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

Reuse of Debuggers for Visualization of Reuse

Robert Biddle, Stuart Marshall, John Miller-Williams,
Ewan Tempero

Technical Report CS-TR-98/10
November 1998

Abstract

We have been investigating tool support for managing reuse of source code. One approach we have been exploring is the use of program visualization in programming tools. A difficulty with this approach is that effective program visualization of ordinary programs requires substantial sophisticated low-level software. Our solution to this problem is to reuse debugging systems in an innovative way. We present our experience with this approach, both to evaluate the reusability of the debugging systems we used, and to provide a case study in reuse.

Reuse of Debuggers for Visualization of Reuse

Robert Biddle, Stuart Marshall, John Miller-Williams, Ewan Tempero
School of Mathematical and Computing Sciences
Victoria University of Wellington
Wellington, New Zealand
http://www.mcs.vuw.ac.nz/comp/Research/renata/

Abstract

We have been investigating tool support for managing reuse of source code. One approach we have been exploring is the use of program visualization in programming tools. A difficulty with this approach is that effective program visualization of ordinary programs requires substantial sophisticated low-level software. Our solution to this problem is to reuse debugging systems in an innovative way. We present our experience with this approach, both to evaluate the reusability of the debugging systems we used, and to provide a case study in reuse.

1 Introduction

Two important strategies for improving programmer productivity are *reuse* of code to reduce the programmer's work, and the development of *tools* to aid the programmer. We acknowledge that other non-technical factors will also be necessary for success, but we believe that making reuse technically easier is still important. We have recently been exploring new ways to combine these approaches by providing programming tool support to provide assistance at various points in the reuse process.

One approach we have been exploring in tool design is the use of program visualization to help the programmer understand various dynamic aspects involved in code reuse. A problem with this approach is that effective visualization of real programs is a non-trivial problem in its own right. In particular, program visualization systems are typically large, complex, and involve substantial low-level programming. We were reluctant to develop such software ourselves, especially because our primary interest is in reuse. Our solution has been to reuse debugging systems in an innovative way, so they become a key component in our visualization tools.

In this paper, we present our experience with this use of debuggers, both to evaluate the reusability of the debugging systems we used, and to provide a case study in reuse. We have developed two different tools, each using a different debugging technology: one for C++ programs that uses Gdb and Expect, and another for Java programs that uses the Java Virtual Machine Debugger Interface (JVMDI).

This paper is organized as follows. The next section briefly outlines our two tools, explaining how program visualization is used in the tools to support reuse, and reviewing what the tools require to allow visualization. Section 3 presents our general design approach for program visualization systems, and our two specific systems are described in sections 4 and 5. In section 6 we discuss the effectiveness of our systems, and evaluate how suitable the debugging systems were for reuse. Finally, section 7 presents our conclusions.

2 Tools to Support Reuse by Program Visualization

The first of our two dynamic visualization tools, *Dy-re* (for *dy*namic *re*use), is an attempt to support programming *for* reuse. Dy-re is intended to assist programmers better understand the structure of the software they themselves are developing, and better identify potentially reusable components within the structure. As the target program runs, Dy-re displays diagrams that show the code components and their connections. The program retains its usual user interface, but Dy-re also provides a simple interface to allow the program to be run, stopped, or paused.

Dy-re uses diagrams to show both the current situation, and also cumulative information. These visualizations make it easy for the programmer to detect patterns of usage and patterns of dependence within a program. These patterns in turn help the programmer determine how best to articulate the structure of a program using components that will be useful and independent for later reuse in other contexts. Section 4 contains more detailed discussion of the of Dy-re, including figures showing sample visualizations, and details of the implementation.

The second tool, *Dyno* (for *dyno*mometer), is an attempt to support programming *with* reuse. Dyno is intended to assist programmers better understand the correct usage and functionality of a component they are considering reusing.

There are several steps that have been identified in the process of reusing a component [MMM95]. Dyno assists at the point in the process where a programmer has identified and retrieved a candidate component. At this point, the programmer needs to verify that the component will actually meet their requirements, and then needs to determine how to actually use the component. Both these steps will require some level of understanding of what the component actually does. For example, the programmer may have found a queue component, but may need to determine whether it provides a necessary mechanism for examining the front element of the queue without removing it.

Dyno provides support for this point in the reuse process by providing a mechanism to allow the programmer to take the component on a "test drive". To do this, Dyno must provide the programmer with the means to control the component, and also the means to understand the component behavior as it is controlled. Dyno uses program visualization techniques to do the latter, allowing both standard visualizations, such as used in Dy-re, and also visualizations tailored to the specific component. Section 5 contains more detailed discussion of Dyno including figures showing sample visualizations, and details of the implementation.

Dy-re and Dyno address different aspects of reuse. However, they both support reuse by providing visualization, and to do this they have a number of key requirements in common. Both tools work with running programs, both tools need to control the programs, and both programs need to be able to determine what the programs are doing. The key requirements are controllability and observability of running programs. Both tools also need to be target independent, and work with ordinary programs and components, not just specially created or annotated code. Moreover, the tools should also be unobtrusive, and not change the code itself to assist visualization.

3 Design for Program Visualization

In this section, we discuss the approach we take to design our tools to support visualization. We use the PMV model, created by Stasko and Roman and Cox, which describes an general architecture for program visualization [RC93]. We are especially influenced by the extended version developed by Noble [NGB95]. Figure 1 illustrates this model. The PMV model is named after its three components: the program component, the mapping component, and the visualization component.

3.1 Program Component

The program component interacts with the target program. Its purpose is to determine what the target program is doing, and send information, in the form of *actions*, about what has occurred on to the mapping component. This information can vary. For example, an action might be a function call or an object instantiation. Actions can also contain extra information about the target program, such as information about the performance of the program.

The method of obtaining the actions from the target program also varies widely. Action generation can come from three possible phases in program creation: at the source code phase, by modifying the source code (manually or automatically) to generate the actions, the compiler phase, by modifying the compiler to insert the actions, and at the executable phase, using post-processing to generate actions from information contained in the executable. Finally, actions can be generated directly from the executable. Post-processing the executable uses information contained in the compiled binary to generate events. This method is highly transparent; neither the source code nor the compiler needs to be modified, however only information that is contained in the executable can be used.

3.2 Mapping Component

The mapping component translates the actions sent by the program component into a format, called *changes*, that can be used to update the visualizations. For example, an action might be a function being called, while the change would be a request to display this event in the visualization. This

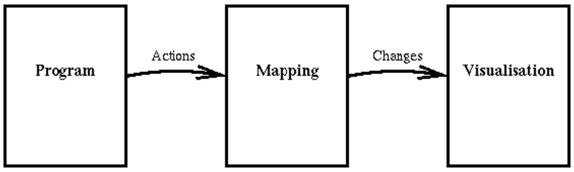

Figure 1: *The PMV model.*

means the visualizations no longer have a direct dependency on the low-level output of the program component. Additionally, the mapping component might perform other tasks, such as filtering out unwanted information.

3.3 Visualization Component

The visualization component receives the changes from the mapping component and uses these to produce and update the output images. The operations of the visualization component can also vary widely. Multiple visualizations can be displayed simultaneously. The output can be produced once, or updated during the execution of the program to create an animated display. Visualizations can be interactive or non-interactive. Interactive visualizations allow the user to modify the graphical features of the display. For example, the user could set display colors, arrange the layout, or focus the visualization on specific areas. At the most advanced level the user might fully define the visualizations using a specification language.

3.4 The PMV Model and Debugger Reuse

The most difficult part of the implementation of a program visualization system is extracting the necessary actions for the program part of the PMV model. We want to provide actions as low-level as the program code, because we cannot predict what aspects of the code behavior the programmer may be interested in. We also need to do extraction in a target independent and unobtrusive way. This means we cannot generate actions at the source code phase, and for portability and ease of use reasons we do not want to generate actions from the compile phase. This leaves the executable phase as the best place to generate actions.

We wanted to create program components with actions generated at the executable phase, but we did not want to undertake substantial low-level software development. The important realization was that a debugger provides much of the functionality required, and any mature programming environment has some form of debugging system. Accordingly, we speculated that a debugging system could provide the critical functionality of the PMV program component with actions generated at the executable phase. In fact, this use of a debugger is really more likely to be a combination of action generation at the compile and executable phases, because most debuggers require a compiler to embed information within the generated code to assist debugging.

The specific idea is that such reuse of a debugger is especially suitable to program visualization. The more general idea is that debugging systems may have a wide range of applications, beyond debugging, and even beyond visualization. Our immediate goal, however, was to explore the practical application of the idea as a proof-of-concept. In particular, we set out to determine whether debuggers do in fact have all of the necessary functionality available in a useful way for the PMV program component in our tools.

4 Dy-re

In this section, we describe the design and implementation of Dy-re. This tool was intended to support programming *for* reuse, by dynamic display of the internal structure of software in development. The general ideas would apply to many programming languages, but Dy-re itself was built specifically to work with software being developed in C++.

The architecture of Dy-re is illustrated in figure 2. The system is composed of three logical subsystems that correspond to the three components of the PMV model. A pure representation of the PMV model might separate the components completely and develop a communication protocol between them. However, in our case, it proved easier for the mapping component and the visualization component to

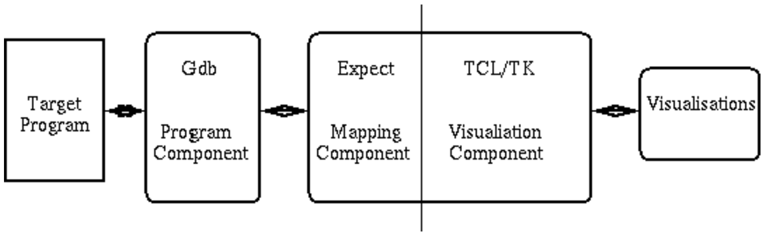

Figure 2: *The structural divisions of the Dy-re.*

be parts of the same physical program. We now detail the design and implementation decisions of the three components.

4.1 The Program Component

The program component interacts with the target program. It extracts run-time events from the target program, and passes these actions on to the mapping component. Dy-re's program component is Gdb, the Gnu source level debugger. Gdb is a debugger for C and C++, with a rich text command interface, and good capabilities for breakpointing and data access.

Using Gdb allows easy access to the target program, and easy access to the debugging information that has been inserted into the executable code by the compiler. Gdb also provides the ability to determine information about the program, run the program, have it stop on various conditions, and allow access to the current state of the program, such as the call stack. When the user selects the target program, Gdb reads in the debugging information contained within the executable. Gdb extracts the names of all the methods in the target program (excluding the basic system library routines). It then places breakpoints at the start and end of these methods so that the program will pause at these places. When the user runs the program, if a message call or return occurs, the program will pause. At this point it is possible to extract information about the current state of the program (for example, the call stack). The program is then restarted, so that from the user perspective, the visualizations appear to be updated continuously.

4.2 The Mapping Component

The mapping component translates the low-level actions sent by the program component into changes that can be used to update the visualizations.

When designing the mapping component, the following implementation issue needed to be considered: how should the mapping component interact with Gdb? We considered two possibilities: the first was to "hard-wire" the mapping component to the internal parts of Gdb to directly access data and control the program. The other was to control Gdb externally through the text-based interface, sending in commands and parsing the output.

Modifying Gdb's internals would remove a layer of communication and mean faster access to program information. Communicating externally with Gdb, and parsing the output of the program, would improve portability. Changing Gdb's source code would mean that the modified version would need to be kept with the visualization system. Gdb is a reasonably large program, and most Unix systems already have a version installed. Keeping Gdb as a separate entity results in a small compact tool that can be easily transported.

The decision to communicate externally with Gdb was mainly due to the availability of Expect [Lib95]. Expect is designed to automate interactive text based programs. It communicates with these programs using a script written by the user. Following the script, Expect knows what output is expected from a program and what the correct response should be. As the user manual says, "All that is necessary is that the interaction can be characterized programmatically". The output of Gdb is fairly formalized and therefore not particularly difficult to parse using Expect. Moreover, we discovered that communicating externally did not significantly reduce efficiency. The primary delays experienced when using the program appears to caused by Gdb itself.

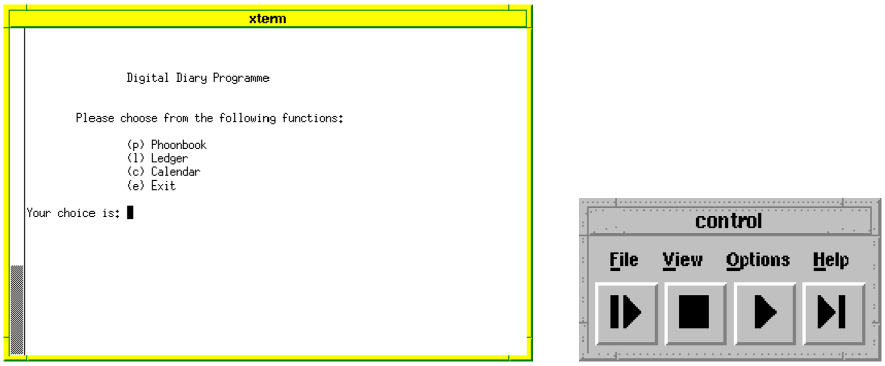

Figure 3: *Interacting with the target program: its usual interface is unaffected by the visualization system, and is shown on the left. The main interface for Dy-re itself is shown on the right.*

4.3 The Visualization Component

The visualization component takes incoming actions from the mapping component and uses them to update the various views. This component is composed of two parts: the visualizations themselves and the supporting structure for the visualizations. The supporting structure provides routines to activate and destroy the visualizations, and acts as a controller ensuring each visualization gets only relevant updates.

In Dy-re we decided that the visualization component should be implemented using Tcl, and the Tk graphics library [Ousterhout, 1994]. Tcl/Tk already provides a number of quite high-level graphics functions for manipulating the provided library of graphical "widgets". Furthermore, Expect works by using Tcl, and use of Tcl/Tk for the visualizations means that connection between the mapping and visualization components is easy. In fact, the version of Expect we use is "Expectk" [Lib94], a combination of Expect and the Tk toolkit. As a result, it was possible for the mapping component and the visualization component to be two parts of the same program. This then became our design, and communication between the mapping and visualization components was made even easier because the mapping component could create data structures that were usable by the visualization component.

4.4 Dy-re in Action

Illustrating the dynamic aspects of reuse internal to a program is the main job of Dy-re. Dy-re runs the target program, and the program may be paused, resumed, or restarted at any time. The main control is based on a tape deck metaphor (see figure 3). This gives the user an intuitive way to manipulate the program events. As the program runs, Dy-re displays diagrams. The diagrams are inspired primarily by our more general research into programming language support for reusability [BT96], where we discuss in detail various program units we call *assemblies*, and the *dependencies* between them.

In the Dy-re diagrams the assemblies are typically objects (instances of classes) and the dynamic structure of the program is shown where different objects are involved in different dynamic 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 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 of particular program units. Figure 4 shows the situation part-way into the execution.

Objects can be expanded, 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 reorganization 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 allows us to 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" that shows how objects use each other over time, as shown in figure 5. 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

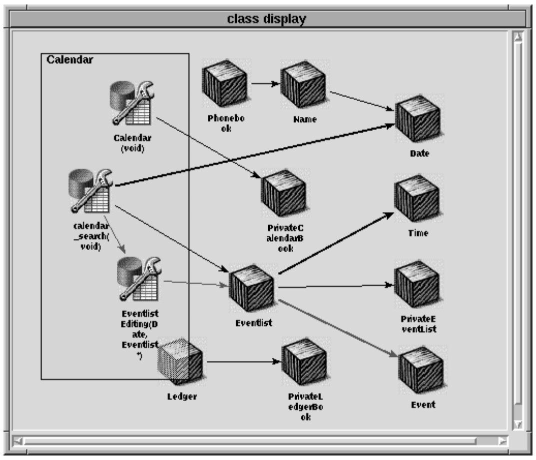

Figure 4: *Midway through execution, several other objects have been created and are in use.*

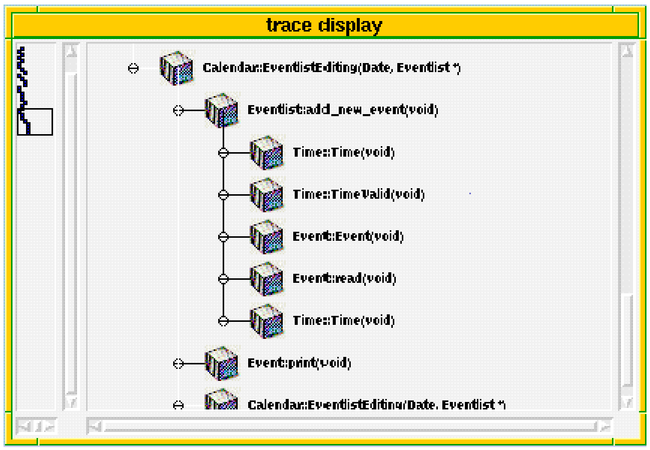

Figure 5: *The trace diagram, showing the history of the run-time stack.*

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.

5 Dyno

Dyno is intended to explore the idea of animation as documentation, specifically to assist the user to understand the behavior of a Java component. Its architecture is also based on the PMV model.

5.1 Program Component

The Program Component in Dyno is implemented using the Java Virtual Machine Debugger Interface (JVMDI), which is part of JDK 1.2. It uses the Java Native Interface (JNI) to provide methods for performing debugging on Java code. It does so in the form of a collection of C/C++ methods that enable the user to place breakpoints in code, determine when methods exit, inspect the execution stack, determine the values of local variables and parameters to methods and retrieve information about methods and classes. JVMDI works by sending signals for events to a method (written in C/C++ using the JNI) specified earlier by the user via another JVMDI method. These signals differentiate between events for:

- breakpoints being reached.
- frames being popped off the stack.
- classes being linked into the Java Virtual Machine (JVM).
- classes being unlinked from the JVM.
- starting a Java thread.
- stopping a Java thread.
- an exception being thrown.
- a single byte code execution.

Dyno uses JVMDI to place breakpoints at the beginning of all methods (both instance and static) other than those belonging to classes in the JDK distribution or those that comprise Dyno itself. The user can then request that methods be executed on an object or class in the system and Dyno's JVMDI code will detect when methods are first entered, forwarding appropriate information to the Mapping component for further processing so that visualization data can be produced.

As well as this, Dyno then sets up the JVMDI code so that when the frame owned by the method whose start-up has just been detected is popped off the execution stack, Dyno will again be informed (at which point it again sends information to the Mapping component). This guarantees that Dyno is capable of detecting method invocations and returns on methods of interest to the user, and that sufficient information is capable of being passed to the Mapping component so that visualizations can be created.

5.2 Mapping Component

A visualization in Dyno can consist of multiple views of the program. The Mapping component translates the events generated by the Program component into a form that the different views use. The Mapping component consists of an event filter and a data model. The event filter determines which events from the program component are needed for the visualization. The data model maps objects in the execution to elements in the visualization.

5.2.1 Event Filter

Not all events will be important to a visualization. The user may want to create a visualization of a stack being used in an algorithm, therefore operations on the stack will have to be forwarded onto the visualization code, however operations elsewhere in the algorithm that don't affect the stack should be ignored.

Dyno uses event filters to determine whether or not an event is relevant to the visualization being created. Event filters keep track of which operations are needed ("push" and "pop" for example) and also stores information given by the user as to which methods in the user's code map to the view's operations. When the Mapping component receives an event of interest from the Program component it takes a snapshot of the part of the program execution state corresponding to data model. The snapshot is then passed to the Visualization component. More than one event filter may exist at any one time inside Dyno, listening for different events for the different views.

5.2.2 Data Model

The data model contains the mapping of a view's attributes to data in the user's code. A stack view might have two attributes, an "elements" attribute and "top" attribute. A user's implementation of a stack might consist of a class wrapped around a java.util.Vector, in which case the data model will map "elements" to the java.util.Vector field and will also map "size" to the return value of the size() method in the java.util.Vector class (determined dynamically whenever the snapshot is taken). When this data model is used by the Visualization component, the stack view can then retrieve values for its "elements" and "size" and use them to display a visualization of a stack. This mapping of a view's elements to data retrieved from executing code may use the Reflection API to interrogate an object for the value of one of its fields, or a return value from one of its methods. The Reflection API is a JDK 1.1 feature that allows for introspection of objects and classes.

5.3 Visualization Component

The Visualization component allows for custom made views to describe the execution of components. The custom view must supply some simple information on how the abstract operations and attributes that the view represents relate to methods or state in the target program for the purposes of the Mapping component. The Visualization component may consist of multiple views at any one time.

A view in the Visualization component reads in the information it needs for the attributes from a file written out by the Mapping component (where it also finds the sequence of operations performed on it, mapped from the methods the event filter passed on). This sequence of events is then displayed one at a time creating an animation as the state of the component being examined changes.

5.4 Dyno In Action

Dyno allows users to load Java classes into the application from the current classpath. Dyno displays the list of loaded classes ordered by the inheritance hierarchy used by Java. Dyno also lists arrays and any interfaces for which there are instances present. The purpose of this ordering is so that when a user needs an instance of a certain type (such as requiring a java.lang.String for use as a parameter to a method), the user can easily see which instances satisfy the type requirements.

The interface allows the user to select a class, object, or primitive, and will then proceed to display information concerning the methods and fields on that object. If the object is a class then the information displayed relates to the static methods and the static fields of that class, otherwise the instance methods and instance fields are shown, or the value of the primitive variable. For objects, the methods and fields for its supertypes are also shown. Figure 6 shows an example of this.

Once the above information is displayed, the user may then elect to alter the state of an object or primitive variable, or invoke a method. Invocation of methods will cause the system to first check if arguments are required, and if they are not then the method will be invoked and a return value and optional execution trace returned. If arguments are required, then the interface prompts the user to supply objects or primitive values for each argument and then will proceed as above.

The execution trace created (if requested by the user) will be information derived from events detected by the JVMDI code in the Program component, and filtered by the Mapping component to be in a form usable by views in the Visualization component that are responsible for displaying given information in certain formats (for example, a Stack view would draw the information passed from the Program component via the Mapping component as a graphical stack of elements).

To allow for the fact that the user may want to review visualizations without having to re-execute the method and re-set its state, the visualizations are stored separately. Dyno uses the serialization API was used for this purpose. Serialization supports the reading and writing of objects and primitive values over a stream, of which a file is an example.

Figure 7 shows a simple Stack view in action, here we see something similar to a tape deck with facilities to *play*, *rewind*, *fastforward* and *stop* the current animation. A progress bar shows how far

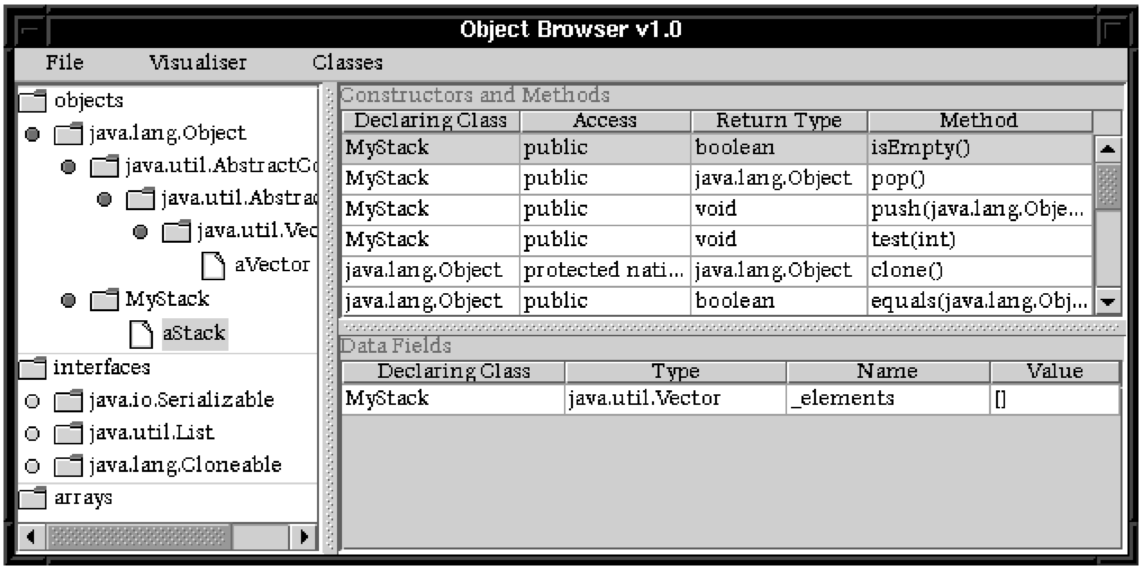

Figure 6: *The main interface of Dyno, showing the available classes and objects in the system. The "aStack" object is currently selected and its methods and fields are shown in the tables to the right.*

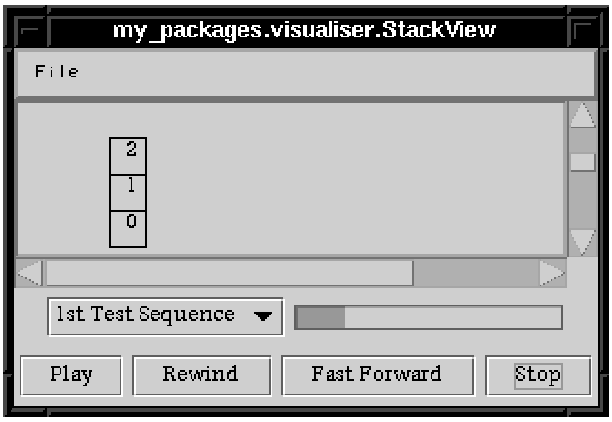

Figure 7: *A view of a stack, about one fifth of the way through its animation. The control panel allows for moving in both directions through the animation and gives the user some idea of the current position in the sequence.*

through the current recording the user is and there can be more than one recording per file, hence the selection box to the left of the progress bar.

The VisualStack class (implementing the Stack view) would have methods relating to the the operations you would expect with a Stack, where each VisualStack method creates an animation or new picture of the state to depict what the Stack method is doing: an element being pushed onto a stack (in the case of the "push" method), or being popped off the stack (in the case of the "pop" method).

6 Discussion

The two prototypes tools, Dy-re and Dyno, differ in many details. Dy-re addresses programming *for* reuse, but Dyno addresses programming *with* reuse. Dy-re works with a whole program and the program's ordinary user-interface, but Dyno works with a particular component and a user interface created to match the programming interface to the component. Dy-re displays the program execution with a small set of standard visualizations applicable to any program, but Dyno displays the component execution with visualizations tailored to the component, or selected from a library of appropriate visualizations. But despite these differences, the two tools had a common set of requirements for tracking dynamic execution. The essential requirements were control of execution, and access to data.

The two debugging systems, Gdb and the JVMDI, also differed in their approach. Gdb is used as separate program that deals with the target program, and its interface is based on text interaction, with user commands and system responses. The JVMDI is used as a library that is linked with the interpreter of the target program, and the JVMDI functionality is available via method calls within the program.

However, both Gdb and JVMDI provide the essential requirements. Gdb allows the program to be breakpointed at function entry, function return, at C++ statements, or at more detailed levels, whereupon the target program will pause at those points. The JVMDI allows breakpointing of instructions inside methods, and also can pause when a frame is popped from the run-time stack. Together these allow Dyno to take control from the target program at method call or method return.

Gdb allows access to static data, data in stack frames, and data on the heap, and allows control of the format used to present the data. Gdb also shows data passed as parameters to functions or returned from function calls. The JVMDI allows access to the fields of any object, including those in local variables, on the stack, and parameters passed to methods. Surprisingly, however, values returned from methods do not seem to be accessible. This data is necessary for visualizations in Dyno, and seems an unfortunate omission. Our work-around is to require method returns to be preceded by a call to a special method to enable detection, with the return value is passed as a parameter.

In addition to the essential requirements of execution control and data access, our approach implies two similar requirements of practical importance. The practical control requirement involves control of the debugging system itself. Using the JVMDI involves linking directly with the target program, and so the JVMDI access is simply part of the larger program, and the JVMDI provides methods to control its behavior.

Using Gdb involves textual interaction with the interface to the Gdb program. To control the textual interaction by means of a program requires a program to communicate with Gdb by passing it text commands, and analyzing text responses. Fortunately, Expect makes it easy to create such a program. Moreover, Gdb itself offers command facilities that ease such control. Not only do both commands and responses use syntax that is easy to deal with, but they also allow use of regular expressions that make it easy to issue commands that apply to many functions or data items.

We were initially concerned that using text control with Expect might make Dy-re too slow. If that had been true, our plan was to modify the Gdb source to directly link it with Dy-re. However, tests showed that the text interface did not affect the speed of Dy-re enough to notice any difference. The text interface also had the benefit of allowing Dy-re to be used on different architecture systems without even re-compilation. Moreover, working with the text interface and Expect proved easy and led to rapid development. One reason for this was because experimentation was possible using direct textual interaction with Gdb, and this led directly to successful Expect scripts.

The practical data requirement that arises when using debuggers involves the structure of the program or component itself. Both Dy-re and Dyno need to know the names of classes, and their constituent parts, in order to work with the debugging systems. In Dyno we use the Java Reflection API, to determine the classes, methods, and fields. So although the JVMDI itself does not provide the details, this is not necessary. Gdb does make it possible to find these details, so in Dy-re we simply use Expect to interact with Gdb to find all the class names, and so on. An alternative approach for both tools would have been to use static analysis and parse the source code of the target programs. We have other tools that do such static analysis, but both the approaches taken with Dy-re and Dyno are much easier.

7 Conclusions

In this paper we described two prototype tools that use visualization to better support reuse in a programming environment. However, we focussed here not the design of these tools, but rather on their implementation. The tools required access to running programs, and instead of tackling this arduous low-level development task, we reused debugging systems.

We view the approach as generally quite successful, most importantly because we were able to develop the tools rapidly while concentrating on issues of tool design and not low-level program tracking. We did have some problems, and in particular two familiar lessons about software reuse were made clear to us once again. The first lesson was to be very careful to ascertain that the reused software does indeed fulfill all requirements, which in the case of the JVMDI was surprisingly not completely true. The second lesson was that it is a significant task in itself just to understand how to use a complex component, such as a debugging system. Ironically, this is a problem that one of our tools was itself designed to alleviate. However, especially in the case of Gdb, it proved very helpful to be able to explore the functionality interactively. Interestingly, this is also the same basic approach taken by our tool.

As far as we are aware, neither of the debugging systems we used was intended to be reused as we have done. This means some problems in reuse should be seen as reasonable, and in fact the ones we encounted were minor. One potentially difficult problem was that Gdb was designed for interactive use with a textual interface. However, Expect allowed us to script the textual interface from within our tool. For those wanting to program with reuse, the general lesson is to consider reusing software if it provides critical and complex functionality, even if that functionality was not intended for reuse. The specific lesson is that program visualization can be made a lot easier by reusing debuggers.

References

[BT96] 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.

[Lib94] Don Libes. X wrappers for non-graphic interactive programs. In *Proceedings of Xhibition '94*, San Jose, CA, June 1994.

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

[MMM95] Hafedh Mili, Fatma Mili, and Ali Mili. Reusing software: Issues and research directions. *IEEE Transactions on Software Engineering*, 21(6):528–561, June 1995.

[NGB95] James Noble, Lindsay Groves, and Robert Biddle. Object oriented program visualisation in tarraingim. *Australian Journal of Computing*, 27(4), November 1995.

[RC93] Gruia-Catlin Roman and Kenneth C. Cox. A taxonomy of program visualisation systems. *IEEE Computer*, 26(12), December 1993.