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 4 463 5341, Fax: +64 4 463 5045
Email: Tech.Reports@mcs.vuw.ac.nz
http://www.mcs.vuw.ac.nz/research

Exploration and Visualisation of Reusable Components Using Java

Stuart Marshall, Robert Biddle, and Ewan Tempero

Technical Report CS-TR-99/4
April 1999

Abstract

Many aspects of program development and understanding involve the dynamic behaviour of the program. In order to develop or use program components, it is often important to experience the dynamic aspects of the components, whether for testing, debugging, or familiarisation. For example, it might be useful to facilitate testing of components by creating objects, and then calling methods, specifying parameters by filling in a form. Understanding of dynamic behaviour might be supported by automatic creation of diagrams tracing program execution through different components, or by automatic connection of the code to a program visualisation system. This paper reports on our exploration of how this approach can be supported using Java, the Java Reflection API, and the Java Virtual Machine Debugger Interface.

1 Introduction

Many approaches are being pursued to improve the effectiveness of software development. Two important approaches are *reuse* of software components to reduce the programmer's work, and the development of *tools* to aid the programmer. We believe that achieving effective software reuse is a difficult problem in itself, one that requires more support than has generally been available. We are exploring the development of tools designed to explicitly support the reuse process. In this paper we introduce Dyno, a prototype tool that allows programmers to "test-drive" a software component, and show in detail how Dyno can be used to explore and visualise component behaviour.

There are two aspects of software reuse: the production of the reusable software, or developing *for* reuse, and the process involved in reusing it, or developing *with* reuse. Our previous work has concentrated on support for developing *for* reuse [BT98, MWBT98]. The work we describe here is supporting developing *with* reuse. In particular, we have been exploring how to best help a programmer understand the reusable software well enough so as to be able to use it effectively. Dyno provides support for reusing code by allowing the programmer to explore the behaviour of a Java component interactively, aided by dynamic visualisation. In doing so, it makes extensive use use of the Java Reflection API, and the Java Virtual Machine Debugger Interface.

The paper is organised as follows. Section 2 discusses motivation for the development of Dyno, and from that section 3 discusses our approach to design and implementation. Section 4 shows Dyno in action with detailed commentary. We present our conclusions in section 5.

2 Motivation

The main motivation for this work is to provide better support for trial use, or "test-driving" of a software component. Writing trial programs, or "test-harnesses", to explore how to use a component is a common practice. For object-oriented programs, such programs typically invoke the methods of the public interface of an object and then display the results returned and the resultant state of the object so the programmer can check they are consistent with expectations. Of course, this is not a substitute for understanding of a component specification, but can be of assistance in better understanding practicalities of actually using a component. Test-harness programs are typically not very sophisticated, but producing them tends to be tedious and time-consuming.

Our ultimate aim is to provide better support for software reuse. One of the barriers we see to reuse of software components is the effort required by the programmer to understand components well enough to choose them correctly and use them properly. The approach we are exploring is to give the programmer the ability to easily "test drive" a component — and "twiddle the knobs" to see what it does. The approach would be similar to using a test-harness, but ideally we would want the test-harness to be available automatically, and not need to be written by the prospective component user. Moreover, we want all this to be possible in the typical component library situation, where the component source code is not available.

To increase the effectiveness of the test-drive technique, we have also been exploring use of visualisation and animation to aid a programmer in understanding what a component does and how it does it. We hope this might help give a programmer a better appreciation of how a component can be reused and where such reuse would be appropriate. The idea behind software visualisation is to tap into the human ability to process complicated visual stimuli by creating pictorial representations of the code at a reasonably high level of abstraction at various points during execution. Such pictures are typically easier to understand than the linear text that comprises most code. With animation, visualisation offers a dynamic view of the code in action, rather than just a static view.

3 Approach

To explore these ideas, we developed a prototype tool. The first aim of the tool was to allow the programmer to test-drive a component without the the need to create custom test-harness code. The second aim was to provide visualisations of the component behaviour, to make use of the component more understandable. We decided to address Java components, both because Java was the language we were using in other research and teaching, and because the run-time environment of Java looked as if it would provide useful support for our tool. We named our tool "Dyno", the usual abbreviation for dynomometer, which is a kind of machine used to provide an artificial load and allow performance monitoring of engines.

In either role, as a test-drive tool or as a visualisation tool, Dyno should require no access to or modification of the source code of the component being examined. Our ultimate aim is to facilitate reuse of components in libraries, and typically source code is not provided. Moreover, modification of source code could easily lead to problems with language compatibility and component maintenance. Finally, Dyno should endeavour to follow Java's "write once, run anywhere" philosophy. In particular, it should require no modification to the Java virtual machine.

The design of Dyno addresses both the test-drive requirements and the visualisation requirements. The design links the two aspects together, because the test-drive facility also acts as a front-end to the visualisation facility. In this way, a user may browse the interface of a component, call methods and inspect results, and optionally view visualisations of the method execution. The implementation of Dyno makes use of several aspects of the Java run-time environment, including features allowing reflection, debugging, and serialisation. The details of design and implementation are discussed below.

The test-drive facility of Dyno is provided by the Java ability to dynamically load classes, and by the Java Reflection API. While running, Dyno can be directed to a Java component, and dynamically load the classes described in `.class` files. The Reflection API allows for introspection of objects and classes. Using the classes, Dyno can then dynamically create new instances or arrays of a class, invoke methods, and can access and modify fields.

Together, dynamic loading and reflection are all the capability needed to support component test-driving. The user can select a component, the classes are then loaded by Dyno, and then the classes, methods, and fields can then be browsed by the user. Objects of any class can be created, and methods called on these objects. Dyno can detect what parameters are necessary, and prompt the user with forms to fill in. When methods return, Dyno can display the results. Forms and displays are simple for primitives such as integers or characters. When objects must be passed as parameters, Dyno allows the user to select an existing object, and when methods return objects, the user may request the object be saved for later use.

Support for detail visualisation in Dyno is implemented using the Java Virtual Machine Debugger Interface (JVMDI), introduced in 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.

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 for further processing so that visualisation 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. 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 on so that visualisations can be created.

Most of Dyno is written in Java and so is as portable as the components it supports. However, the JVMDI facility is accessible only through the JNI, and here Dyno uses C++. This code is compiled as a library, and is invoked from Java code using a stub method of a class that is prefixed with the `native` keyword and matches the name of a method in the C++ code. Unlike the rest of Dyno, the JVMDI part thus does need to be explicitly created for each machine architecture, and relies on the portability of C++.

4 Introduction

We now move on to illustrating how interaction would occur between Dyno and a user interested in exploring the capabilities and structure of some component. The particular scenario that we will be using is a scenario in which a user is looking at understanding what a component called *MyNetwork* does and how it does it. *MyNetwork* implements some functionality that enables a programmer to send and receive simple messages over the internet. The user will do this by exploring the possible methods that may be called on an instance of *MyNetwork*, and by looking at how the state changes and what the state comprises of. Then, the user will attempt to create animations representing actions performed on the instance of *MyNetwork* that show what methods are invoked, and when to fulfill the user's requested actions. By seeing these animations, the user can gain a better appreciation for what the *MyNetwork* component is doing and how it is doing it. Multiple views can give multiple perspectives, and the user can also keep the mapping information stored on file for later use, maybe as an additional form of documentation in showing a fellow programmer how *MyNetwork* works.

4.1 Initiating a Dyno Session

Dyno is a Java application. The user may start Dyno in a manner consistent with the standard procedure for starting up Java applications in the user's current environment. Having started a Dyno session, the user will be presented with the main screen as shown in Figure 1. From here the user has access to a menu bar that may, amongst other things, be used to load Java classes, create instances of Java arrays, and handle the Visualization capabilities. The rest of the main screen is split into two main areas.

4.1.1 The Object Cache & The Object Cache Browser

We use the term *Component* to refer to a Java class or Java interface that we load into Dyno. Furthermore, we use the term *Entity* to describe an element from the set of components and the instantiated instances of these components. The place where these entities are stored in Dyno is referred to as the *Object Cache*. The left hand side of the screen in Figure 1 contains the *Object Cache Browser*, through which the user may select which entity to inspect. The

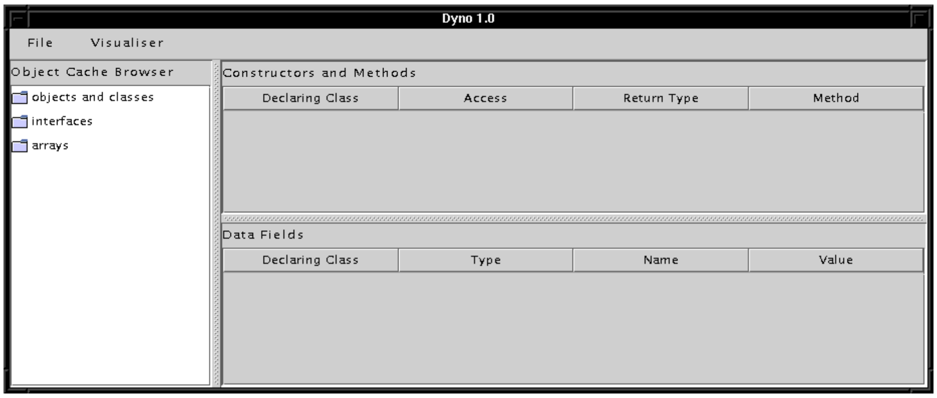

Figure 1: Dyno's main screen that the user is presented with on startup.

object cache browser is organized as a tree-like hierarchical structure. The three subtrees at the top level represent the three categories that entities in the object cache may be organized into.

The three categories are:

- Objects and Classes
- Interfaces
- Arrays

A component can be loaded into Dyno from a Java class file or from a previously saved object cache read from a file. Instances of a component result from the return values of methods and also from invoking constructors in the components. The subtrees are ordered hierarchically using a combination of the Java Inheritance model for the components and by ensuring that all instances of the components are seen as child nodes of the node representing the instantiated component.

Some entities may exist in more than one of the three subtrees. This might occur if an entity was an instantiation of a component, in which case it will appear in the "Objects and Caches" subtree, and also conforms to a particular interface, in which case it will appear in the "Interfaces" subtree under the relevant Interface node. The purpose of grouping the entities in this manner is to aid the user in finding an entity that conforms to a certain type for use in method invocations later on, and as a convenient way of grouping like entities together.

4.1.2 The Entity Information Browser

The right hand side of the main screen holds the *Entity Information Browser* and its window is split horizontally. The top half contains a table that lists any methods or constructors that can be performed on the currently selected entity in the object cache browser. If nothing is selected, this table is empty. The bottom half contains a table listing the fields contained within the selected entity, or likewise blank if no entity has been selected. If the entity selected is a component, then only the static methods and static fields will be shown. If the entity selected is an instance of a component, then only the non-static methods and non-static fields will be shown.

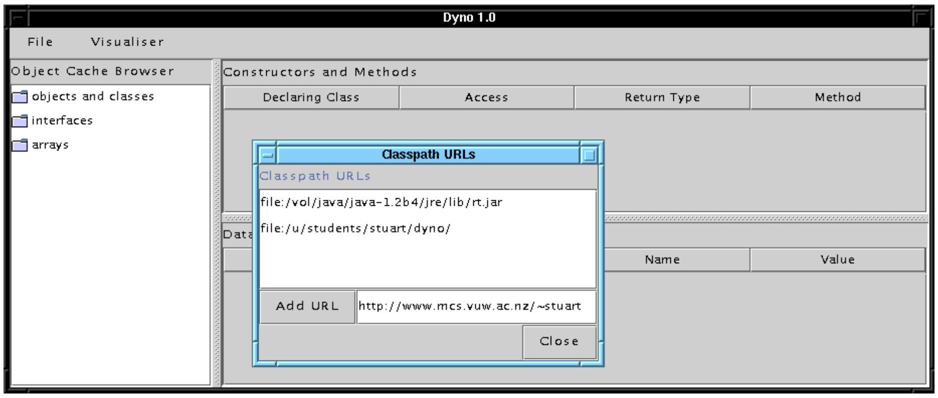

Figure 2: Interface for adding new URLs to the current Classpath, enabling Dyno to find classes later specified by the user.

4.2 Loading A Component

We have discussed how the main screen is laid out. We demonstrate this layout by using our example whereby the user decides to load a component called *MyNetwork*.

Loading *MyNetwork* requires at most two steps. Firstly, the user should check to see if Dyno's list of URLs that it uses to search for class files contains the URL that points at the directory or Java Archive file (known as *jar* files) where *MyNetwork* is stored. This can be done by going to the *File* menu and selecting the *Add To Classpath* option. This will then bring up a new window that contains a list of the URLs that Dyno knows to look at and a text field and button with which to add new URLs.

In the example we are following the URL of the directory where *MyNetwork* can be found is not currently listed. The URL in question is "http://www.mcs.vuw.ac.nz/~stuart/". The user types this URL into the text field as demonstrated in Figure 2, and then either hits the *Return* key or clicks on the button to the left, at which point the list box will be updated to include the new URL.

Having now been satisfied that Dyno knows which directory to look in, the user then proceeds to select *Load a Class* from the *File* menu. At this point the user is presented with a simple dialog box requesting the name of the component that the user wants to load. As components can be loaded from anywhere on the Internet or local file system that the user has access to, Dyno does not offer the ability to browse through all available class files so as to simplify Dyno's implementation. Having supplied Dyno with the fully qualified class (or interface) name, in this case just *MyNetwork*, via the usual form of interaction with the dialog box, the system uses the URL list to find the class. Failure to find such a class will result in feedback to the user noting that the name is not valid. However, in this case the user correctly typed in *MyNetwork* and, as the correct URL has been added at an earlier stage, Dyno is able to find the *MyNetwork.class* file in the directory pointed at by "http://www.mcs.vuw.ac.nz/~stuart" and loads it into the system. At this point the object cache browser is updated so that *MyNetwork* is visible, as can be seen in Figure 3.

4.2.1 Specifics of the Entity Information Browser

The user has now added a component to the object cache, which has resulted in an automatic update of the object cache browser. The class *MyNetwork* inherits directly from *java.lang.Object* and a node in the subtree whose root is "Objects and Classes" appears for *java.lang.Object* and a child node of that node appears for *MyNetwork*.

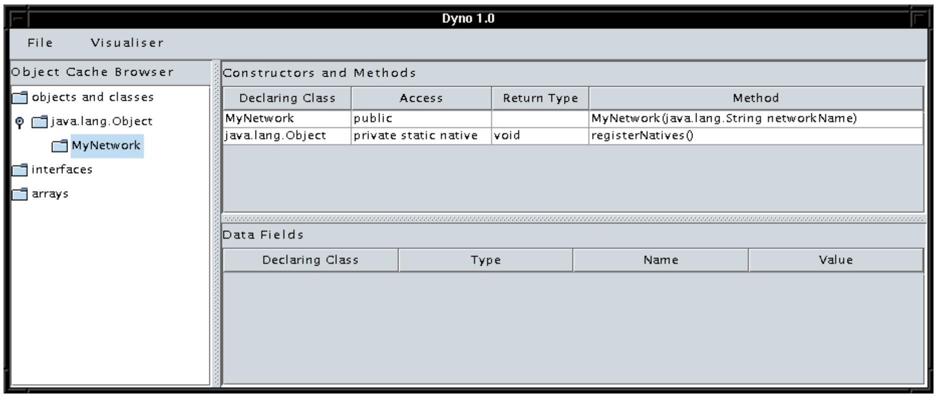

Figure 3: Main Screen updated for the now loaded MyNetwork component.

Any other entities that are later loaded that inherit directly from *java.lang.Object* will also appear as children of the *java.lang.Object* node along with *MyNetwork*.

The user clicks on *MyNetwork* in the object cache browser and Figure 3 shows the updated screen. *MyNetwork* is a component (by virtue of being a Java class) and so only the static methods and fields are shown.

The information that can be seen in Figure 3 includes the access properties of a method, constructor or field, the return value in the case of a method, the class in which it is defined (in case it is inherited from a super class) and the signature (name and, for methods and constructors, the parameter list).

4.3 Creating Instances of Components

The user decides to go ahead and create an instance of *MyNetwork*. This can be achieved by double-clicking on a row containing the name of a constructor in the method/constructor table. At this point there are two possible responses. If the constructor requires no parameters then Dyno will have asked for confirmation that the user wants to invoke that particular constructor. It would then invoke the code, alerting the user upon completion of the constructor that a new object had been created and requesting a name for the object (analogous to a variable name in normal programming) for future identification in the object cache. The instance is then placed in the object cache and the object cache browser updated in the same manner as previously described for components.

In this case however, the user has selected a constructor that takes a single *java.lang.String* argument. Dyno now prompts the user for the value of this argument is and displays a window (shown in Figure 4) that lists the argument types and names and asks for literal values or for the names of an object of a conforming type. If the user were to specify an object name rather than a literal value, then that object name must match a name already stored in the object cache.

The user doesn't have any *java.lang.String* objects or any of a conforming type, so the user decides to enter a literal value. The name of the parameter is "host" so the user decides to go with "debretts.mcs.vuw.ac.nz". The user clicks on *Ok* and now gets the confirmation message as mentioned above and proceeds from there. The result of all this is that the user now has an instance of the *MyNetwork* class. The user has called this new object *myNetworkObj* and this is displayed in the object cache browser as a child of the *MyNetwork* node.

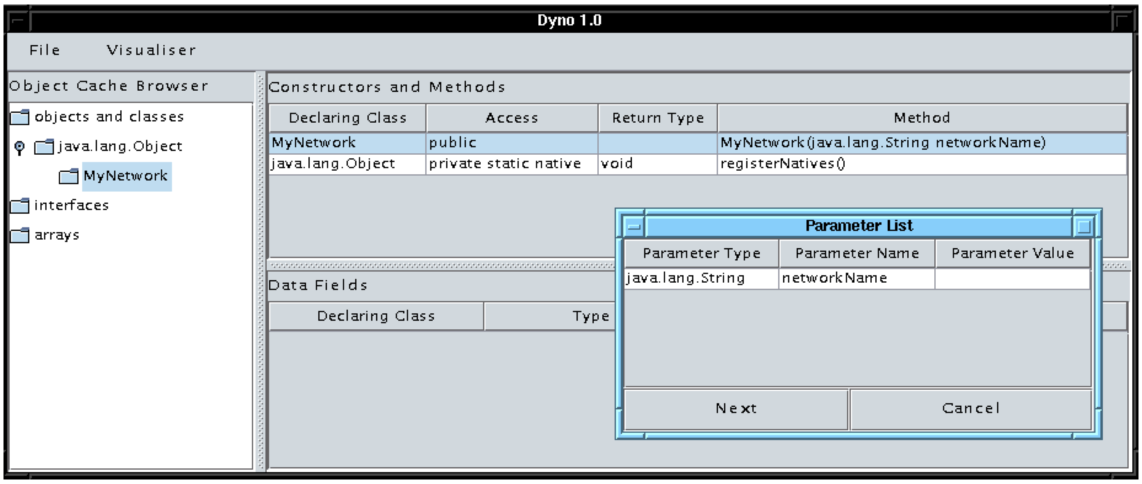

Figure 4: An interface that allows the user to specify values for parameters to be passed to a method or constructor.

4.4 Browsing Components and Their Instances

Selecting the *myNetworkObj* node updates the entity information browser to show the instance methods and fields only. The user can browse through the list of methods and fields at will and may derive some useful understanding of the class from the naming scheme used in the class.

The user decides to go one step further and test-drive the *MyNetwork* class by invoking some of the methods and seeing what happens. Invoking instance methods is done in the same manner as the invocation of the constructor. If the implementation of the method requires any interaction with the user then Dyno will let that interaction occur as normal as if Dyno was not there. The only situation where Dyno makes its presence felt during the method invocation would be if the method implementation resulted in an exception being raised, at which point Dyno will bring up a message box identifying the type of the exception and any other useful information. The user must remove this message box before the method can continue executing (if it is capable of continuing).

Upon the method running to a successful completion, Dyno will again alert the user of any returned values and allow the user the opportunity to save those return values to the object cache for later use in other invocations or for further study.

If the method resulted in the state of the object being modified, then the fields in the fields table will be updated to show these new values.

4.4.1 Browsing Through Data Fields

The user can continue to invoke methods in this manner so as to get a feel for the side-effects that result, both in the shape of the return value and in the nature of the modifications to the fields. Extending from this idea of inspection, the user can also look in more detail at the fields of an object or class. As shown in Figure 5, the fields are displayed in a single entry, the size restrictions of which limit how much the user can learn of its true value. An object value in a particular field may have several fields in its own state. However, unless the field is a primitive type, or one of a select a few classes that implement a more extensive way of printing its own state (examples of which are *java.lang.String* and *java.util.Vector*), what is shown will typically be merely the class name and a unique hash code to distinguish it from other instances of the same class.

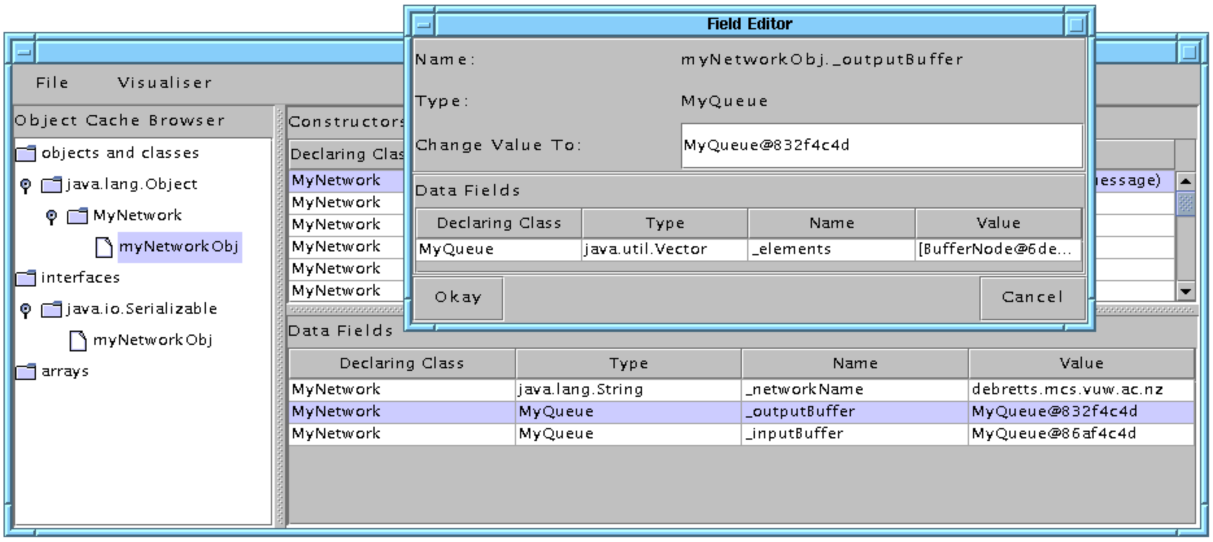

Figure 5: This interface allows the user to browse and edit a field's own internal state, to add a finer level of control over the state of the component under inspection.

In our example *MyNetwork* class, there are two private fields that demonstrate this behavior. Both _outputBuffer and _inputBuffer are instances of MyQueue, however from how they are shown in Figure 5, it is not possible to get a good appreciation for what their state is beyond the fact that they are different instances as they have different hash codes following the class name.

The user wants to know more about _outputBuffer, more specifically what is currently stored in the queue-like structure. The user double-clicks on the _outputBuffer's row in the data fields table and is then presented with a new window, as seen in Figure 5. This new window lists the fields that comprise _outputBuffer (had the user double clicked on a primitive type, then the table would have shown no fields but merely repeated the primitive value). At this point the user can see the internals of _outputBuffer and get a better appreciation for what is in it at this present time. The user can recurse even further, double-clicking on fields in this window, thus causing a new window of a similar style to appear, showing the state of the field just chosen.

4.5 Creating Animations

The user has now had a chance to invoke methods and investigate the state thoroughly. However this is only one part of Dyno's functionality. As well as enabling the user to invoke certain methods of a component, the user may request that Dyno create visualizations based on their interactions with the component.

The visualizations are written ahead of time in Java. We refer to these Java classes as being *Views*. A view contains methods that may be invoked by Dyno to create animations that represent the actions (method invocations, method returns, field accesses or field modifications) in the components the user has performed. Certain actions may result in a particular method in the view being selected, a number of different types of action may result in the same view method being called, and a number of different view methods can be called by one action. As well as this, the views can get state information from the executing component and use this in the animation. In the example we use, the View the user selects is a TraceView showing an execution trace in a UML style Sequence Diagram. An execution trace is only one of the types of visualizations that Dyno can handle. A view could also represent a graph

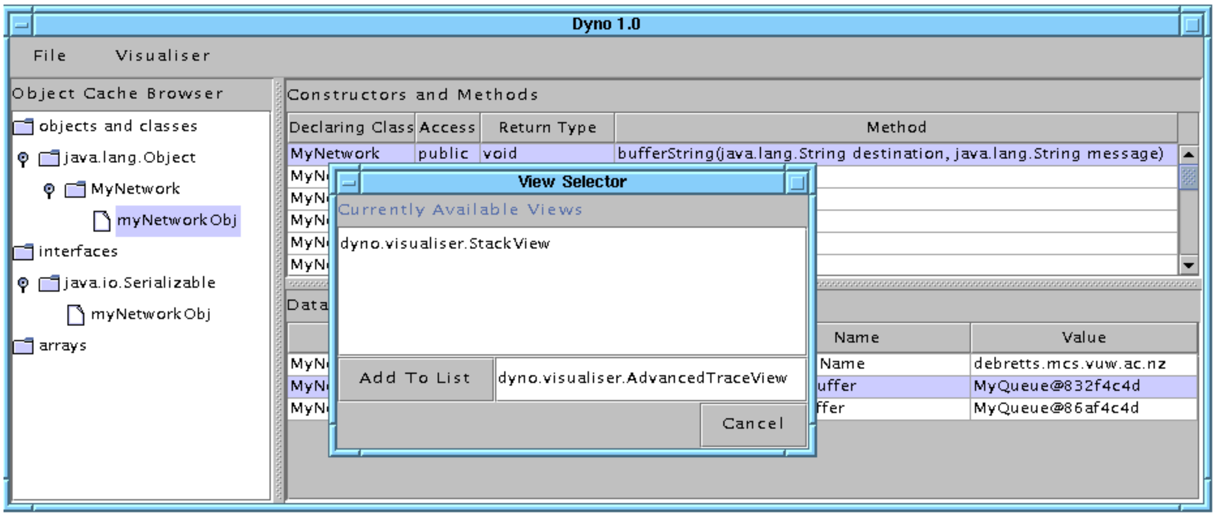

Figure 6: This interface shows a list of the views that Dyno currently knows about. These views are needed not only to create animations, but to give the user more information about what is needed in the way of mappings.

animation, a tree animation, a stack animation or some more user-specific animation written with a particular component in mind. Dyno does not offer an editor or compiler with which to write these views, but there are classes within the Dyno implementation that the person writing the views can use to aid in retrieving recursive or historical state information.

4.5.1 Supplying Dyno With Mapping Information

For Dyno to be able create such visualizations, the user must first give Dyno some information to enable it to map from actions in the component implementation to methods in the selected view, and also identify the parts of the component state that the view should use when presenting information in the animations. The user decides they want to create a execution trace visualization of *myNetworkObj*. The user already knows of the existence of a view class that animates the information passed to it as an execution trace. This view is called the *TraceView* class. For the user to use *TraceView*, they must first check to see if it has been currently loaded into the system. This is achieved by going to the *Visualizer* menu and selecting *New Filter*. A *Filter* is a representation inside Dyno of the information needed to be able to map from the concrete implementation details in the loaded components to the methods and state used by the views. Dyno can detect events in the runtime system that relate to method invocation, returns, field access, field modifications and exceptions being thrown. When Dyno detects events in the runtime system, it first passes it through the filter to determine if the event is relevant to the view requested and if so, in what way.

The user will be presented with a new window similar in style to that which the user saw when adding a new URL to the classpath earlier in the session. Figure 6 shows the result. The list at the top shows the currently loaded views, of which *TraceView* is not among them. The user types *TraceView* into the text box, either hitting the return key or the button to the left upon completion. *TraceView* is now loaded into Dyno and the user can now create a new filter that will collect information on state and actions for that kind of view to translate into animations at a future date. The user does this by double-clicking on the name in the list.

Having done this, yet another window will appear that is split into two halves as shown in Figure 7. This window lists the methods in the view class and the attributes that it uses

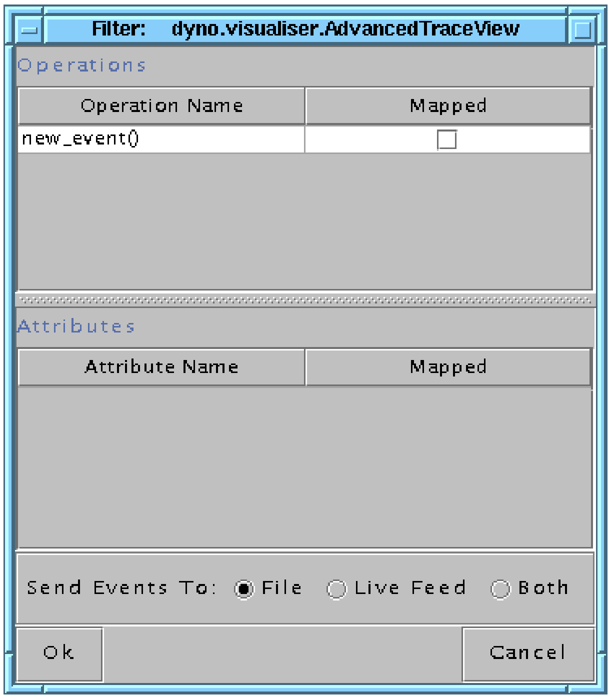

Figure 7: Having selected the TraceView, the user is presented with a list of the attributes and methods in that view. These must then be mapped to some counter-part in the component under inspection.

to draw the animations resulting from those methods being called. The attributes are used as places on which to map state from the user's component. The TraceView does not contain many methods or any attributes. The methods are listed by name and have a boolean tick-box associated with them, that shows a tick if there are any actions mapped to that method.

Mapping To View Attributes If the view had attributes then they would be listed by the name the view writer had given them, the type of data needed by that attribute (whether a *java.lang.String*, *java.util.Vector* or some other object or primitive value) and a boolean box which shows a tick when something has been mapped to that attribute by the user. For a filter to be valid, all attributes must have something from the user's components that maps to it. However not all methods in the view class need have actions mapped to them.

Mapping to View Methods As figure 7 shows, TraceView only has one method, *new_event*, that needs to be mapped. Remembering back to earlier in this section, an action can be the invocation of a method, the return of a method, the accessing of a field or the modification of a field.

Double-clicking on *new_event()* will result in the user being presented with a window through which they can add actions to a list stored against this method, as seen in Figure 8. The list is currently empty and there are buttons on the left hand side that allow the user to manipulate the list. The user decides to add a new action to this list and so selects the Add button.

Figures 9 & 10 show the interface that allows the user to specify the nature of the action being mapped. There is a text field in which the user can type in the name of the target object on which the actions are being performed. The valid values are as follows: NULL, in which case the action later specified will be mapped if it occurs on any object, or the name of

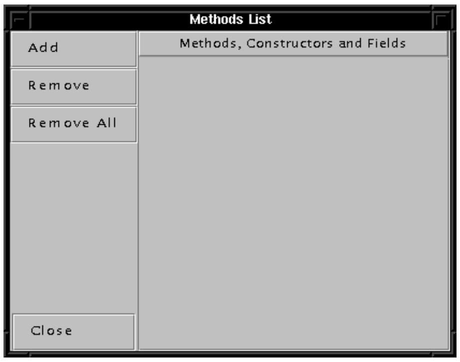

Figure 8: This lists the current mappings for the selected method in the selected view.

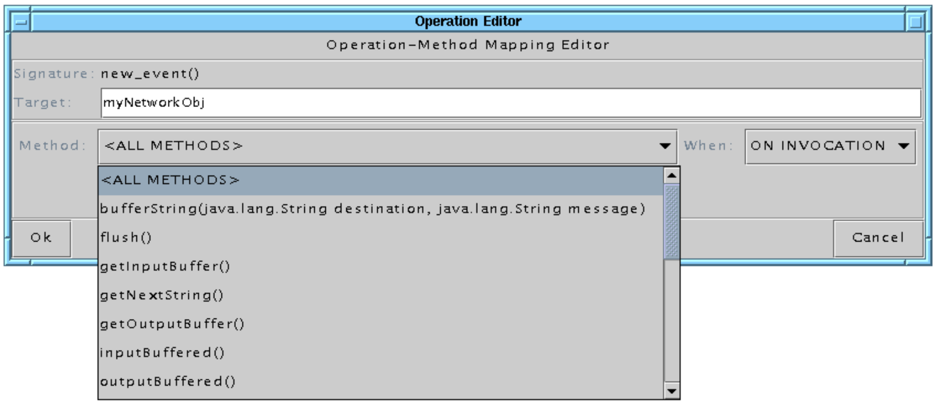

Figure 9: This interface allows the user to specify a mapping to a view method from methods on some component or component instance in the object cache.

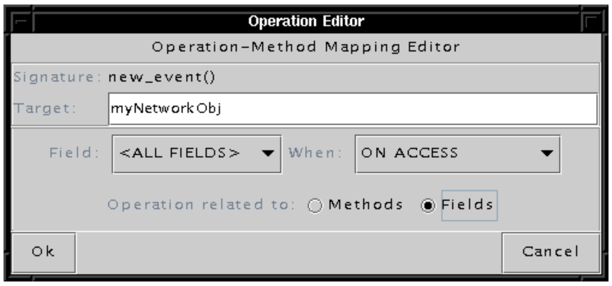

Figure 10: This interface allows the user to specify a mapping to a view method from events on fields on some component or component instance in the Object Cache.

an object currently stored in the object cache. As well as the text field, there are two check buttons that enable the user to switch between specifying a method-based action (invocation or return of a particular or all methods) and a field-based action (access or modification of a particular or all fields) on the object specified in the text field, or all objects if the text field is empty. Depending on which check button is enabled, the two popup menus also on the window will either list the methods and types of method actions (see Figure 9) or will list the fields and types of field actions (see Figure 10).

The user has one value currently stored in the object cache, called *myNetworkObj*, and so types this name in at the prompt, hitting the return key upon completion. Having now done this, Dyno updates the two popup menus currently on display. In this case, they are the method-based menus and the first popup menu now lists all of the methods that may be invoked on the *myNetworkObj*, i.e. all the instance methods. The second popup menu gives three options, "On Invocation", "On Return" or "Both". The user decides they want a new event to be signaled in the TraceView when the *flush()* method returns, so selects *flush()* from the first menu and "On Return" from the second. Having done this the user now clicks on *Ok* and the window disappears, and the previous windows that listed the mapped actions will be updated to include an entry for this mapping. The user has second thoughts however and decides that rather than just *flush*, they would like to have the TraceView know about all the method invocations and returns on *myNetworkObj*. The user could add a second mapping but since the first isn't needed they may as well modify that (or remove the first and then add a new one). The user double-clicks on the entry for the previously created mapping. This brings the operation-method mapping editor (refer to Figure 9 or Figure 10) back up with all the values set as the user had specified. The user now pulls down the methods list and rather than selecting *flush()*, decides to select the top-most item *<ALL METHODS>*, which is a special entry and not a method name. The user also changes the "when" condition from "On Return" to "Both". Clicking on *Ok*, the entry in the list of mappings now changes accordingly.

Clicking on *Ok* in the mapping list window will return the user back to the original Filter window that listed the actions in the view and any of its attributes (refer to Figure 7). As we have mapped something to *new_event()* there is now a tick against it.

As there are no attributes to map, the user proceeds to select where they would like the information that the system collects to be stored. The options are: to a file whose name will be asked for later, a live feed that opens up a *ViewScreen* directly and pipes the information to the animation code as the component is running, or both. The user selects the *file* option and then clicks on *Ok*. Dyno will then fire up a file browser window and ask the user for the name of the file they wish the information to be saved to. The user types in "mySaveFile" and clicks *Ok* on the file browser. At this point the user has now successfully given Dyno sufficient information to be able to animate the component based on a sequence of the user's interactions with the component.

4.5.2 Creating a Sequence of Actions

The user decides to put this to the test and invokes a few methods that they hope to see animated. Before doing so, the user must turn on the Event Trace Detector, by going to the *Visualizer* menu and setting *Event Trace* to be on if it is not already so. A tick will appear next to it when it is on.

Having done this, Dyno is now waiting for events to detect. Using the same procedure as before, the user invokes a number of methods on *myNetworkObj*, passing arguments through and saving return values as necessary. All the while, Dyno is storing all the relevant information on this sequence of interactions to the file "mySaveFile" for future use by a viewscreen.

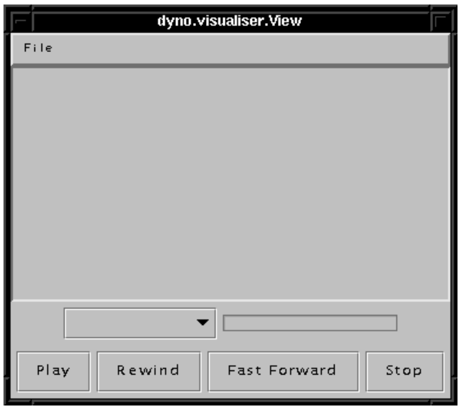

Figure 11: A blank ViewScreen, with no loaded file.

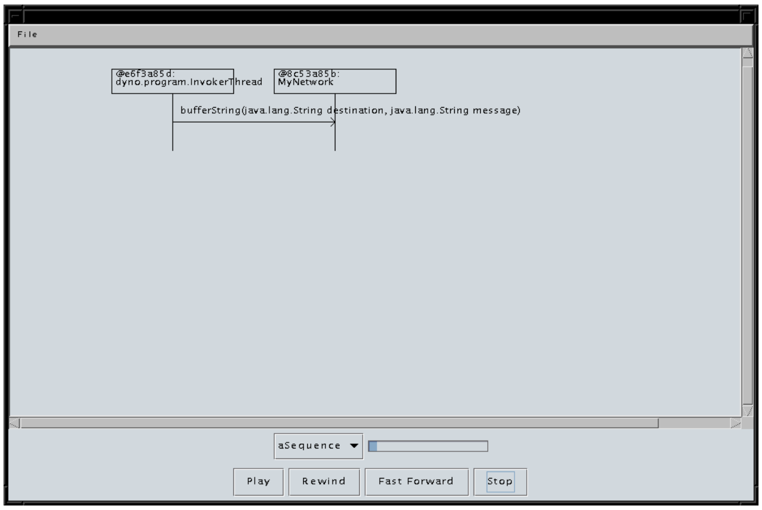

Figure 12: Early on in the Trace of the myNetworkObj. The trace is described in terms of a UML style sequence diagram.

After the user has finished the sequence of interactions they want animated, they go back and de-select *Event Trace* in the *Visualizer* menu. This will now prompt the user for a name to identify that sequence in the "mySaveFile" (as more than one sequence can be stored per file). The user types in "aSequence", clicks *Ok* and now has all the information that is required for Dyno to create an animation.

4.6 Viewing the Animation

The next step for the user is to then try and view the animation. They do this by opening up a viewscreen, which can be achieved by selecting *New View* from the *Visualizer* menu. This now brings up a viewscreen, as seen in Figure 11. The viewscreen works on a tape deck analogy, whereby the user has control over the chronological direction and speed of the animation via rewind, play, stop and fast forward buttons. The viewscreen has a main window where the animation will be drawn, which may be scrolled if the animation is bigger than the window size, as well a progress bar which shows how far through the animation the user currently is. The last thing on the viewscreen is a popup menu that lists all of the sequences in the currently loaded file (no file is loaded at this point in the example).

The first thing the user must do when presented with this screen is load in the information

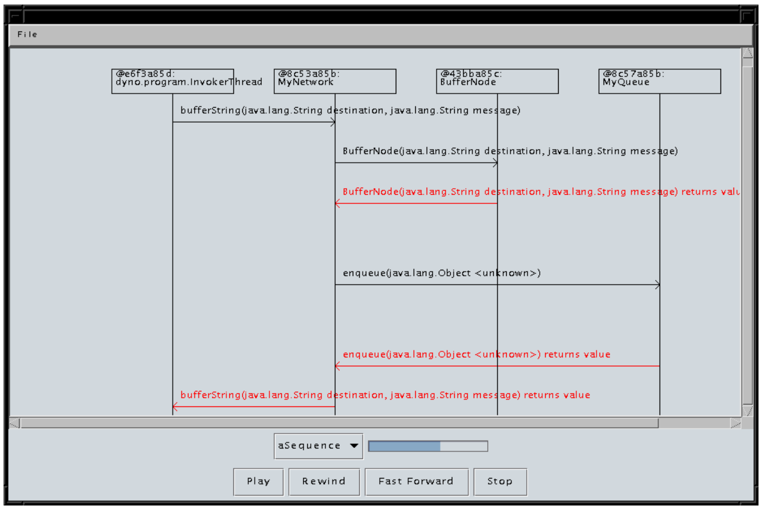

Figure 13: Later on in the Trace of the myNetworkObj. By this point several methods have been invoked and returned. The darker lines indicate a method invocation and the lighter lines indicate a return.

from the previously created "mySaveFile". The user can achieve this goal by selecting *Change File* from the *File* menu in the viewscreen. The *Change File* menu item will open up a similar file browser as was seen when the Filter asked for a file to save to earlier in the session. Selecting "mySaveFile" from the directory the user put it in, the user may then select the *Scan File* menu item from the *File* menu to retrieve all of the sequences that are stored in the file. The sequence names will then be put into the popup menu on the viewscreen display. As there is only one sequence in the file, that one sequence is automatically selected and the user may now play, rewind, fast forward or stop the animation at will. Figure 12 & Figure 13 shows the animation in action at different points in the execution.

The user can have more than one viewscreen open at any one time. They may want to do this because they have more than one different type of filter in the system. The user may have also given Dyno mapping information to map actions on one of *myNetworkObj*'s Queue private fields to a QueueView, or because they wish to see two different parts of the TraceView at the same time.

Dyno also allows the user to save or load the contents of the object cache by using the *Save Cache* and *Load Cache* menu items in Dyno's *File* menu. This opens up the standard Java file browser window and the user selects a file to save or load from in the usual manner. The object cache will only store those objects that are serializable, so some parts of the Cache may be lost between separate sessions.

Once the user has finished with either a particular viewscreen or Dyno, they can select the *Exit* menu item in the respective *File* menus. This completes our illustration of how interaction would occur between Dyno and a user interested in exploring the capabilities and structure of a component.

5 Conclusions

Our motivation for creating Dyno was to explore better support for test-driving software components, and to explore whether visualisation could be used to add further support still. Our prototype shows that the concept is coherent, and demonstrates that the Java run-time environment is sufficient for implementation of this kind of tool.

Dyno is really intended to help users of a component better understand how to use the component, and so is similar to a documentation system. For example, JavaDoc [AG98] is a documentation tool that scans the Java source code of a component, and then creates an HTML page to help the component user understand the interface of the component. This approach is certainly useful, but it requires the source code, and only produces static documentation. Dyno can work with the compiled code, and allows both interaction and animation resulting in a kind of dynamic interactive documentation. This form of documentation has the benefits of documentation embedded in source, in that it resides with the component itself, but without the costs of having to explicitly keep the documentation up to date.

Further work is now required in two areas. Firstly, we must explore the effectiveness of Dyno in actual practice with programmers considering larger and more complex components for reuse in new applications. Secondly, we must study whether our approach to providing visualisations for software component behaviour allows visualisations specific enough to be effective, but without requiring burdensome effort.

Our prototype successfully demonstrates the promise of the approach, and also shows how the Java run-time environment proves useful in the implementation of this kind of tool.

References

[AG98] Ken Arnold and James Gosling. *The Java Programming Language.* The Java Series. Addison-Wesley, 2nd edition, 1998.

[BT98] Robert Biddle and Ewan Tempero. Towards tool support for reuse. In *Proceedings of the Conference on Software Engineering: Education and Practice*, Dunedin, New Zealand, January 1998. IEEE Computer Society Press.

[MWBT98] John Miller-Williams, Robert Biddle, and Ewan Tempero. Rapid implementation of a program visualisation system. In *Proceedings of Uniforum New Zealand*, New Zealand, May 1998.