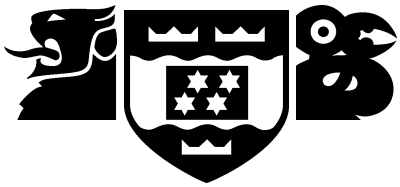
VICTORIA UNIVERSITY OF WELLINGTON



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

Modeling units of reusability

Robert Biddle and Ewan Tempero

Technical Report CS-TR-96/19
1 November 1996

Abstract

When talking about software reuse, a problem that very quickly becomes apparent is "what is being used?". In this paper, we introduce the concept of *assembly*, a unit of reusable software. This allows us to more sensibly compare different strategies for software reuse. We believe this concept is scalable to all levels, from intra-program reuse to inter-program reuse. In particular, it allows us to unify two forms of reuse, composition reuse and generative reuse, that have traditionally been regarded as distinct. This is part of a model we are developing for describing how software reuse is supported by software development systems.

1 Introduction

Software reuse covers many topics, from large scale issues such as domain analysis to smaller scale issues such as the detailed structure of components. Also, there are different issues involved in design *with* reuse when distinguished from design *for* reuse. Our work primarily concerns design for reuse, at the smaller scale level. While larger scale issues are important, in the end code must still be created. We feel a better understanding of basic principles for creating reusable code is necessary in order to avoid code being created now becoming legacy problems in the future.

Ultimately our interest is in reducing the difficulty and cost of creating reusable code. There have been many proposals and discussions about how to do this, from guidelines for its creation [8], to more formally stated rules [10], to new language features [11], to disciplines with support tools and formalisms [14]. There are many ideas, and it is difficult to understand how they are related, and exactly how they truly improve the development of reusable code. To help us relate the different ideas, we have developed a model that helps us understand how language features impact the reusability of code [4]. We have found the model useful in guiding both our research and our practical design work; we have also found it very useful in our teaching [2, 3].

Our model has concentrated on providing insight about reusability of components, discussing aspects of the "component role" and "context role" played by units of code. In this paper we examine the nature of a unit of code, and introduce the idea of an "assembly" to generalise several kinds of related structures.

The term "component reuse" is sometimes used to enable a distinction with another approach to reuse: "generative reuse". Typically component reuse involves direct reuse of code; generative reuse involves indirect reuse of code by transforming the code in some way for the reuse. Discussion about these two approaches to reuse has been largely separate, even while acknowledging importance of both approaches [9]. As well as allowing refinement of existing aspects of our model, assemblies allow the model to encompass not only "component reuse", but also "generative reuse".

In this paper, we describe assemblies and show how they are used to explain generative reuse. The next section summarises our model to date. Section 3 discusses the issues related to describing code fragments and introduces the assembly concept. Section 4 explains generative reuse in terms of our new model and section 5 presents our conclusions.

2 Model

Our model is intended to reflect the kind of code arrangements depicted in figure 1. On the left of the figure, two code 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

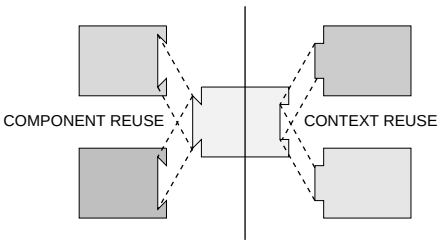


Figure 1: *Component Reuse, as shown on the left of the diagram involves several different contexts invoking the same component. Context Reuse, as shown on the right of the diagram, involves one context invoking different components using the same interface.*

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.

> A dependency between two segments of code is a condition that one segment meets so that it can use the other segment, or so that the other segment can use it.

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, flexibility, 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.

flexibility: Flexibility is a property of the language. A language that introduces dependencies that do not contribute to the behaviour 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.

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

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.

The flexibility and customisability 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. 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.

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

Table 1: *Model summary*

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 the compiler, linker, or 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.

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.

```
class AClass {
  void member1() {
    function1();
  }

  void member2() {
    ...
  }
};

void function1() {
  // ....
}

void function2() {
  AClass c;
  c.member2();
}
```

Figure 2: *Kinds of segments*

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 [13].

Table 1 summarises our model.

3 Assemblies and dependencies

The concept of dependency has proved useful for thinking about the reusability of a piece of code, however we have found that our definition is not precise enough for some of the detailed examinations we want to do. The problem with this definition is that it is based on terms that are not sufficiently well defined, specifically "segment" and "use".

3.1 "Segment"

Consider the example in figure 2.

This example shows (at least) two kinds of "segments": C++ functions and C++ classes, and because one kind can use the other kind and vice versa, our definition of "segment" must capture both kinds. This raises the complication of dealing with non-contiguous code. With C++ classes, it is quite common to use to files to describe a C++ class — a class definition (in a ".h" file) and the declaration of its member and friend functions (in a ".cc" file). And in principle, each member function can be declared in separate files. This non-contiguity applies even in languages that do not attempt to separate interface from implementation, such as Eiffel or Java. The use of inheritance in these languages (and this is true of all object-oriented languages) means that the definition of inheriting child classes will be non-contiguous. And if you consider Smalltalk classes, the physical

relationship between the different parts of a class is not clear at all.

We do not want to limit ourselves to just functions and classes either. We also want to be able to refer to such things as C++ namespaces, Java packages, or GenVoca realms [1]. We may need to distinguish blocks, even "basic blocks" or arbitrary lines of code. To represent all of these ideas, and to avoid the implication of contiguity implied by "segment", we introduce the concept of an *assembly*. An assembly is any set of descriptions of source-level code. This allows us to describe any pattern that associates pieces of code together, and, since one possible description of source-level code is an assembly, the description can be hierarchical. The description of an assembly is the sequence of source-code-level statements associated with it. If the descriptions of two assemblies do not overlap, we say those assemblies are disjoint.

A *subassembly* is one part of an assembly. For the purposes of discussion, it is convenient for subassembly to only refer to "top-level" parts of an assembly, that is, an assembly is exactly the combination of a set of disjoint subassemblies. Note that there may be more than one way to divide an assembly into disjoint subassemblies.

3.2 "Use"

Consider the C++ example in figure 3.

This example shows code in the class `Parent` being referred to in various different ways. The classes `Child` and `Composite` demonstrate references that have traditionally been labelled as "reuse", namely through inheritance and composition (a.k.a. aggregation), and so much be considered as forms of "use". But what about class `User`? Which of its references to `Parent` constitute a "use"? References `4` and `5` begin to illustrate the difficulty — is a declaration of a variable of some type a "use" of that type? One could argue that in the case of `4`, code will be executed (the constructor `Parent()`). But that is not true of `5`, and nor is it true of languages like Eiffel or Java, where the constructors must be explicitly invoked, as has happened at `6`. Furthermore, such declarations are not strictly necessary, as is shown by languages such as Smalltalk or Self. What both `4` and `5` are doing, is specify that those variables will only be used for "Parent-like" activities in the scope of their declaration.

Reference `3` looks similar to `4`, in that it is both a declaration, and the invocation of a constructor. However, it is possible that the code that is executed is not code that was written by the implementor of `Parent`, but is an automatically-generated copy constructor. If the notion of "use" is limited to references that cause code to be executed, should it only apply to code written by a programmer? `8` and `9` are similar to `3`, in that the constructor for `Parent` that gets executed may not be one that was written by the programmer, however the involvement of `Parent` in these cases follows directly from the "uses" `1` and `2`, and so it is not clear that they should be treated as separate from `1` and `2`. On the other hand, if we only consider situations where code is executed, then it would follow that neither `1` nor `2` could be considered "uses".

Now consider the consequences of taking the strict line that only situations that result in programmer-written code being executed. For example, `7` definitely results in code associated with `Parent` being executed. However, does it make sense to say this is a "use" of `Parent`, when only *some* of the code associated with `Parent` could possibly be executed in this case? Even it this is considered "use", the situation is more complicated with the remaining cases. For example, `11` does not appear to involve any Parent code, but it is quite possible it does, as is illustrated by `12`. In the case of `12` and `13`, there is apparent reference to `Parent`, but in fact it is possible that no `Parent` code actually gets executed. And just to complicate matters more, `15` does look like `Parent` code being executed, but in this case it is possible that no such code is involved.

Next we have to understand how assemblies interact. As with any code, both control and data can pass between assemblies. Control can pass between pieces of code because they are adjacent, through an explicit procedure call, or through some other language mechanism (such as invoking Prolog rules).

```
class Parent {
public:
  Parent() { ... }
  void parent() { ... }
  void overridden() { ... }
  virtual void polymorphic() { ... }
protected:
  int parent_bit;
// ...
};
class Child: public Parent {       // 1, inheritance
public:
  Child() { ... }
  void child() { ... }
  void parentuse() { ...
    parent_bit = 0;
    parent();
    ...
  }
  void overridden() { ... }
  virtual void polymorphic() { ... }
};
class Composite {
public:
  Composite() { ... }
private:
  Parent p;                        // 2, composition
};
class User {
public:
  void member(Parent arg) {        //3, parameter
    Parent p;          // 4. Declaration + constructor
    Parent* pp;        // 5. Declaration only
    pp = new Parent() // 6. explicit construction
    p.parent();        // 7. Member function call
    Child c;           // 8.
    Composition comp; // 9.
    c.non-virtual();  // 10. Parent code
    c.child();         // 11. Child code
    c.parentuse();     // 12.
    c.overridden();    // 13. Member function call
    c.polymorphic();  // 14. Member function call
    pp = new Child;
    pp->polymorphic();// 15. polymorphic call
    useit(arg);        // 16. passing on a value
  }
};
```

Figure 3: *Difference uses of class* `Parent`

Data can pass because the assemblies share a name space, through an explicit mechanism such as parameter passing, or because memory is shared.

Assemblies can also interact because one uses information associated to the other. For example, in figure 3, the assembly `member` makes use of assembly `Parent` to specify what properties its argument should have. As is shown by `16`, it is possible that the argument is not directly used by `member` but is instead passed on to some other function. However there still remains a link between `member` and `Parent` through the argument. In this case, neither control nor data passes between the assemblies, however there is a reference of one by the other.

3.3 Refining the model

Each of the ways assemblies interact represents a *use* of one by the other. This use has directionality, namely the direction that the control or data passes, or the reference to one by the other, so we refer to one assembly *using* another. The assembly doing the using, is the *context*; the used assembly is the *component*.

In order for a context to use a component, conditions have to be met by one or both assemblies. Any such condition is a *dependency*.

Dependencies will usually apply to both the context and component. The context will have to meet certain conditions before it can correctly use the component (such as pass parameters of the correct type) and the component will also have to meet certain conditions before the context will want to use it (such as have member functions with the right name).

Determining the dependencies associated with one assembly is not just a matter of looking at how it is used by another assembly. There may be dependencies it could meet if needed, but which the specific use being examined does not need (for example, a `Stack` ADT implementation the provides a `Top` operation but which is not always used). Thus for a single assembly, it is necessary to consider all possible uses of it to determine its dependencies.

3.4 Application

In this section, we demonstrate our model by applying it to various aspects of programming languages. While we do not claim to be saying anything new here, we believe our model allows a more concise and precise description of programming language features, which ultimately allows a more principled discussion of programming languages. We begin by looking at some general features found in most programming languages, namely *scope* and *encapsulation*, and then we look at a language feature in specific language, namely *C compilation units*.

Scope and Encapsulation To examine the distinction between *scope* and *encapsulation*, we first need to introduction the notion of a *symbol*. The symbols of an assembly is the names (or identifiers) that are defined as part of its description. Often, one of these symbols will be the name of the assembly itself. For example, in object-oriented languages with the concept of class, each class will have a name. We use "symbol" rather than "identifier" because we want to deal with the case when an identifier is defined more than once (for example in overlapping scopes). We regard these as distinct symbols. The language rules for disambiguating multiple uses of the same identifier dictate which assembly the symbols belong to.

The *scope* of an assembly is the set of assemblies that can directly interact with it. This set will often be described in terms of the conditions that must be met in order to refer to it. For example, in Java, a non-public class may only be referred to by other classes in the same package.

The *encapsulation* of an assembly is the set of its symbols that may not be accessed by other assemblies. Here, “accessing” by an assembly means that the symbol is mentioned somewhere the assembly’s description. Thus, encapsulation restricts the scope of symbols. It may be that the mention

is qualified, in that the assembly to which the symbol belongs is mentioned as well, or it could be unqualified, such as is the case in overlapping scopes. What the set of symbols is sometimes depends on the other assembly involved. For example, in Smalltalk, instance variables in a class may be accessed by any subclass of that class, but not by any other class.

Some language mechanisms can add more assemblies to the scope of assembly. For example, the scope of classes in Java is limited to the package in which they are declared. The `import` statement is a mechanism that adds the assembly (package) in which is appears to the scope of the target of the `import` statement.

C compilation unit The description of a C compilation unit is the sequence of C-source statements in a physical file. Any decomposition of a unit would be acceptable subassemblies, but a useful one is each function definition being a subassembly and the remaining definitions (global variables, structs, and macros) together in one subassembly.

The symbols in the unit consist of the names of the subassemblies as well as the symbols of the subassemblies (such as automatic and static variables). Note that the compilation unit itself has no symbol associated with it. What the symbols look like may depend on the context. Assemblies not written in C may have to refer to the symbols using different identifiers than was used in the compilation unit.

The scope of a C compilation unit is any other compilation unit (not necessarily written in C) that may be, after it has been compiled, linked with the unit.

The symbols that name functions and non static global variables have the same scope as the compilation unit in which they are declared, and so are accessible to any other compilation unit. All the remaining symbols are encapsulated within the compilation unit.

4 Component Reuse and Generative Reuse

So far all our discussion about support for reusability has concerned what is often called "component reuse". This term is usually used to enable a distinction with another approach to reuse: "generative reuse". Typically component reuse involves direct reuse of code in the form it exists; generative reuse involves indirect reuse of code by transforming the code in some way for the reuse. Although generative reuse is a more complex approach, it is regarded as making reuse possible in circumstances where simple component reuse would be difficult, or tedious, or both. However, the simplicity of component reuse can be an advantage, because it allows more direct support in programming languages and environments.

Many discussions of programming and reuse concern only component reuse, perhaps because of the direct support possible [9]. Many discussions of generative reuse focus on sophisticated applications where component reuse would be inadequate. In this section we explain how our model and the concept of assemblies address not only component reuse, but also generative reuse, and thus enables better understanding of the relationships between the two approaches.

There are many different domains in which generative reuse has been successful. For example, many early systems for creating graphic user interfaces were based on generators: input took the form of graphical or textual descriptions of graphical elements, and output was complicated code in some programming language, with many calls to a graphics library. Perhaps the most widely used form of generators are compilers for programming languages. The input to a compiler is a program in some high-level source language, and the output is a program in some low-level target language, typically machine code. In between these two examples are many others. Parser generators such as YACC, for example, require a grammar as an input specification, and the transformation system computes the automaton corresponding to the grammar, and adapts a generalised parser which it then produces as output.

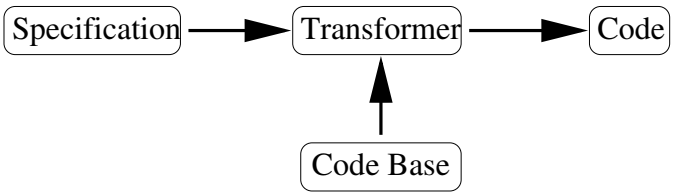


Figure 4: *Generator structure, showing basic elements.*

Many discussions about generators focus on dealing with the input specifications and the process of transformation. In the case of compilers, this seems reasonable because the input specification is complex, and requires lexical analysis and parsing. However, there is another critical element involved, because neither the input specifications nor the transformation algorithm can create the necessary output code. Involved somewhere are *code fragments* that can form the basis for the output code, subject to transformation according to input specifications.

For example, the GUI generators mentioned above used a library of code for graphical elements that formed the basis of the output program. Even compilers use a set of fragments in the output language that can form the basis for target code generation. In some compilers these fragments may be interwoven with the transformation program, and indeed the transformation program may itself be structured by the parsing of the input language. Regardless, the set of fragments do form a critical element in the compilation process.

In general the field of generative reuse, or reuse by transformation, is very diverse, and also includes translators from one language to another, and human-assisted automatic programming systems. Feather [7] surveys and classifies various approaches to program transformation, and the relationship between program transformation and reusability has been discussed for some time [5, 6, 12].

The process involved in generative reuse is thus typically regarded as involving four elements, as shown in figure 4. The input consists of specifications, and the output consists of usable code. The generator takes the input specifications, accesses the code base, and produces the output code. In terms of our model, we see the code base as a set of assemblies. The assembly concept covers a range of code structures, and different generators involve different points along this range. The different kinds of assembly used in generators correspond to other differences in generators.

For example, in simple generators used for GUI creation, assemblies might be complete code components, and the specification input might simply select which were necessary. The transformation would simply involve selection of the components from the set, and copying them for output. As a next step, assemblies might be sections of code with textual formal parameters, and the input might select sections, and specify actual parameters to be substituted. The transformation would then involve the basics of macro-processing, where the code sections were selected, substitution done, and the result output. Eventually we reach the compiler situation, where the assemblies are more like templates for very small fragments of code, and the format for input specification is so complex we call it a language. The transformation system must involve lexical analysis and parsing for the language, and then generate code by selecting and tailoring the code fragments as necessary.

Throughout this range, code is always being reused, and the significance of the transformation increases. The last step heavily depends on the transformation, and without it the small fragments would be difficult and tedious to reuse. The first step, on the other hand, involves very little transformation, and is essentially no different to component reuse. As a limiting case at one end, we might also include the situation where the assembly is an entire program, and input specification is unnecessary: in this case transformation is reduced to simple copying. While we have been focusing on one transformational step, it is usually the case that the output of one transformation can be used as input for another. As a limiting case at the other end, we might also include the situation where instead of the resultant code being output, it is directly interpreted: in this case transformation is as

level	specification	code base	transformer
no reuse	empty	program	copy
simple component reuse	selector	components	selective copy
component reuse	selector + parameter values	parameterised components	selection + parameter substitution
full generative	program	fragments	compiler

Table 2: *Ranges of characteristics of generator elements.*

complex as general computation. Table 2 summarises the range of possibilities.

In terms of our model, the prime benefit of generative reuse concerns generality. Because the transformation process can modify the assemblies in arbitrarily complex ways, the generality of the assemblies can be increased as desired, and with fine precision. In a way, the generality possible means that generative reuse resembles "white-box" reuse, because the transformation system has access to the internal structure of the code.

As well as improved generality, generative reuse also allows related improvements in checkability. As the transformation of the code is carried out, it is possible to do sophisticated checking of explicit and implicit dependencies between assemblies. Sophisticated checking requires a more complex transformation system, however, and it will be more difficult to check dependencies involving invocation beyond the set of assemblies involved at transformation. Moreover, some informal dependencies will involve run-time invocation, and so cannot be checked at an earlier time of transformation.

Finally, there is typically also a strong connection between generative reuse and usability. As the level of selection and customisation increases, the input specification must necessarily become more complex. Where this input is generated by human programmers, this complexity can easily become burdensome. However, the syntax of the input specifications can be made more sophisticated, becoming a "language" corresponding to a higher level of abstraction, and more suited for human use. This input language can be processed by the transformation system, driving the transformation of the code assemblies into output code.

These strengths of generative reuse can be seen all working together in a compiler for a programming language. Generalised fragments of code are reused over and over in differing circumstances as the transformation system generates output code. Dependencies between such fragments can be well checked by the transformation system, but there are limits concerning invocations of components beyond the system, and checks that must be made at run-time. The input specifications can be made more usable by introducing a syntax suited to the application domain, and suited for human programmers: a programming language. Of course, programming languages themselves introduce a variety of code assemblies to support reusability, and so generative reuse is not only based on component reuse, but ultimately also supports component reuse as well.

5 Conclusions

We have been developing a conceptual model to help us understand the relationship between the many different proposals for improving design for reuse. An important issue in the development of such a model is how to specify exactly what code is being "reused". We have refined our existing model by introducing the concept of *assembly*, which represents a unit of reuse.

Assemblies have proven useful for describing the interaction between different pieces of code, such as scope and encapsulation. However they also provide a way to unite two forms of reuse traditionally regarded as distinct: component reuse and generative reuse. This makes it easier to see the common structure of the different kinds of reusable units, and the tradeoffs between the different approaches.

Having a model that covers a wide variety of forms of reusability helps improve our understanding of reusability by providing a common vocabulary with which to relate the different forms. It can also help us to identify recurring themes, and possibly determine gaps in our knowledge. It also provides a principled way to describe concepts in reusability, which we have found especially useful for teaching and in evaluating software design.

References

[1] Don Batory, Vivek Singhal, Jeff Thomas, Sankar Dasari, Bart Geraci, and Marty Sirkin. The genvoca model of software-system generators. *IEEE Software*, pages 89–94, September 1994.

[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. Teaching programming by teaching reusability. In *Proceedings of Reuse'96: An Integral Part of Software Engineering*, Morgantown, West Virginia, USA, April 1996.

[4] 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.

[5] J. M. Boyle and M. N. Muralidharan. Program reusability through program transformation. *IEEE Transactions on Software Engineering*, 10(5), 1984.

[6] T. E. Jr. Cheatham. Reusability through program transformation. *IEEE Transactions on Software Engineering*, 10(5), 1984.

[7] M. S. Feather. A survey and classification of some program transformation approaches and techniques. In *Proceedings of IFIP WG2.1 Working Conference on Program Specification and Transformation*, Bad Tolz, Germany, 1986.

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

[9] E. Karlsson. *Software Reuse: A Holistic Approach*. Wiley, 1995.

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

[11] 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.

[12] J. M. Neighbours. The draco approach to constructing software from reusable components. *IEEE Transactions on Software Engineering*, 10(5), 1984.

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

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