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 the impact of language features on reusability

Robert Biddle and Ewan Tempero

Technical Report CS-TR-95/17
September 4, 1995
(Revised January 12, 1996)

Abstract

We present a *conceptual model* for helping us understand the nature of software reusability, particularly to help us understand how language features affect the reusability of software. The fundamental concept for our model is that of *dependencies*. We identify properties of dependencies between segments of code that are important to reusability. We validate our model by showing its application to well-understood principles of reusability. We demonstrate that being able to describe these principles in a single framework allows us to gain a better understanding of reusability.

Keywords: reusability, conceptual models, programming language support, software micro architectures

Publishing Information

This paper appeared in the proceedings of *The Fourth International Conference on Software Reuse*, Orlando, Florida, USA, April 23-26, 1996.

1 Introduction

Creating reusable software is difficult, and there are many barriers that impede progress. Much current research concentrates on domain analysis and domain specific architectures especially designed to facilitate later reuse of components. (See [22] for a broad discussion of these issues.) We are concerned with the smaller scale issues involved in what Hollingsworth and Weide refer to as *micro-architecture* — the detailed structure and behavior of components in software systems [5]. Our concern comes from the observation that the code being written today is the legacy code of tomorrow. We believe avoiding the problems of today's legacy code requires a good understanding of micro-architectural issues.

We are particularly interested in the impact of programming languages on reusability. 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" [11]. Thus, to create reusable code, we must understand how languages affect reusability. To this end, we have been developing a principled way of thinking about reusability.

The history of science shows that the process of understanding phenomena is often facilitated by constructing a *conceptual model*, a set of fundamental concepts to help describe, discuss, and explore the nature of the phenomena. Early in the process, such a model is useful because it establishes a vocabulary with which to describe observations of the phenomena. Because these observations are being described with the same vocabulary, it is easier to identify recurring themes. The next step is to base predictions on these themes, which in turn can be used to refine (or replace) the model. Our goal is to create a conceptual model of reusability.

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 discuss the future.

The previous work most similar in character to ours is the "3Cs" model [4, 7, 21]. 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.

There has been much work that has concerned "good programming style", that is, insightful guidelines about how to write more reusable code. Even some early discussions of general software design and design methodologies (for example [15]) identify issues similar to those directly addressed by our model. Some work has resulted in informal maxims [6, 22], and others in more formally stated (but not enforced) rules [8]. Some guidelines have been developed further and given rise to new language features and disciplines with support tools and formalisms [9, 17]. All these guidelines have been developed from experience, and so provide a basis on which we can validate our model.

There have been other attempts to describe reuse and reusability with a focus on the programming language level. However, many are typically concerned with creating *metrics* for measuring either past or potential reuse of code. Poulin gives a summary of metrics that apply to reusability [16]. These metrics have all focused on how to *indicate* how well a segment of code can be used in a new situations, rather than our goal of describing how to *create* more reusable software.

There have also been various discussions about how particular languages or language features improve reusability (see for examples [11, 12, 13, 18, 19]). Earlier expositions of our model [1, 2] also concerned one programming language (C++). However, we intend the model to span programming languages, and believe a contribution of our model is to show that many issues can be helpfully discussed within a single conceptual framework.

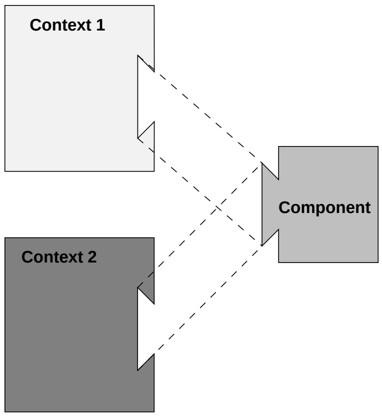

Figure 1: *A simple structure for understanding reuse and reusability: the common case is where several contexts reuse a component.*

In this paper, we describe our approach to modeling the effect of programming language features on the reusability of code. When building a model, it is important to establish exactly what is to be modeled. We do so in section 2. The fundamental concept for our model is that of *dependencies*. We discuss dependencies and their properties, and then introduce our model in section 3. To be useful, any model must at least describe current knowledge. In section 4, we apply our model to various language features. Finally, we present our conclusions.

2 Foundations

To build a model of reusability, we need to be clear about what we mean by "reuse" and "reusable". 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 between them. 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 [6]. We believe this confusion has misled some programmers into thinking they are creating reusable code when in fact what they are doing is reusing code.

To understand how code can be made more reusable, we must look at how code might be reused, which in turn means understanding what is meant by code reuse. 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 code reuse are:

- 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 reuse depicted in figure 1. The figure shows three segments of code, By "code", we mean source-level statements in some programming language. Programming languages provide a variety of ways of grouping statements together, such as juxtaposition, unnamed and named blocks, classes, modules, and namespaces. We use the term "segment" as a general name for such groupings.

In the figure, two segments are related in that at some place or places one somehow uses or *invokes* the other. The effect is as if the second segment of code has 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 in figure 1. Sometimes however, 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, 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*.

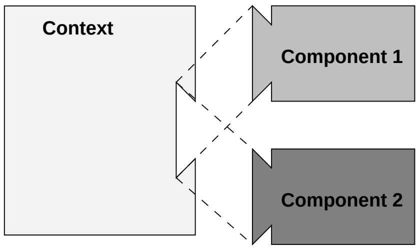

Figure 2: *Context Reuse: in this case one context invokes different components using the same interface.*

Context and component are really *roles* played by segments of code. 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.

Giving even an informal definition of reusable software is difficult. Tracz defines reusable software as "software that was designed to be reused" [22]. Micallef describes a reusable component as one that "can be combined, adapted and modified to fit in a new application in ways unforeseen by the implementor of the component" [13]. While these definitions appeal to intuition, they are of not much practical benefit since they give no advice on how to create reusable software. More importantly from our point of view, they do not directly explain how a language feature may support or inhibit the reusability of software.

We define reusability in terms of components and contexts. We say that a language feature increases the reusability of a segment of code if it increases the number of contexts from which it can be usefully invoked (as a component) or if it can increase the number of components it can usefully invoke (as a context). This definition has the advantage of being fairly objective, though it is still incomplete. In particular, it does not take into account the effort required to do the reuse. For example, some forms of reuse require extra support code that may not be needed for other approaches. Nevertheless, it provides a good foundation on which to base a discussion of reusability.

3 The Model

In this section we present our model. As we introduce each concept, we give simple examples to illustrate it. A more detailed discussion follows in the next section.

The fundamental concept for our model is that of a *dependency*. 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*. For example, a parameter to a procedure represents a dependency by the procedure on the caller supplying a value for the parameter.

There are two kinds of dependencies between any context and component: those that the component requires the context to meet (component dependencies) and those that the context requires the component to meet (context dependencies). For example, the parameter example above is an example of the former; an example of the latter is when the context's call to the procedure assumes that it takes that parameter. In this case, the two dependencies match, resulting in a successful invocation.

For an invocation of a component by a context to be a success, both the component and context dependencies must be met. Thus, the reusability of a component is directly related to its ability to satisfy the dependencies of a possible context, and on that context's ability to satisfy the component's dependencies. This means that to describe the reusability of a segment, we must understand how different dependencies affect reusability.

We are not the first to study dependencies. However, we believe it is seldom realized that not all dependencies are bad. For example, as discussed above, parameters to procedures represent dependencies, and everyone agrees that parameters are important for increasing the reusability of a procedure. Accordingly, we begin by looking at aspects of dependencies that have a beneficial effect on reusability.

There are three aspects relevant to dependencies that have a beneficial effect on reusability: checkability, flexibility, and customizability.

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

For example, 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.

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

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.

flexibility: Flexibility is a property of the language. A language that introduces dependencies that do not contribute to the behavior of a segment is not as flexible as it might be.

For example, with Pascal arrays there is a dependency between the type of the array and the size of the array, which reduces flexibility. At the other extreme, the Unix loader just checks that all the required symbols have been defined, even though the definitions may not be what are expected. This is very flexible, although it has other problems.

The flexibility and customizability of a segment together represent the *generality* of that segment.

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

contract: Contract dependencies are those that are deliberately introduced to the segment, either as context or component dependencies.

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.

Our interest is in how language features affect software reusability, and we have found it useful to classify dependencies according to the support provided by the software development environment. Note that each category can be divided into contract and non-contract dependencies.

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

dependency between	context component
beneficial effect on reusability	checkability generality (= flexibility+ customizability) usability
intention of dependency	contract non-contract
language support for dependency	explicit implicit informal

Table 1: Model summary

as more languages include constructs for specifications.

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.

In the previous section, 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. Although the usefulness of a code segment is a micro-architecture issue, it is less affected by the programming language, and we do not discuss it in this paper. Research in human-computer interaction has explored the concept of "usability" and has resulted in improved understanding of many issues in user interface design [14].

Table 1 summarizes our model.

4 Applying the Model

In this section we survey a set of language features that have supported reusability, and use our model to explain how this support is accomplished. This demonstrates the validity of the model, showing that it is consistent with mechanisms that are known to support reusability. Moreover, it also demonstrates that the model assists us in better perceiving common patterns and structure in these mechanisms.

4.1 Procedures

Procedures allow a segment of code to be written in one place, and invoked from many others; procedure libraries allow this benefit to to be realized on a larger scale. In most languages, the procedure structure itself allows a segment of code to be named, and the call mechanism allows context code to invoke the procedure by name.

The name is the basis of an explicit dependency of the procedure segment: it is a dependency because in order to use the procedure it must be invoked using that name, and it is explicit because the programming language supports the association of the name with the segment. Calling a procedure also involves an explicit dependency: the caller depends on a component with the called name being available, and the language has a mechanism for indicating that a procedure is being called. The context code also depends, typically implicitly, on anything that the procedure depends upon. For example, procedures can call other procedures, and the context depends on these as well.

Where a procedure is called, checkability is usually supported by the name matching. In some languages (such as early C), that is all that is done, allowing the possibility of a procedure being called with parameters of the wrong types. Other languages include more sophisticated checks. Some languages (such as early Pascal) require that the implementation of a called procedure be available when the call is being translated. This is a dependency between the procedure name and its implementation that is imposed by the language. Other languages remove this dependency, allowing any procedure matching the name and informal semantics. This flexibility is the basis for separate compilation, and this in turn is the basis for procedure libraries. This flexibility allows new procedures to be created to match existing calls, thus supporting context reuse.

Procedures and Scope

The name space of the context may be shared with the procedure. To the extent that it is, this allows implicit dependencies both ways on the names in the shared "global" space. The context can set or retrieve values associated with the "global variable" names that are also used inside the procedure, and so affect the procedure's behavior. These dependencies allow customization, thereby increasing the generality of the procedure.

The implicit nature of these dependencies is a drawback for reusability, because it may not be immediately clear what a context must provide in order to reuse the procedure. The checkability of these dependencies concerns whether the names used in the context and the procedure are compatible, but informal compatibility can only be checked at run-time. Type systems, however, are very useful in allowing some checking at compile-time. Such type checking really involves introducing a new, easily checked, dependency on the type, to stand in for the informal dependencies.

Customization through use of global variables involves dependencies on *names*, and this is another weakness. Reuse is only safe where the name spaces do not clash, and this is difficult to plan for. Many languages have facilities to limit how much of the name space is shared between contexts and procedures, allowing control over which names can be involved in dependencies. Some facilities (nested scope, for example) allow control only one way, disallowing dependence by the context on names in the procedure. Moreover, even where the dependencies are clearly limited to names used in customization, the dependence is still on the names themselves, rather than on the values.

Procedures and Parameters

Most languages with a procedure mechanism also provide for parameters, specifically to support customization. Where a context uses parameters instead of global variables, the implicit dependencies are replaced by an explicit dependencies. Moreover, now the component depends on the context to supply a *value* to the parameter, and external names are irrelevant.

Pass-by-value parameters can be seen as involving dependencies of 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 dependencies are only in one direction. On the other hand, pass-by-reference parameters also introduce dependencies by the context on the component — the context now relies on the component changing the value of the formal parameters in the "expected" way.

The checkability of these dependencies can only be assured in the same limited ways as before, and informal dependencies on the values cannot be statically checked at all. However, type checking has again proven very useful in allowing some consistency checking between actual parameters and formal parameters.

4.2 User Defined Types and Classes

User defined types allow a programmer to describe a data structure once, but use it to create as many instances as are necessary. User defined types enable data declaration reuse in a similar way to how procedures enable control code reuse. Data declaration details are distinguished as a named component that is invoked by a context in order to declare data. The context is explicitly dependent on the availability of a type of that name, and implicitly on anything that the type itself depends upon. For example, the technique of composition (also called aggregation) involves types described using other types — the context implicitly depends on these as well.

Reusability of types involves important issues concerning the name space. There are broadly two issues: the dependence of the context code on names in the type, and the dependence of the type on names from the context.

Classes and Encapsulation

In a simple user defined type (like those in original Pascal) the context must have access to the parts of the type, because there is no other way the type can be used. This access means the context will be dependent on the internal structure of the type, and this dependence has consequences detrimental for reusability. In particular, the context code cannot directly use a type component with a different internal structure.

For example, if a list type is to be changed from using an array to using a linked structure, the context must likewise be modified. The problem here is that the context has implicit dependencies on on the implementation details of the type. If the context is indeed concerned with these, reusability is constrained. If, however, the context is only concerned with some aspects of the type, then it should depend only on those aspects. This increases reusability of the context accordingly, because more components can be used with that context.

This problem is addressed by encapsulated user defined types, usually called classes in object-oriented programming. In this approach, the names internal to a type may only be accessed by a set of procedures associated with the type. These then become the interface of the type, and context code is then explicitly dependent on the procedures, instead of implicitly dependent on the type internals. The procedures can then be designed to support longer term context reusability, and the type component can be changed to one with a different internal structure when desirable, because contexts can no longer depend upon the internal structure. In this way, encapsulation supports context reusability.

Genericity

A type description may depend on type names or constants from the context. This is similar to a procedure using global variables from the context. The implicit nature of these dependencies is a drawback, as a programmer must explore the internal structure in order to determine what must be provided by the context in order to reuse the type.

A common reason user defined types depend on names from the context is to increase generality. This is especially important for "container" classes, where the containing structure is significant, but the type of the contained items is not. 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. This supports greater reusability, but it is limited in two ways.

Firstly, the dependence is again on the *name*, and so reusability can be made difficult by name clashes. Secondly, the generality is limited by the way types work in many languages. To allow early type checking, many languages limit flexibility and require names and types to be bound at compile-time. This means that context names can be used to make user-defined types customizable, but only once in a name space. For example, a `list` type may allow customization via a `list_item` type, but `list_item` can only be bound once.

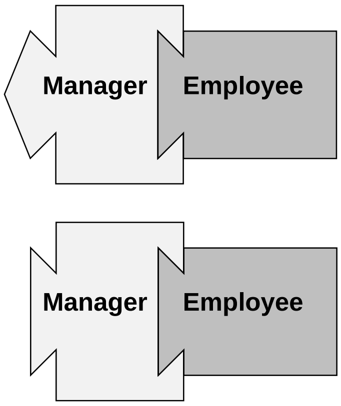

Figure 3: *This jigsaw diagram illustrates interface conformance. A "manager" class can reuse an existing "employee" class by composition (top) or inheritance (bottom). However, composition produces in a different interface (see tabs on top diagram), whereas inheritance can produce an interface that conforms to the employee class interface (see tabs on bottom diagram).*

For procedures, customization can be better achieved by using parameters, and the same is true for user-defined types. By using parameterized types (such as Ada generics or C++ template classes) the dependencies can be made explicit, and the dependencies can avoid involving type names from the context. Moreover, because there is no dependence on context type names, this approach can be applied in languages where types are bound early.

Inheritance

Languages differ in details about inheritance, but we intend our remarks in this section to concern common approaches. As with composition, inheritance allows a context class (the child) to reuse an existing class (the parent) as a component. What is different about inheritance is that this affects the interface of the new class: the interface *includes* the interface of the parent class.

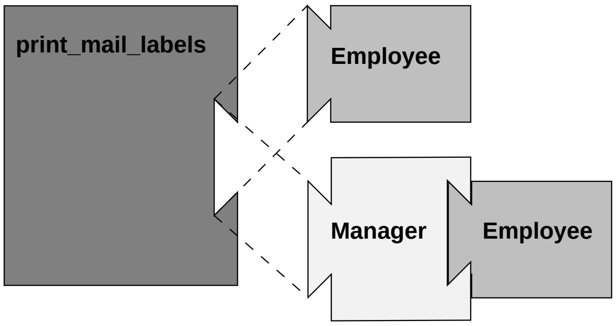

Figure 4: *This jigsaw diagram illustrates reusability of context code through interface conformance. Because the interface of the manager class conforms to that of the employee class, objects of the manager class can be used in place of objects of the employee class in the context code. Accordingly, 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.*

Figure 3 illustrates this distinction.

For reusability, this *interface conformance* is the important aspect of inheritance. Interface conformance implies that instances of the child class may be used anywhere instances of the parent class may be used. The context code has a dependency on the interface of the parent class, but because child classes conform to this, they are also acceptable. Thus with inheritance, we can use a context with several different classes, even in the same program. This flexibility increases the reusability of the context code. Figure 4 illustrates this key relationship between interface conformance and reusability.

Polymorphism

Sometimes the context depends on the interface of the parent class, but not on its exact behavior. In this case the behavior of the child class can differ from its parent, and yet still be used by the same context code. This is an explanation of *polymorphism*: where context code is used with different classes that conform to one interface, but where each has differing behavior. Polymorphism allows a name to refer to more that one segment during the execution of a program. This removal of a dependency increases the flexibility of the language. In a similar way, propagation patterns [9] can also be seen as increasing flexibility. Increased flexibility, however, raises the issue of whether checkability can be supported sufficiently for reusability. One approach to this is to involve specification of behavior in the type system [10].

An important use of inheritance and polymorphism concerns *abstract* classes: classes with an interface, but no behavior. The 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.

Abstract classes are themselves the basis of object-oriented frameworks [6]. 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 lead to "context libraries".

4.3 Discussion

In the sections above we have detailed a number of language mechanisms that support reusability, and so help programmers create reusable code. In order to demonstrate the validity of our model, we have explained in terms of the model how each of these mechanisms support reusability. We feel that the model did provide the necessary low level structure and nomenclature that made the explanations possible, and so conclude the model is reasonable. In fact, we feel that the explanations in terms of the model demonstrate that using the model sheds new light on the subject, and so claim the model is a significant contribution toward better understanding reusability.

In some cases, the model simply explains in new ways principles that are well understood. For example, programmers have known for years that procedures are better customized with parameters than by using global variables. The model addresses one obvious aspect of this because global variables involve implicit dependencies, whereas parameters are explicit and so more easily managed.

However, the model goes further and addresses deeper structural aspects of reusability support. To stay with the same example, the model makes it clear that another key issue in comparing global variables with parameters is that global variables involve a dependency on the name, where only a dependency on a value is really required. Moreover, the model makes it clear that type checking of parameters is really introducing a new dependency on the type in order to ease checkability. This also shows there is a tradeoff between checkability and generality.

The symmetric layout of context and component in the model also allows significant relationships to be highlighted. While discussion about reuse and reusability usually concerns code in the component role, the model draws us to consider context reuse. For example, nested scope rules often disallow contexts from accessing internal names of a procedure. Because this ensures that a context cannot depend upon the internals of a procedure, this means procedure implementations can be changed freely. This is context reuse, and so we see nested scope supports context reusability rather than component reusability.

At a higher level, the model also makes clear the strong similarities between different kinds of mechanisms. For example, procedures and classes are very different mechanisms, but our descriptions showed many of their characteristics to be the same. In particular, the importance of context reuse was emphasized in both. For example, encapsulation is shown to support context reusability by ensuring that there are fewer non-contract dependencies from the context on the implementation of a class, so allowing different implementations to be used with the same context. This parallels the point made about procedures and scope.

More significant is what the model makes clear about inheritance and polymorphism. This is the primary connection between inheritance and reusability: *inheritance supports context reusability*. This observation has important consequences because it inspires guidance about when to use inheritance, and guidance about how to use it. Briefly, the guidance is: use inheritance when it will be useful for context code written for a parent class to be reused with child classes. This has influenced us both in our own design work, and in how we explain inheritance and polymorphism to students [3].

The sections above have mentioned only the main aspects of a selection of very well known mechanisms. However, we have done similar analysis for a variety of other mechanisms, and have found the approach usefully illuminating. In particular, the observations about inheritance also apply to multiple inheritance. Moreover, the principles of the model are also useful in discussing other kinds of mechanisms involved in reusability, such as exceptions and automatic memory management. We are also interested in more sophisticated programming language facilities, such as support for concurrency [20], and hope to use the model to gain new insight about the implications for reusability.

5 Conclusions

We have presented a conceptual model for helping us understand the nature of software reusability, particularly to help us understand how language features affect the reusability of software. Our model focuses on the two roles played by segments of code, *context* and *component*, and the *dependencies* between them. We then introduced the significant properties of dependencies. Some (explicit/implicit/informal) classify language support for describing them, some (contract/non-contract) represent the intention of the programmer, and some (checkability/flexibility/customizability) classify how dependencies beneficially affect reusability.

Any useful model of a phenomenon should at least provide a way to describe observations of that phenomenon. Our model does allow us to describe well-understood principles of reusability. In doing so, we have improved our understanding of these principles; for example, we now realize that reusable software must include reusable contexts, as well as reusable components. A model is especially useful if it is simple enough that recurring themes can be identified. The symmetry we have observed between context and component is an example of such a theme.

We continue to develop and validate our model. In particular, it seems worthwhile investigating how to incorporate the concept of "usability" from research in human-computer interaction. We also continue to use our model to analyze various mechanisms and strategies that support reusability. We are interested to see how the model relates to other current ideas for creating reusable software, such as adaptive software and open implementations. Eventually, we would like to extend our model beyond language features, and use it to guide the development of reusable software. We believe our model already provides useful advice:

- consider both component and context reusability,
- be aware of dependencies, and make them as explicit as possible,
- avoid non-contract dependencies, and those that have no benefit for reusability,
- seek out contract dependencies that are beneficial to reusability, and
- consider carefully the tradeoff between checkability and customizability.

References

[1] Peter Andreae, Robert Biddle, and Ewan Tempero. How to reuse code, really: A guide to creating reusable code with C++. In *Proceedings of the 13th New Zealand Computer Society Conference*, Auckland, New Zealand, August 1993. New Zealand Computer Society. Also available as Technical Report CS-TR-93/6.

[2] 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. Also available as Technical Report CS-TR-93/12.

[3] 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.

[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] Joseph E. Hollingsworth and Bruce W. Weide. One architecture does not fit all: Micro-archi-tecture is as important as macro-architecture. In *Proceedings of the Seventh Workshop on Institutionalizing Software Reuse*, August 1995.

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

[7] 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.

[8] Karl J. Lieberherr and Ian M. Holland. Assuring good style for object-oriented programs. *IEEE Software*, pages 38–48, September 1989.

[9] 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.

[10] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[11] 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.

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

[13] Josephine Micallef. Encapsulation, reusability and extensibility

in object-oriented programming languages. *Journal of Object-Oriented Programming*, pages 12–36, April/May 1988.

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

[15] D. L. Parnas. Information distribution aspects of design methodology. In *Proceedings of the International Federation for Information Processing Congress*, pages 339–344, Ljubljana, Yugoslavia, 1971. North-Holland.

[16] Jeffrey S. Poulin. Measuring software reusability. In William B Frakes, editor, *Proceedings of the Third International Conference on Software Reuse: Advances in Software Reusability*, pages 126–138. IEEE Computer Society Press, November 1994.

[17] Murali Sitaraman and Bruce Weide. Special feature: Component-based software using RESOLVE. *ACM SIGSOFT Software Engineering Notes*, 19(4):21–67, October 1994.

[18] Alan Snyder. Inheritance and the development of encapsulated software components. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object Oriented Programming*, pages 165–188. The MIT Press, 1987.

[19] Lynn Andrea Stein. Delegation is inheritance. In *1987 Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, pages 138–146, 1987. Published as ACM SIGPLAN Notices, 22(12). December 1987.

[20] Ewan Tempero. A concurrency model for object-oriented languages. In *Proceedings of the 12th New Zealand Computer Conference*, pages 91–102, August 1991. Also available as Technical Report CS-TR-91/5.

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

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