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
Email: Tech.Reports@mcs.vuw.ac.nz
Http://www.mcs.vuw.ac.nz/comp/Publications

Tools to Aid Learning Reusability

Robert Biddle, John Miller-Williams, Ewan Tempero,
and Eduard Vaks
`http://www.mcs.vuw.ac.nz/comp/Research/renata/`

Technical Report CS-TR-98/3
May 22, 1998

Abstract

Software development can be facilitated by the use of tools within the development environment. In this paper we investigate the use of tools within the development environment to improve learning. Specifically, we are interested in tools that support our approach to teaching design through emphasising reusability. We discuss criteria for evaluating such tools, present prototypes of two tools we have developed, and discuss our experience.

Publishing Information

A version of this paper appeared in the Proceedings of the Third Australasian Conference on Computer Science Education, 1998

1 Introduction

Students learning to program are often more concerned with program implementation than program design. Even when they understand the programming language features they use, they often do not understand when or how to apply them properly. As a result, they often create poorly designed programs without being aware of it, despite often having to expend extra programming effort because of poor design. We are exploring the design of tools to address this problem.

Our general strategy is to help students better understand design by concentrating more on design principles with clear practical benefits. In particular, we stress the goal of software reusability. We take care to explain how reusability is affected by program design, and how such design is well supported by many language features [BTar]. For example, we have been successful in explaining how and when to use inheritance in terms of reusability [BT96a]. This approach is in accordance with other work suggesting advantages to an earlier emphasis on design [BMC96].

While we are pleased with our approach, we still find that students have difficulty in understanding design implications while they are actually developing programs. The task of program implementation is sufficiently complex that students can lose track of their design, especially when their design and their program evolve. We therefore speculate that it would be useful for student programming environments to include tools to help the students better understand their program design during development. We have investigated existing tools to determine their suitability for this purpose, and have developed some prototype tools to better understand how such tools might work.

The next section outlines our model of software reusability that underlies our teaching approach, both in the classroom, and now in tool support. Section 3 is a discussion of issues involved in the design of tools to support students learning design via reusability. Section 4 then presents the prototype tools we have designed and implemented. Section 5 reviews how well these tools met our design goals, and explains what we learned as a result. Finally, in section 6, we present our conclusions.

2 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 via 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

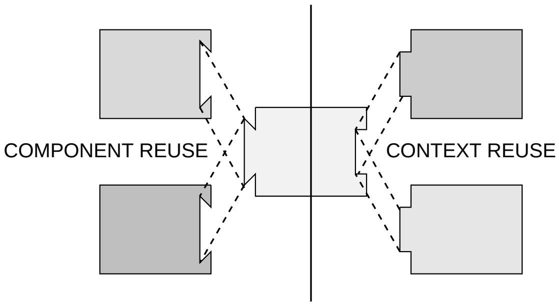

Figure 1: *Component Reuse, as shown on the left of this "jigsaw" diagram, involves several different contexts invoking the same component. Context Reuse, as shown on the right of the diagram, involves one context invoking different components using the same interface.*

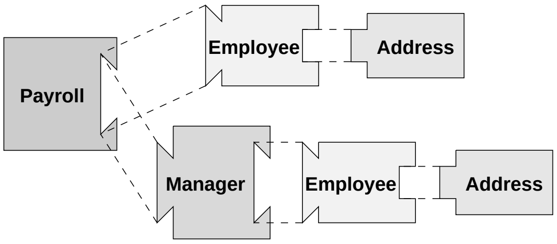

Figure 2: *This jigsaw diagram illustrates how interface conformance makes inheritance different than composition. An "employee" class can reuse an existing "address" class by composition (top), but composition produces a different interface for employee than for address (see tabs on top part of diagram).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

"usefulness". The other is whether or not there are characteristics 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 [BT96c, BT96b]. We use our model in our teaching to facilitate explanation of a variety of principles, involving simpler topics, such as procedures, user defined types, or scope, and also more complex topics such as encapsulation, inheritance, and polymorphism [BTar]. For example, we use our model to explain how inheritance differs from composition, because it uses interface conformance to facilitate context reuse, as shown in figure 2. We discuss this in detail separately [BT96a].

3 Tool Design

To determine how suitable tools are for our purposes, we have been developing criteria for evaluating tools. Our method is to establish principles on which to base the criteria. In this case, the criteria are intended to establish how well tools provide support for reuse in a learning context.

We believe that the two main principles underlying evaluation of tools used by students are:

1. Show concepts clearly.

   At any time, the students must know what concepts are being shown and how those concepts apply to the program under investigation. There should be nothing hiding them, and only a small number of concepts shown at a time. This helps the students check their understanding of the concepts, as well as reinforcing the concepts.

2. Minimise distractions.

   Students should only spend time doing work directly associated with the task at hand. Any time spent on other work is unproductive. In a teaching environment, such time often interferes with learning, since it forces the student to spend time on something that has little to do with the concepts they are supposed to be learning. Since their grasp of these concepts is likely to be weak, they may not learn anything. Furthermore, in situations where the users are not sure what is supposed to be happening, they can spend a lot of time on this other work, convinced that they are making progress.

The concepts we want to support are those associated with reuse and reusability. Our model shows that these are: assemblies, context reusability, component reusability, and dependencies. Most existing tools use some form of, typically graphical, notation to describe the relevant aspects of the software under consideration. This means that evaluating how well the tools support reuse requires evaluating the notation, thus many of our criteria address notations and the diagrams created with them, rather than the tools directly. Applying our first principle leads to the following criteria:

- **Emphasises dependencies.** The notation should show any dependencies that can be detected.

- **Emphasises usefulness.** Since reuse within a program can be a good indicator of likely reuse between programs, the notation should indicate when such reuse is occurring.

- **Differentiates between context and component reuse.** When reuse is being shown, the kind of reuse taking place should be indicated.

- **Emphasises interface conformity.** This is related to the criteria above, but it is possible that the interfaces of two assemblies conform, even though they do not participate in context reuse in a given application.

- **Distinguishes interesting types of assemblies.** Different assemblies can have different properties. For example, an assembly that comes from a library cannot be changed in the same way as one that the programmer has written; but it may be more reliable.

Our second principle leads to criteria that are primarily to do with usability. Iivari reports that one key reason why CASE tools are generally less successful than had been hoped is the usability of the tools themselves [Iiv96]. While the motivation for our work is providing support for students to learn about reuse, usability is equally important: if our students don't use the tools then they might as well not exist. We therefore consider criteria evaluating the usability of tools. The HCI literature contains a number of criteria related to the usability of tools, and we have chosen those that are consistent with our principles. We will not discuss these in detail except to say that our second principle argues strongly for tools that perform layout automatically, a point we discuss below.

The kind of support a tool provides depends on what information the tool has available. For example, a tool that has access to the run-time environment may have different information than one that has access to source code. Our initial work has focussed on "browsers", that is, tools that facilitate the examining of code. We have considered both static browsers (CASE tools) and dynamic browsers (program visualisation systems).

For the static browsers, we considered two existing tools: Rational Rose [Rat96], and Cayenne Software's Teamwork [Inc95]. Rational Rose was the better of the two according to our evaluation, primarily because its usability was better. Teamwork requires the user to do a lot of work that should be done automatically, such as resizing notation elements so that names will fit or placing elements within other elements, and typically done through dialog boxes. Teamwork also sometimes requires the user to add information that it should be able to determine itself, and misleads the user as to what should be done. Rational Rose, on the other hand, provides more help to the user, performing as much work as possible automatically.

While Rational Rose fares quite well in its usability, it does have one problem, namely its speed. It is such a big piece of software that if it is not on a fast enough computer, it performs so slowly as to be unusable. This is an important aspect of usability, and in particular affects a teaching environment in several ways. First, most teaching environments tend to be on under-powered workstations. Second, the delay in feedback means that students lose their concentration, disrupting the learning process.

Neither tools provide much support specifically for reuse. 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 emphasise reusability within the notation, but this would require the user to follow (and remember) conventions, and so does not really constitute support. This is particularly a problem for students, since they would have to follow the conventions before they really understand the concepts.

We also examined a number of program visualisation tools discussed in the literature. Although none of these tools specifically targeted reuse, several did use forms of visualisation that illuminated some of the concepts from our model. The concepts that we chose to explore further were graph visualisation, found in systems by De Pauw et al. [DHKV93], Cerno-II [FHM94], and the Look C++ debugger [And95], and the execution history diagrams of Jerding et al. [JSB96]. The graph visualisations can show both assemblies and certain kinds of dependencies. By including a mechanism to indicate frequency of calls, graph visualisations can also give some indication as to the usefulness of assemblies. The execution history diagrams can be used to detect *message patterns*, that is, common sequences of method calls. Such patterns may be useful for determining useful *subsystems*, that is, groups of classes that combine to perform a logical function. Being able to reuse subsystems represents an effective form of reuse, and so identifying potential subsystems is important.

4 Prototype Tools

In this section we describe the prototype tools we developed. Our goal in developing these tools was to support student program development by providing linkage between the actual program under development and the general principles of design for reusability that are the basis of our teaching. We developed two tools, one addressing the static aspects of program structure, and the other addressing dynamic aspects. In this section we only briefly describe these tools; full details of the tools, and of the criteria outlined earlier, can be found in the theses by Vaks [Vak97] and Miller-Williams [MW97].

For this study, we chose to focus on the programming 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. The development environment is Unix with the X window system, using the GNU G++ compiler, and the Emacs text editor. We did not want to change this environment at this time, and so chose to develop tools as standalone programs that work within the environment with G++ and Emacs. In further work, we would consider creating a more integrated environment, such as that in the Blue system [KR96], where diagrams and source code can be more easily used together.

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. Their design and their program evolve, and it is in this activity the students truly begin to feel the effects of design

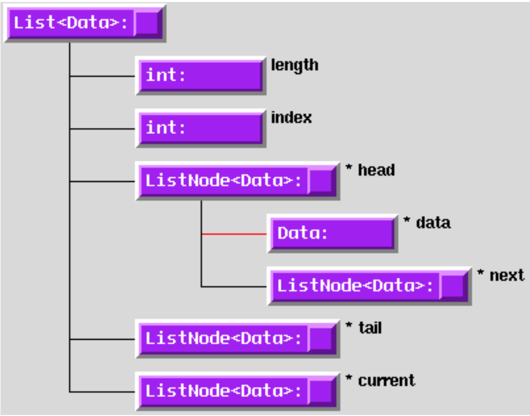

Figure 3: *A class featuring composition, showing the classes used for the data members.*

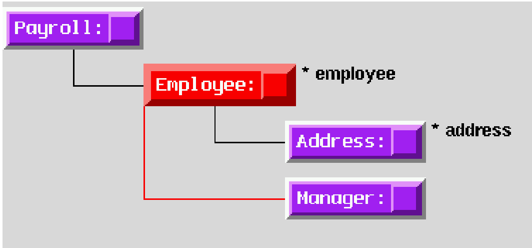

Figure 4: *A class featuring both composition and inheritance, showing how inheritance is diagrammatically presented as interface conformance.*

decisions. This is a critical time because there arises a temptation to concentrate on implementation, rather than on design. By providing tool support that relates their program to the concepts from our teaching, we hope to support the students during this phase of development.

4.1 Static Structure

Our first prototype tool concentrated on static program structure. The design of the C++ language supports an emphasis on the static structure of programs, because classes, their data members, and member functions are all specified statically and cannot be created dynamically. The usual implications of this are that C++ can offer static type checking, and generation of efficient code. For tool builders, it also means that that much understanding of the code can be obtained from static analysis. It also means because programmers must design code structure prior to compilation, so a tool focus on static structure is relevant.

Our approach was to use static program analysis to produce diagrams of the program. In this way, as a student develops a program, they will be able to see the structure reflected diagrammatically. The goal was to use the diagrams to forge links in the student's mind between the structure of their program, and the concepts of program design in our teaching. Our first principle was to show these concepts clearly. To do this, the tool generates diagrams using a notation similar to what we use in teaching, but illustrating the student's program. In this way, the tool provides feedback on the student's program in terms of our design concepts.

The main inspiration for our diagrammatical notation is the pictorial structure shown in the kind of "jigsaw diagram" shown in figures 1 and 2. The general layout stresses reusability by showing a structure consisting of separate assemblies related by invocation. Assemblies are shown playing context and component roles, and both component reuse and context reuse are indicated. In particular, the diagrams highlight the nature of inheritance as allowing interface conformance for context reuse. However, the tool cannot directly use the jigsaw diagrams themselves, because using tab shapes to show interface conformance does not generalise, and because the layout does not scale well.

Two specific sample diagrams generated by the tool are shown in figures 3 and 4. In figure 3, 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 leading to the data member classes. This structure allows component reuse to be emphasised. In figure 4 a class is shown featuring both composition and inheritance in a similar way to that of our teaching diagram in figure 2. The derived class is below and to the right of the base class, and the "is-a" relationship is indicated by means of a line from the *left* of the base class leading to the derived

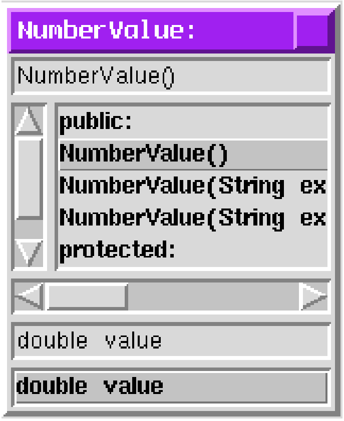

Figure 5: *A class in the composition diagram opened for inspection or modification of data members or member functions.*

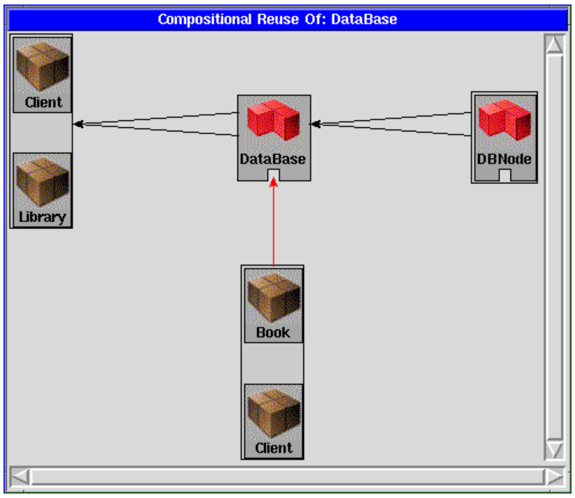

Figure 6: *A reuse diagram, depicting a focus class in the centre, those that use it (on the left), and that it uses (on the right). In this case the focus class is a template, and classes used as parameters to the template are shown below it.*

class. This structure highlights interface conformance, and emphasises context reuse. Although it is not evident in the monochrome diagrams in this paper, colour is also used to distinguish "has-a" and "is-a" lines.

While the figures described above focus on particular kinds of detail in a program design, they are both examples of the same general kind of diagram. In our tool, this kind of 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 composition diagram can be manipulated interactively to expand or reduce the detail shown. For any class shown, the programmer may choose to reveal or conceal any inheriting classes, any classes used in member data, and even the declarations of data members and member functions (see figure 5).

The composition diagram does include both composition and inheritance, and does show both component and context reuse. However, it is a hierarchy and only shows structural relationships involved in one direction. To specifically show *all* structural relationships that involve a class, the tool produces a specific "reuse" diagram. The reuse diagram, shown in figure 6, 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.

The programmer may use both reuse diagram and composition diagram to show various aspects of the program structure. The reuse diagram can be used to focus on any particular class, and it will provide a separate diagram for each class, with that class as a centerpiece, showing immediate relationships from that perspective. The composition diagram may be used to detail structure beginning at any class, and showing "has-a" and "is-a" structure from that point. The structures shown on the diagram can be expanded or concealed interactively, so showing the necessary level of detail. The whole program can be displayed as an integrated presentation of a set of classes.

The diagrams outlined above formed our initial attempts to reflect to the student programmer the static structure of their programs in terms of our teaching principles. They show assemblies and invocation, context and component, interface conformance for inheritance, and so on. However, we wanted our tool to do more than merely *allow* student programmers to be able to see such diagrams. We wanted the students to perceive some practical benefit to using the diagrams, and we wanted them to be easy to use. Our second principle was to minimise distractions. To attempt this, we tried to make use of the diagrams as simple as possible: we eliminated any unnecessary data entry or option selection, made diagram layout and formatting automatic, and allowed control of detail using progressive disclosure in the diagrams themselves.

We also attempted to involve the diagrams in the program development process by allowing them to be used to do more than show program structure: they can also be used to navigate the program source code. In our Unix/X environment, the diagrams can be used as an interface to the Emacs text editor; this is implemented via the Emacs communications utility "Gnuserve". The programmer can then use the diagrams to drive Emacs by indicating which class or method Emacs should present in a buffer, whereupon that class or method will be opened automatically. The composition diagram can also be used to actually modify the program. New classes can be created, and existing classes can be modified by modifying their data member or member function declarations. So the connection between program source and the diagrams is bi-directional: the diagrams are produced by background analysis of the source code, but these same diagrams can then be used to work with, and change, the program itself. However, the diagrams do not deal at all with C++ code inside class member functions.

4.2 Dynamic Structure

C++ supports a design emphasis on static structure, so 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. The static structure of a program can really only outline possibilities: what *might happen* when a program is executed. The dynamic

Figure 7: *An object with a method call to another object completed, and return underway.*

structure is what actually *does happen* at execution time. We developed a second prototype tool to focus on dynamic structure, but still with the same goal, to forge a link between the structure of student programs and the design principles from our teaching.

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. As before, the diagrams are inspired by our jigsaw diagrams, and show invocation from assemblies in context roles to assemblies in component roles. In this tool, however, the assemblies are not classes, but objects (instances of classes) in order to show the dynamic structure of the program, where different objects are involved in different structures.

The diagrams show objects 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 (see figure 7). 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.

Objects can be expanded, as illustrated in figure 8, thus showing the methods of the object. When calls are made to or from expanded objects, the lines connect to the methods involved. Expanded objects are displayed transparently against the background diagram to aid comprehension that 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 follows our first principle

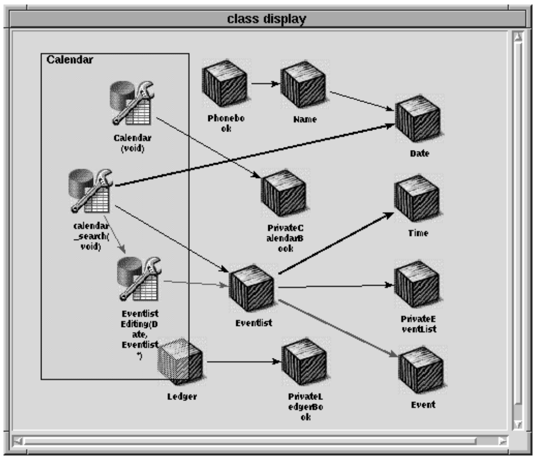

Figure 8: *An expanded object, showing which methods within the object have made calls to other objects.*

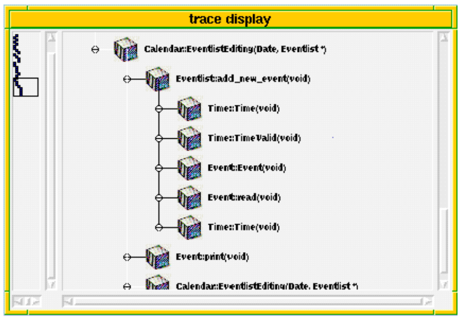

Figure 9: *The trace diagram, showing the history of the run-time stack. The window may be scrolled, and the panel at the left shows the global view in lower resolution, with the currently visible section indicated by the rectangle.*

(show concepts clearly), and supports understanding of how both context and component reuse work at run time.

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 9. 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 message 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 9. The smaller size used for methods allows for less detail, but for longer periods to be visible, so enabling detection of longer-term patterns. The global view can be shown as a diagram itself, and can be adjusted to show methods in individual columns, further facilitating pattern recognition (see figure 10).

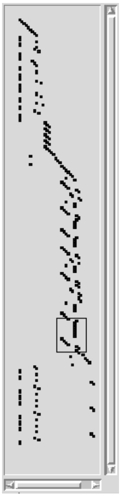

Figure 10: *The global view displayed as a diagram by itself, showing how message patterns can be detected.*

5 Experience

The tools we developed were intended to support student programming and student learning. We designed the tools with two main principles in mind: to emphasise reusability in the way our teaching does, and to achieve usability critical for any tool success. Moreover, we had identified criteria for evaluating such tools with respect to these principles. However, these were only prototypes, and we have not used tested the tools with students in ordinary lab and project situations. However, we did test the tools with student programs, and consulted with students during development and refinement. We also realised that the tools can be very useful for presenting concepts in the classroom, by allowing "live" demonstrations.

We now review our prototype tools with respect to our principles and evaluation criteria, in light of our experience.

5.1 Reusability

The primary concern of our tools was supporting reusability, and we had identified several specific criteria.

- Distinguishes interesting types of assemblies.

  Both tools identify assemblies, the primary unit in our model. Both tools do distinguish kinds of assemblies inherent in the C++ language, such as classes, templates, methods, etc. However, we did not strongly address the need to distinguish user from library assemblies, old from new, or tested from untested.

- Differentiates between context and component reuse.
  Emphasises interface conformance.

  In both tools our diagrams are based in several ways on our ideas involving the component and context reuse, and the ability of inheritance to allow interface conformance and support context reuse. We follow the lead of our jigsaw diagrams, and show this using principles of two dimensional design, for example using left-right symmetry between component and context reuse, and by using confluence of lines to signify interface conformance. While these structures have been successful, we believe closer attention to some details of graphic design might make implications clearer. Also, 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. Finally, although we believe our diagram structure can highlight aspects of reusability better than standard design notations, it might help to conform to some standard notation where this does not conflict with our efforts.

- Emphasises usefulness.

  Usefulness prediction involves showing assemblies that are invoked from, or invoke, different places. The static tool does this particularly in the reuse diagram, where one class is shown with all users of it, as well as all classes it uses. The dynamic tool shows usefulness by increasing line width when invocation of an assembly has come from different places. Both the tools might possibly do more: the static tool could provide hints of usefulness in the composition diagram, and the dynamic tool could provide longer history, even spanning program executions.

- Emphasises dependencies.

  Both our prototype tools emphasise invocation as a primary way in which assemblies are related. The static tool especially shows explicit dependencies such as function and template parameters. However, neither tool properly addresses the need to show implicit dependencies, such as use of global values. This is a weakness because such dependencies can seriously limit reusability. Programmers, especially student programmers, could well use help in identifying such dependencies as early and often as possible. However, determining such dependencies involves more extensive program analysis than we used in our prototypes — we comment further in our brief discussion of implementation concerns, below.

In summary, we feel our prototypes have indicated that such tools can support student understanding of program design and reusability. We feel that the diagrammatic notation is useful, and can provide support for understanding many principles in the design of reusable software. There are several issues that we must explore to improve this support, and we plan to address these in later work.

5.2 Usability

While reusability was our primary concern, we were aware that usability was critical to successful support of *re*usability. There are already aspects of usability in the design of our diagrammatical notation, discussed above. However, we attempted to achieve good usability in other ways in support of the main diagrams.

Our main attempts to achieve usability related to simplifying the process of using the tools. Our experience using other CASE tools in our courses has been that student work has sometimes been distracted by the tools themselves. Sometimes the students have found the distraction bothersome, such as when data is requested from the user when it should have been inferred. Other times the students found the distraction interesting, and so, for example, spent time arranging diagrams in various ways. Our second principle was to avoid distraction, and we attempted to avoid both kinds of distraction above: we kept data input to a minimum, and used automatic diagram layout. We have been pleased with the results, and find this strategy does successfully keep the focus on the program design, rather than the tools themselves.

We also pursued several other strategies to improve usability. In the static tool, the technique of progressive disclosure was useful. The class diagrams do not initially show full details, but where full details are available, the class symbol shows a button. When clicked, the class is expanded showing data members and member function declarations. In the dynamic tool, where diagram layout was more complex, we used found transparency useful. Again, the situation was expanding classes to show member details. Because the diagram layout was more complex than in the static case, we found it distracting and disorienting to change the layout in order to expand the class. Instead, we displayed the expanded class as a transparent layer superimposed on the otherwise unchanged diagram, and so provided the extra detail without distraction.

There were various weaknesses in usability as well. In particular, the total functionality of the tools is not made adequately easy to use, and relies on special key sequences and so on. However, the tools were only prototypes, and never intended as complete finished software products.

We feel that our attempts to achieve usability showed that our approach was right, and that usability did indeed significantly improve our support of understanding reusability. The most significant insight we can offer about usability in our prototypes concerns simplification of the process of using the tools. Simplification does indeed avoid distraction. However, we saw signs of a deeper effect as well. As using the tools became simpler, the barriers between program and tool began to diminish: the diagrams seemed to *become* the programs, and the structure of the diagrams began to influence our thinking about the programs. The experience was a sensation similar to that in interaction Schneiderman calls "direct manipulation", and he argues there is a strong relationship between direct manipulation and learning (see his discussion in section 6.3 of [Shn98]). We find our experience an encouragement for further work.

5.3 Implementation Issues

Although the technology used to implement the tools is not a primary concern of this paper, it is worth commenting on some interesting lessons. Both tools used TCL/Tk [Ous94] for the graphical interface, and both tools fitted in with GNU software used in our development environment.

The main lesson learned from the implementation of the static tool concerned difficulties involved in parsing. While we saw it as important to be able to analyse C++ source code, we underestimated how much effort would be needed to parse the code sufficiently. C++ is syntactically complex, and many detail issues became bothersome, even in dealing with library code or code from student projects. It is more clear to us now that this kind of tool requires either a large and complicated parser, a simpler subject language than C++, or a limited need to deal with existing code. All this relates strongly to the ability to faithfully indicate implicit dependencies between assemblies.

The main lesson learned from the implementation of the dynamic tool concerned the value of scripting and the GNU debugger GDB. GDB is a source level debugger for C and C++, with a rich text command interface, and good capabilities for breakpointing and data access. Although the dynamic tool works as a complete tool with its own user interface, internally it uses GDB extensively. To be more precise, the tool uses the Expect [Lib95] scripting language to drive GDB, passing it commands and interpreting the results. GDB is used in this way to run the target program, pause at points relevant to diagram drawing, and provide details necessary to the diagrams. This method proved easier, more flexible, and more powerful than expected. Curiously, it made some static structure more easily accessible than it was in our static tool, because it harnessed static analysis already done by the compiler in generating debugging information. As a result, we now view dynamic analysis as less forbidding, and are exploring new ideas that might use this approach.

6 Conclusion

In this paper, we have discussed tool support to aid students learning about reusability. Our motiviation for this is to assist our efforts in using reusability to teach program design. Our methodology has been to develop criteria for evaluating tools that provide such support, based on principles for learning. We have used those criteria to guide the design of, and evaluate the success of, two prototype tools.

Our conclusion is that tool support for learning about reusability can be effective. 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. In particular, it appears to give the benefits to learning associated with direct manipulation. We believe this is an important realisation, and we plan to explore this further.

We observe that while we have been concerned with evaluating tools with respect to their suitability for supporting the learning process of our students, all of our criteria also apply to tools used in industry. This has led us to speculate that the relationship between the experience of students while learning, and the experience of professionals in industry, is closer than is usually assumed.

References

[And95] Eddie Anderson. Dynamic visualisation of object programs written in C++. Objective Software Technology Ltd, Software Innovation Centre, 1 Michealson Square, Kirkton Campus, Livingston, Scotland, EH54 6QA, 1995.

[BMC96] Robert Biddle, Rick Mercer, and Alistair Cockburn. Report on workshop on teaching and learning object design. In *Addendum to*

Proceedings of 1996 ACM OOPSLA Conference, San Jose, California, USA, 1996. ACM SIGPLAN.

[BT96a] Robert Biddle and Ewan Tempero. Explaining inheritance: A code reusability perspective. In Karl J. Klee, editor, *Proceedings of the Twenty-Seventh ACM SIGCSE Technical Symposium*. ACM Press, February 1996. Also available as Technical Report CS-TR-95/18.

[BT96b] Robert Biddle and Ewan Tempero. Modeling units of reusability. Technical Report CS-TR-96/19, Victoria University of Wellington, November 1996.

[BT96c] 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.

[BTar] Robert Biddle and Ewan Tempero. Teaching programming by teaching principles of reusability. *Journal of Information and Software Technology*, To appear.

[DHKV93] Wim DePauw, Richard Helm, Doug Kimelman, and John Vlissides. An architecture for visualising the behaviour of object-oriented systems. In *1993 Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, pages 326–337, 1993.

[FHM94] S Fenwick, J Hosking, and W Mugridge. Cerno-II: A program visualisation system. Technical Report Auckland Computer Science Report No. 87, University of Auckland, February 1994.

[Iiv96] Juhani Iivari. Why are CASE tools not used? *Communications of the ACM*, 39(10):94–103, October 1996.

[Inc95] Cayenne Software Inc. *Teamwork, Version 6.1.2*. 8 New England Executive Park, Burlington, MA 01803 USA, 1995.

[JSB96] Dean F. Jerding, John T. Stasko, and Thomas Ball. Visualising message patterns in object-oriented program executions. Technical Report GIT-GVU-96-15, Graphics, Visualisation, and Usability Center. Georgia Institute of Technology, May 1996.

[KR96] Michael Kolling and John Rosenberg. An object-oriented program development environment for the first programming course. In Karl J. Klee, editor, *Proceedings of the Twenty-Seventh ACM SIGCSE Technical Symposium*, pages 83–87. ACM Press, February 1996.

[Lib95] Don Libes. *Exploring Expect: A Tcl-based toolkit for automating interactive programs*. O'Reilly and Associates, 1995.

[MW97] John Miller-Williams. A program visualisation tool for emphasising the dynamic nature of reusability. Master's thesis, Victoria University of Wellington, 1997.

[Ous94] John Ousterhout. *TCL and the TK Toolkit*. Addison-Wesley, 1994.

[Rat96] Rational Software Corporation. Rational Rose/C++: Version 3.0.3 for Solaris, 1996.

[Shn98] Ben Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Addison Wesley Longman, third edition, 1998.

[Vak97] Eduard Vaks. A CASE tool for emphasising reusability in object-oriented program development. Master's thesis, Victoria University of Wellington, 1997.