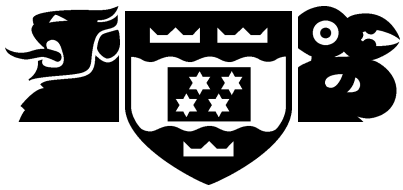
VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui



School of Mathematical and Computing Sciences
Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 471 5341
Fax: +64 4 495 5045
Internet: Tech.Reports@mcs.vuw.ac.nz

Towards Tool Support for Reuse

Robert Biddle and Ewan Tempero

Technical Report CS-TR-97/7
November 1997

Abstract

Many approaches are being pursued to improve the effectiveness of software development. Two important approaches are the development of *tools* to aid the programmer and *reuse* of code to reduce the programmer's work. The Renata Project at VUW has begun investigating combining both approaches. In this paper we report on our experience in developing tools that directly support the reuse of code in software development.

Publishing Information

This is a version of a paper that appeared in the proceedings of *Software Engineering: Education & Practice (SE:E&P'98)*. Dunedin, New Zealand, 26-29 January 1998.

Towards Tool Support for Reuse

R. L. Biddle and E. D. Tempero
School of Mathematical and Computing Sciences
Victoria University of Wellington
New Zealand
{Robert.Biddle|Ewan.Tempero}@mcs.vuw.ac.nz
http://www.mcs.vuw.ac.nz/comp/Research/renata/

Abstract

Many approaches are being pursued to improve the effectiveness of software development. Two important approaches are the development of tools *to aid the programmer and* reuse *of code to reduce the programmer's work. The Renata Project at VUW has begun investigating combining both approaches. In this paper we report on our experience in developing tools that directly support the reuse of code in software development.*

1 Introduction

Many approaches are being pursued to improve the effectiveness of software development. Two important approaches are the development of *tools* to aid the programmer and *reuse* of code to reduce the programmer's work. Both approaches have led to modest improvements in productivity, but not to the degree hoped for. We believe that one reason for this is that achieving effective reuse is a difficult problem in itself, one that requires more support than has generally been available. We propose the development of tools designed to explicitly support the reuse process. We are investigating what kind of support tools may be able to provide for reuse, how to build such tools, and are developing prototypes of tools to test our ideas. In this paper, we discuss the approach we are taking and our progress to date.

The rest of this paper is organised as follows. The work we describe here is the current emphasis of the Renata project at Victoria University of Wellington. The first stage of the Renata project involved developing a model of reuse. We believe having such a model is necessary for the principled development of reuse-based tools, and so we summarise our work to date in Section 2. We also discuss what tools can do to improve productivity in general, and in particular how they might help improve the application of reuse to software development. Our work has not been completely theoretical; we have already been working on the development of reuse-directed tools to test and refine our ideas. We discuss the results of this work in Section 3, and then present our conclusions.

2 Tool support for Reuse

Our goal is to investigate tools to provide direct support for reuse. Our approach is to use a *model* of reuse to guide our investigation. We use our model to identify those aspects of reuse that are important to making reuse effective, and so help us to concentrate our efforts on tools that are likely to be productive. In this section, we summarise our model and then use the model to identify some potential tools.

2.1 Understanding Reuse

In order for reuse to take place, something must be reused. The things being reused are often referred to as "reusable assets". There are two aspects of dealing with reusable assets: the production of the assets, or developing *for* reuse, and the process involved in reusing them, or developing *with* reuse.

Reusable assets can come from any part of the software lifecycle, from requirements, to analysis, design knowledge, or test suites. In our work, we are currently focusing on reuse in the creation of *source code*. There are two standard reuse approaches to creating source code, *building block reuse*, which involves reusing existing source code, and *generative reuse*, which involves automating the creation of source code. These are usually discussed separately, although we believe there is a fundamental connection between the two [2]. For simplicity however, we will restrict our discussion here to building block reuse.

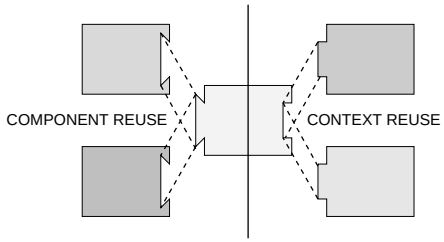


Figure 1: *Component Reuse, as shown on the left of the diagram involves several different contexts invoking the same component. Context Reuse, as shown on the right of the diagram, involves one context invoking different components using the same interface.*

2.1.1 Modeling Reuse

Our model focuses on units of reuse, which we call *assemblies*. An assembly is anything that may ultimately become valid source code. This includes simple entities such as functions, classes, or modules, but also more complex entities such as all public methods of a class, or parts of a macro definition.

Our model is intended to reflect the arrangement of assemblies depicted in figure 1. On the left of the figure, two assemblies are related in that at some place or places one uses or *invokes* the other. The effect is as if the second assembly has somehow been inserted into the first. In such situations, we call the invoking assembly the *context* and the invoked assembly the *component*. Each different context that invokes a component represents a use or reuse of the component. For example, a function may be called from different places. We call this *component reuse*. Furthermore, each different component that is invoked by the same context, as shown on the right of the figure, represents a use or reuse of the context, or *context reuse*. For example, in object-oriented languages, one message send may result in different methods being invoked because of polymorphism.

The main goal of developing our model is to understand what features of an assembly improve or reduce the likelihood that that assembly will be reused, that is, what features improve or reduce the assembly's *reusability*. Context and component are really *roles* played by assemblies. As the figure shows, an assembly can play both roles. This observation, together with the notion of context reuse, shows that any discussion of reusability must encompass both component and context.

When considering the reusability of an assembly, there are two aspects of the assembly that are relevant. One is whether or not the assembly is likely to be used again, its "usefulness". The other is whether or not there are aspects of its implementation that would prevent it from being used again. This second aspect depends on the expectations the context and component have of each other, which we call *dependencies*. A significant part of our model involves classifying dependencies, and understanding how the different categories of dependencies affect reusability.

The concepts of assembly, context, component, usefulness, and dependency are the foundation of our model, and are all that are necessary for our discussion here. More detail is available elsewhere [1, 2, 3, 4].

2.1.2 Developing for Reuse

Developing for reuse means making a conscious effort to increase the reusability of assemblies when they are created. Our model suggests which aspects of an assembly we need to concentrate on to achieve this. Exactly when this conscious effort should take place can be complicated by uncertainty of the usefulness of any new assembly, so it will often be the case that efforts to improve reusability will be applied to already existing assemblies. Given already existing code, the process for developing reusable assemblies will look something like the following.

First, coherent assemblies must be extracted from the existing code. Exactly what constitutes a coherent assembly will be language dependent. For example, individual methods can not normally exist independently from classes in Java, whereas C++ does allow functions to exist outside of classes.

Once assemblies have been determined, a decision must be made as to whether they are likely to be useful enough to be worth the effort involved to improve their reusability. They will then be analysed to determine what roles they can play, and what dependencies apply to them. Each of these steps can benefit from some form of tool support. We discuss two early attempts in section 3.

2.1.3 Developing with Reuse

Developing with reuse means using existing assemblies when creating new ones. The technical factors associated with developing with reuse can be broken down as follows. This outline has been partially adapted from Mili et al. [8].

If something is to be used, then it first must be *identified*, thus the cost of the *search* process affects whether or not reuse takes place. If searches almost always take a long time, then the developer may prefer to develop the needed assembly from scratch.

The cost of searching includes the cost of formulating the *query* and the *accuracy* of the result, and there is usually a tradeoff between the two. A query that returns 100 (or even 10) matches still leaves some work to be done before an acceptable assembly is identified. A more sophisticated query may return a more accurate answer, but it will likely increase the time taken to formulate the query (which may

involve learning a query language), as well as the search time itself.

Another part of the identification process is *verifying* that the assembly is acceptable. Sometimes this can be dealt with by a suitable query that precisely specifies what is required. Really, this should be part of the accuracy of the search, but we make this distinction since it is usually the case that, even when the search process has returned a very good match, that the developer will want to assure herself that the assembly really is the one needed.

Once an acceptable assembly has been identified, it must then be *retrieved*. Retrieval of software is not the same as retrieval of physical components — it may not be necessary to make a copy of anything (although it may be, if cut-and-paste is the only option), but there may be some work required to make the assembly available for reuse. For example, the developer may need to add directories to a search path (or even figure out what the appropriate directories are), add the appropriate tags to the source file (such as an `import` statement in Java source), or even install a class library.

Once a potential assembly has been identified and retrieved, it must then be *adapted* to the new environment. In the case of a 100% match, the cost of adaptation will be zero, otherwise a new assembly will be created, based on the retrieved assembly.

Finally, the assembly resulting from the adaptation process must actually be *used*. This will typically involve describing the assembly in the source code (usually by giving its name), but it may also involve some effort in converting the information at hand to that expected by the assembly, or making sure that the information passed to the assembly is as expected. For example, the assembly may expect one of its arguments to be a language specific "string" type, but all that is available is an array of characters, or the assembly may expect two integers, in which case the developer must ensure that the integers available are passed to the assembly in the correct order.

Whether or not reuse takes place depends on whether the cost of the reuse process described above is less than that of developing the needed assembly from scratch. Complicating this analysis is that programmers, particularly inexperienced ones, are poor at accurately estimating the time to develop assemblies from scratch. Thus it will not be enough that reusing an assembly costs two thirds of that of developing it from scratch if the developer underestimates the cost of developing from scratch by half!

2.2 Supporting Reuse

We expect tools to enhance our ability to perform tasks, and we can use our model to identify tasks associated with code reuse that may profit from tool support.

One task that is clearly relevant to code reuse is the development of assemblies. It would be nice to have a tool that could churn out reusable assemblies just by turning a crank. Unfortunately, writing code has a large creative component, and we cannot expect tools to improve creativity. Nevertheless, there are aspects to writing code that can benefit from tool support; just as CAD tools remove some of the tedium of dealing with producing blueprints, so too can tools make writing code easier. Many such tools exist, including text editors, compilers, debuggers, test systems, revision control systems, and configuration management tools. These tools are used to create any code, not just reusable assemblies, but we believe they can be enhanced to provide direct support for code reuse.

Our discussion above indicates tasks that are specific to producing code that is intended to be reused. Our model identifies assemblies and their dependencies as important concepts, thus it would be useful to have tools that help identify and classify these concepts.

Increasing the reusability of an assembly requires more effort than just creating it in the first place. If the assembly is not going to be used again, then we do not want to invest any more resources in its development. Thus, it would be helpful to have some indication as to how useful an assembly is.

Ideally we would like to be able to produce code that is as reusable as possible when the code is first written. Reusable code is seldom developed from scratch. Usefulness is an important part of reusability, and when code is first written, its usefulness may not be completely understood. Thus tools that can analyse an assembly and report on its reusability would be of value, and tools that show how to improve the reusability of an assembly would be even better.

Tools of this sophistication are currently beyond our ability, however we can produce tools that provide more low level information about an assembly. For example, as mentioned earlier, tools that classify assemblies and dependencies would help in this respect. Furthermore, making existing code more reusable requires having a complete understanding of what the code does. Thus tools that show the relationship of the assemblies that make up the existing code and show how it behaves would be useful. Work done with *browsers* and program *visualisation* can help in this respect, especially if they are enhanced to be reuse-specific.

Developing with reuse has a number of tasks associated with it. There is clearly plenty of scope for tool support for searching for assemblies, and in fact most of the work in tool support for reuse is being done in the construction and management of code libraries or code *repositories*. Such tools perform tasks directly related to reusing code, such as developing queries and presenting their results, possibly with some measure of how good a fit they are. There are also tools needed for the management of repositories, such

as classification and storage.

Less work has been done on other tasks associated with developing with reuse. Such tasks include adapting assemblies for new roles and retrieving and using assemblies to construct new assemblies. One important task that is implicit in all aspects of reuse, but does not get the tool support that perhaps it deserves, is the production of documentation. One example is `javadoc` [5, chapter 18], a tool that produces documentation from structured comments embedded in Java source code.

There is clearly a large number of tools, even counting only tools with direct support for reuse. It is helpful to classify the kinds of tools that we are interested in. McClure [7] asserts that there are five types of tools that support the creation and use of a reuse library: repositories, classification tools, browsers, configuration management tools, and cataloguing tools. It should be noted however, that this classification does not cover tools that may be useful for developing for reuse, such as tools for designing, analysing, and testing assemblies.

3 Early Exploration

In this section we outline our early efforts to explore how tools might support reusability in the software construction process. We report on our general findings here. Full details concerning the two prototype tools developed can be found in the theses by Vaks [11] and Miller-Williams [9].

We decided to focus initially on a limited software development situation. Specifically, we choose to focus on the software construction done by students in our second year object-oriented programming course. The practical projects in this course involve designing and implementing C++ programs that will involve a variety of kinds of classes and relationships, typically about 20 main classes, comprising several thousand lines of code. In this course we have felt it to be important that students design and construct substantial programs from scratch, to make sure the students understand these processes well. Accordingly, we stress developing *for* reuse, rather than developing *with* reuse.

The students do their main design work on paper, and before they start to write code they have a reasonable idea of the major classes and the main relationships between them. Their software construction process consists of implementing their design, but this is often not simple or straightforward. As they actually write code, they find that they need to change their design as programming difficulties arise.

Many such issues actually involve reuse and reusability: they realise they could avoid writing duplicate code if they could customise code written once, they begin to see that functionality they thought distinct is in fact structurally similar, and they see inconsistencies that could be avoided if code was written in one place and used in several others. As they address all these issues while coding, they spend much time navigating their program: inspecting, searching, checking, running the program and inspecting intermediate results, all in aid of understanding. They begin to understand and identify which assemblies, classes and methods, are truly useful — worth reusing. And they begin to appreciate which qualities it is necessary for these assemblies to have so that they can be reused easily.

This kind of experience in the middle of program implementation seems typical for beginners. But this is not the whole story. In fact, we believe even experienced programmers still have this kind of experience at times. For beginning and experienced programmers, it is important to recognise these complications as feedback on the design. They suggest opportunities to gain better understanding and improve the design and implementation accordingly. For our first steps in exploring tool support for reusability, we choose to address these issues involving understanding structure while programming. We chose to address the static and the dynamic structure separately.

3.1 Understanding Static Structure

The design of the C++ language supports an emphasis on the static structure of programs. For example, classes, their data members, and member functions are all specified statically and cannot be created dynamically. For these reasons, C++ can offer static type checking, and generation of efficient code. It means that much understanding of the code can be obtained from static analysis. Moreover, it means that code structuring is typically done statically prior to compilation.

Tool support for constructing and understanding the static structure of programs is not new. For example, CASE tools with class browsers use diagrams to illustrate static program structure and facilitate navigation and understanding of program code. We considered some existing tools, such as “Rational Rose” [10], which uses Booch or OMT notation as aid to navigating and understanding C++ programs. We then developed our own prototype tool, to explore how reusability might be better supported.

We identified several ways in which the CASE tools could better support understanding structures involved in reusability. Perhaps most importantly, the diagram notations used did not particularly focus on reuse, and so did not make clear where several contexts used a component, or where several components were used by one context. They did not address the issue of identifying which assemblies were most “useful”, nor highlight qualities that were important for reusability. It is true that the user of these notations does have some freedom in how they are used, and so it might be possible for a user to emphasize reusability within the notation, but this does not really constitute support.

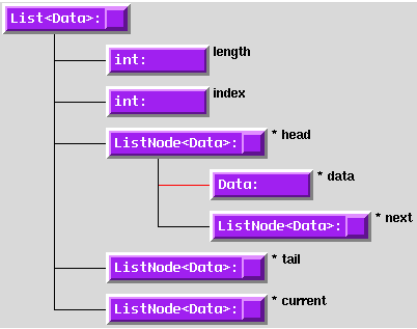


Figure 2: *A class featuring composition, showing the classes used for the data members.*

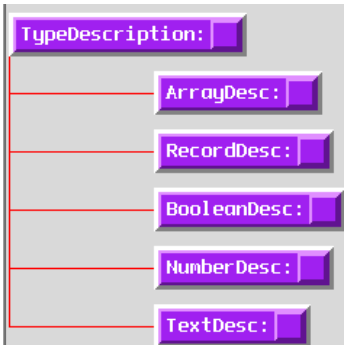


Figure 3: *A class featuring inheritance, showing several derived classes. This illustrates how inheritance is diagrammatically presented as interface conformance.*

In Iivari's study addressing why CASE tools are generally less successful than had been hoped [6], he suggests one key reason is the usability of the tools themselves. Taking this point, together with the one above, we consider that freedom in diagram layout is more likely to be a burden on the user than a valued opportunity to create certain emphasis. We conclude that diagram layout should be automatic, and that the automatic layout should support understanding reusability.

Other factors affecting usability may also affect reusability. For example, users should be able to introduce existing code into a program without having to manually describe the code to a CASE tool. Also, the users environment should ideally still be usable with the CASE tool, so that if possible the user can still use a familiar editor, compiler, debugger, and so on.

The prototype tool we developed addresses these concerns. In particular, we have explored new notation for describing the design of a object-oriented program in such a way as to emphasize reusability. Our diagrams were suggested in part by the kind of "jigsaw diagram" shown in figure 1. The general layout stresses reusability, the symmetry of the context and component roles, and the nature of inheritance as allowing interface conformance for context reuse.

Aspects of the main diagram used in the new tool are shown in figures 2 and 3. In figure 2, a class is shown with the classes of its data members below and to the right. This illustrates the composition or "has-a" relationship by means of a line from the centre of the class to the left of the data member classes. This structure allows component reuse to be emphasised. In figure 3 a class is shown with its derived classes below and to the right. This illustrates the inheritance or "is-a" relationship by means of a line from the lower left of the class to the left of the derived classes. This structure allows context reuse to be emphasised.

Both these structures are combined in a single diagram. The diagram is known as the "composition diagram", because although it shows both composition and inheritance, it especially emphasizes how they work together with inheritance supporting composition by allowing several classes to conform to the same interface. The structure shown on the diagram can be expanded or concealed interactively, so showing the necessary level of detail.

The composition diagram is also the main interface to the tool. In the diagram, each class can be opened, and the member functions and data can be inspected or modified. New classes can be created, and linked to the existing structure, or placed separately. All layout is done automatically. Existing code can be opened, and diagrams for the code are drawn automatically. The diagrams can be used to navigate the code text using the Emacs text editor. The Emacs communications utility Gnuserve is used so that classes and methods are opened automatically, and changes made in text are reflected in the diagrams.

The composition diagram does include both composition and inheritance, and thus does show both component and context reuse. However, it is a hierarchy and only shows structural relationships involved in a left-to-right composition. To specifically show all structural relationships that involve a class, the tool uses a specific "reuse" diagram. The reuse diagram, shown in figure 4, focuses on a given class and shows all the classes that use it (on the left) and that it uses (on the right). The C++ template mechanism that supports parameterised types is also directly supported, and the reuse diagram shows classes that are parameters to a template below the class, and templates to which the focus class is a parameter are shown above the class.

3.2 Understanding Dynamic Structure

The static structure of a program can really only outline possibilities: what *might happen* when a program is executed. The dynamic structure is what actually *does happen* at execution time. Because C++ supports a design emphasis on

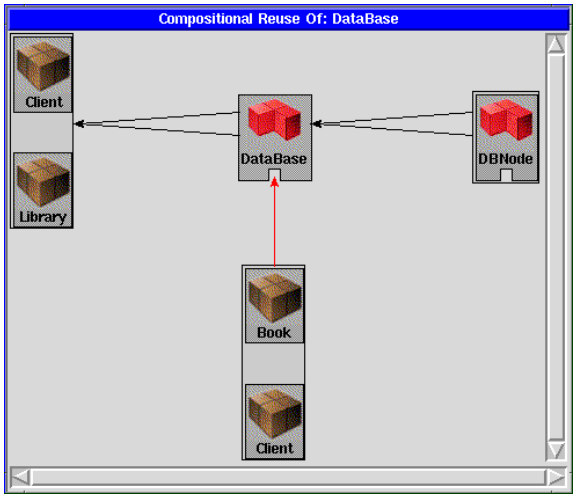


Figure 4: *The reuse diagram, depicting a focus class in the centre, those that use it (on the left), and that it uses (on the right). In this case the focus class is a template, and classes used as parameters to the template are shown below it.*

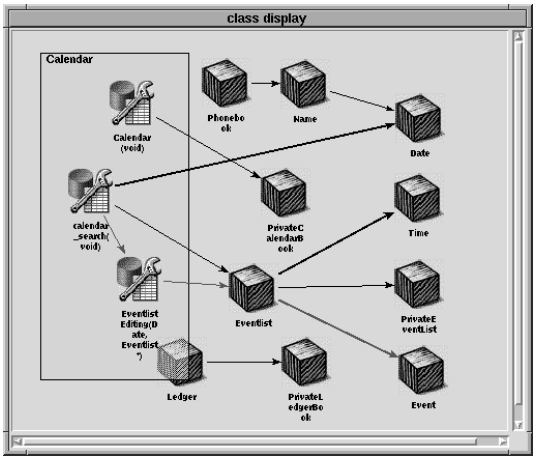


Figure 5: *An expanded object, showing which methods within the object have made calls to other objects.*

static structure, tool support based on the static structure can help a great deal. But, no matter how static the behaviour of a program, the static structure can never tell the whole story. While dynamic structure primarily relates to one particular execution of a program, in fact we can learn a lot about the general dynamic structure by inspecting a number of particular executions.

The simplest programs to help understanding of dynamic structure are debuggers, but these typically focus on low level implementation details. Algorithm visualisation systems focus more on high level issues, but focus on algorithms rather than actual program structure. More research has led to "program visualisation" systems, and many do support understanding program structure, especially those relating to reverse engineering. Nevertheless, none of these systems specifically address issues of reusability.

We developed a second prototype tool specifically to explore program visualisation emphasising reusability. The tool runs the target program, and the program may be paused, resumed, or restarted at any time. As the program runs, the tool displays diagrams showing the structure by showing objects (instances of classes) as they are created. When one object calls a method of another object, a line is drawn between the objects, and an envelope is animated moving in one direction at the call, and back when the call method returns. Any call still current is represented by a red line, and calls already complete are represented by black lines. The thickness of the line is increased as calls from distinct places are detected, to greater emphasize the occurrence of reuse, and highlight potential usefulness.

Classes can be expanded, as illustrated in figure 5, thus showing the methods belonging to the class. When calls are made to or from opened classes, the lines connect to the methods involved. Opened classes are displayed transparently against the background diagram to aid comprehension which diagram reorganisation might otherwise hinder.

These animated diagrams make it clear which objects use and are used by which other objects, and also extend to the methods within objects. This emphasis supports understanding of how both context and component reuse work at run time. Nevertheless, it can be difficult to understand larger scale patterns that occur. To try and support understanding of larger scale dynamic patterns, the tool also provides a "trace view" which shows how objects use each other over time, as shown in figure 6. The trace view is based on the run-time call stack, and the calls from one object's method to another object's method are depicted from left to right on the horizontal axis. When a method returns it remains shown, and later calls from the same method are then displayed below using the vertical axis. In this way, any period of execution is shown as a sideways tree, with the root at the left, and any path through the tree represents the run-time stack at a particular time. Together the horizontal and vertical axes are used to show the run-time trace over a period of time. The trace view is displayed in real time as the program executes.

The trace view allows the user to see repeated patterns in the dynamic structure, as structural repetition is depicted as graphical repetition. To make it easy to see such patterns over longer periods of time, a "global view" is also provided, as shown on the left of figure 6. The smaller size used for methods allows for less detail, but for longer periods to be shown, so enabling detection of longer-term patterns.

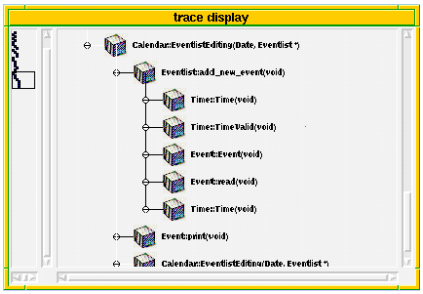


Figure 6: *The trace diagram, showing the history of the runtime stack. The window may be scrolled, and the panel at the left shows the global view in lower resolution, with the currently visible section indicated by the rectangle.*

3.3 Experience

The new tools described above are prototypes, and our main aim in developing them was to develop better understanding of how such tools might support understanding related to reusability. Although we designed the tools with student users in mind, we did so primarily to help us focus on real users, and have not so far made the tools available for student use. We have, however, used actual student projects to test the tools, and consulted with students on design issues.

Our main conclusion is that it is certainly a reasonable idea to support understanding of reusability by using diagrammatical notation. By using the prototype tools, we have gained better understanding of programs that we thought we knew well. It seemed important that the diagrams were generated automatically, because we have always been able to generate such diagrams ourselves, but the effort involved meant that exploration was always very limited. With greater freedom to navigate and explore programs in development, we found it easier to identify key assemblies, and found the depiction of qualities affecting reusability helpful. We especially feel that the principled use of layout is an important factor in the understanding assisted by our diagrams.

While we are optimistic, we feel that more work is necessary. In particular, we have several different diagrams that show similar information that should perhaps be better integrated. For example, the static composition diagram shows some information also shown in the reuse diagrams, but separately. Moreover, if the static diagrams show possibility and the dynamic diagrams show actuality, then the static diagrams might encompass the dynamic diagrams. We also wish to explore other ways of bringing the static and dynamic tools together. For example, the dynamic tool could keep a cumulative history across many executions, creating a middle case between one possibility and the general case.

We view our exploration with our prototype tools very positively. We feel that the prototype tools have shown the value of diagrams in supporting understanding that relates to reusability, but there are several issues, both with the design of the diagrams and the implementation of the tools, that suggest other approaches are worth study. We must remember that actual reusability is our concern and make sure we see beyond understanding to actually achieving better reusability. We must also remember that these prototype tools addressed only some issues of particular aspects of the software construction process.

4 Conclusions

We believe that software development can be made more efficient by developing tools that provide direct support for reusing code. We have been studying the process of reusing code to determine what kinds of tools may be most effective. In this paper, we first briefly described our model for understanding language support for reuse. We then outlined the processes involved in developing software both *for* reuse and *with* reuse, and suggested how tools might support these processes. Finally, we discussed our first steps in practical exploration, involving two prototype tools, a static and a dynamic analyser, both of which were developed with support for reuse as their main goal. In both our theoretical and practical early work, we have found that an explicit emphasis on reuse support seems beneficial and practical.

References

[1] 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.

[2] Robert Biddle and Ewan Tempero. Modeling units of reusability. Technical Report CS-TR-96/19, Victoria University of Wellington, November 1996.

[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] James Gosling, Bill Joy, and Guy Steel. *The Java Language Specification*. The Java Series. Addison-Wesley, 1996.

[6] Juhani Iivari. Why are CASE tools not used? *Communications of the ACM*, 39(10):94–103, October 1996.

[7] Carma McClure. *Software Reuse Techniques*. Prentice Hall, 1997.

[8] Hafedh Mili, Fatma Mili, and Ali Mili. Reusing software: Issues and research directions. *IEEE Transactions on Software Engineering*, 21(6):528–561, June 1995.

[9] John Miller-Williams. A program visualisation tool for emphasising the dynamic nature of reusability. Master's thesis, Victoria University of Wellington, 1997.

[10] Rational Software Corporation. Rational Rose/C++: Version 3.0.3 for Solaris, 1996.

[11] Eduard Vaks. A CASE tool for emphasising reusability in object-oriented program development. Master's thesis, Victoria University of Wellington, 1997.