes: classes with an interface, but no behaviour. Their advantage is that context code can be written in terms of an 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 the sections above, 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. In the next section we discuss implications of our approach, and our experience in applying it so far.

model	**procedures**	**data types**
component reuse	procedure reuse	data type reuse
context reuse	procedure call reuse	data declaration reuse
customisability	global variables	global constants and types
	procedure parameters	parameterised types
checkability	name and type checking	name and type checking
	interface conformance	inheritance and polymorphism

4 Discussion: Implications and Experience

In this section, we discuss how our approach restructures a number of concepts involved in learning to program. In particular, we argue that several new connections are made between concepts usually regarded as distinct. We claim this is true not only at the detail level, but also at higher levels affecting the way programmers work. Moreover, this restructuring does not merely affect abstract understanding, but assists in practical programming by leading to guidance about how to design programs and use programming language features for maximum advantage.

Later in the section, we describe our actual experience so far in using our proposed teaching approach, and outline some plans for future consideration.

4.1 Learning about Inheritance

The topic that initially focused our attention on this topic, both as a research opportunity and an opportunity to develop our teaching, was the widely acclaimed connection between object-oriented programming and code reuse (for examples, see [4, 7]). We felt that although many learners appreciated the claims in theory, many had difficulty in designing programs in practice. In particular, learners often had difficulty in understanding when to use inheritance, and even more difficulty with multiple inheritance.

As described in the previous section, our model and our approach address the role of inheritance in a novel way. We have been using this approach to explain inheritance in our object-oriented programming courses, both to second year university students, and to programmers in industry. Our experience has been very encouraging, and indeed some students who had previous exposure to other explanations have expressed relief at finally understanding what it was all about. However, the issue is a subtle one.

Many teachers and writers focus on inheritance as a technique for reusing base classes, and students have no difficulty understanding that. However, many students (and some teacher and writers) seem to stop at that point. They typically do not consider that ordinary composition supports reuse in a very similar way, and they seldom recognise the key aspect to be interface conformance, nor do they see how it supports what we call context reuse.

Moreover, many writers 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 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].

4.2 Learning about Encapsulation

Whereas we believed that inheritance was a concept that needed to be taught and learned better, we initially felt the topic of encapsulation was typically well explained and understood. However, we did feel that encapsulation, both of procedures and of data types, did play a role a significant role in making code reusable, and so we addressed encapsulation in our model. In doing so, we ourselves began to better understand the role of encapsulation, even though we had previously thought we understood it perfectly well.

While our goal was to explain encapsulation via reusability as a way of maintaining a focus on reusability, we in fact do more: by focusing on reusability, we explain encapsulation better than we used to be able to. There are two aspects to this.

Firstly, there is the relationship between maintenance and reuse. Maintenance, and various management aspects relating to it, used to be our main concern when explaining encapsulation. We believe this to be the standard explanation: encapsulation forces a separation of concerns, so that development or maintenance can be carried out on separate units of a program, knowing that integrity will be preserved. This is a sound principle, and as true as it ever was. However, we now point out that this can be seen as an issue of reusability. Instead of concentrating on parts of the program that are changing, we concentrate on the parts of the program that stay the same. Although maintenance involves changing some parts, it involves keeping — or *reusing* — other parts.

Secondly, there is the relationship between the reuse and the encapsulation. We believe that traditional explanations of encapsulation, both of procedures and of classes, concentrate on the part that is encapsulated. However, as we have outlined in the previous section, encapsulation typically promotes the reusability and reuse of the context code, not the component code. As a result, our explanations result in a different focus of attention. When considering or evaluating encapsulation, we argue, the critical issue concerns the reusability of the *context* code. If that is worthwhile, then encapsulation is worth considering; if that is well supported, then the encapsulation is doing its job.

We started out believing that traditional explanations of encapsulation were fine, and that we and our students understood it all. However, we now suspect that our decisions and evaluation concerning encapsulation were too vague. With our new approach to explaining encapsulation, not only do we maintain a focus on reuse and reusability, we also feel we are giving more practical and more specific advice about when and how to use it. We feel that the students are becoming better programmers by better understanding reusability.

4.3 Learning about Type Checking

An issue related to encapsulation is that of type systems and type safety. Again, this is something that we had not felt was poorly understood, and we feel our new approach is an improvement. While students had previously understood the importance of type safety as important for preservation of integrity, we suspect that they may have had too philosophical an approach to understanding types.

For example, we recall many discussions with students concerned about whether a class was "really" a stack, or a list, or map, or whatever. Our new approach to explaining types concentrates on their role in providing checkability between context and component, so benefiting reusability and reuse. This has the result that students become less concerned the philosophical aspects of the nature of types, and more concerned with the engineering matter at hand.

4.4 General Experience and Future Plans

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 during this time, 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 earlier [1].)

We are suggesting that this approach is a way of making software reuse and reusability a central issue right from the start. However, we must admit that so far we have only used the approach with students who have already learned the fundamentals of programming beforehand. In our university work, the students have learned Pascal, and done some programming "in the small". In our professional courses, the students have typically done a limited amount of programming, often some time ago. In either case, we assume their knowledge of simple programming concepts, and start afresh on the pretext of teaching object-oriented programming for reusability.

Although we have not used this approach with complete beginners, we are interested in exploring this, and speculate that it might work well. A number of even primitive concepts of programming have strong connections with reusability. For example, repetition might be regarded as a kind of reuse of the statements repeated; selection might be regarded as a kind of customisation. This kind of presentation might provide an early foothold for consideration of reusability, and soon it would be time to introduce procedures and user defined types, whereupon our model would apply.

On the other hand, we also suggest it would also be worth using our model at more advanced levels of learning. We have had some experience in doing this in teaching graduate courses explicitly surveying the object-oriented paradigm. However, we feel it would also be useful in more practical courses in software engineering, common in the final year of many university programs in computer science. In such courses, reuse is often explicitly recognised, and our model could be more fully explained, and used to survey how reuse is supported by programming languages. However, we also feel that the practical orientation of the model would also help in facilitating the evaluation and assessment so important in software engineering, both of tools to be used in development, and in the software being developed itself.

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.

So far, we are satisfied that our efforts have been worthwhile. However, we do recognise that our approach has one weakness that stems from novelty itself. While many students with some experience of object-oriented programming find our approach significantly illuminating, others are confused that we don't echo the books and magazines they have read. In some cases these students are themselves having difficulties applying what they've 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 continual focus on reuse and reusability, 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.

5 Conclusion

We have suggested that reusability can be made a primary focus while teaching how to program. Our approach to doing this involves an explicit model for how reusability is supported. Our model involves both contexts and components, and the dependencies between them. We concentrate particularly on how some dependencies benefit reusability by supporting customisation and checkability, despite other dependencies affecting reusability adversely. With our model, we can explain how the key features of programming languages really work, and explain how they can be used to create reusable software. Our experience in exploring this method of teaching has been positive.

Further development is needed both for our teaching approach and the conceptual model upon which it is based. In particular, we intend to address more conceptual design issues involving domain analysis into our model, and experiment with how this can be introduced to beginning programmers. Such analysis has long been a part of higher level software engineering, and has also become an important basis for user-interface design. However, these issues are typically neglected or avoided when programming is being learned in the first place, and we are concerned that this may lead to problems later on.

We believe that there are several factors involved that make our approach worthwhile. Firstly, we think that although many techniques and programming language features that support reusability have been developed over the years, there is a need to further develop a principled way of discussing the common goals and common themes involving reusability. We intend our model to address this need. Secondly, we feel that many guidelines to beginning programmers are too vague, and not sufficiently operational enough to make it clear how programming goals can actually be achieved with the tools available. Finally, we hope that by explicitly recognising reusability as a key goal in program design and construction, we will influence beginning programmers to keep focused on both reusability and reuse throughout both their education and professional work in software engineering.

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 ACM 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] Gerhard Fischer. Cognitive view of reuse and redesign. *IEEE Software*, 4(4):60–72, 1987.

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

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

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

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

[9] Will Tracz. *Confessions of a Used Program Salesman: Institutionalizing Software Reuse*. Addison-Wesley, 1995.