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

Dyno: A tool for dynamic interactive documentation

Stuart Marshall, Robert Biddle, and Ewan Tempero

Technical Report CS-TR-99/5
April 1999

Abstract

We describe Dyno, a tool that provides dynamic, interactive, visualisation of components for Java classes. Dyno is intended to provide support for the code reuse process by allowing the programmer to "test-drive" a code component. This allows the programmer to determine how suitable the component is for reuse, and to understand how to use the component correctly. In this way, it provides a form of dynamic documentation. Dyno uses program visualisation, particularly animation, to provide this form of documentation. It reduces the cost of providing such visualisations through extensive use of the Java Virtual Machine Debugger Interface, and the Java Reflection API.

Keywords: Tool support for reuse, Documentation, Program visualisation, Java, Reflection.

Publishing Information

Appeared in the Proceedings for the *First Symposium on Constructing Software Engineering Tools (CoSET'99)*, May 1999.

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 describe Dyno, a prototype tool that allows programmers to "test-drive" a software component.

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 [3]. 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 develops Dyno's requirements. Section 4 describes Dyno's design and implementation, while section 5 shows Dyno in action. Section 6 discusses our experience, and we present our conclusions in section 7.

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, and so producing them tends to be tedious. Nevertheless they can be time-consuming to construct.

For example, figure 1 shows a typical "test-harness" program. This is a Java application that was developed for use by our first year students to demonstrate implementations of a linked-list. The test harness code is about 300 lines of Java, and uses the Java Abstract Window Toolkit.

In some cases, a test-harness program may also require the programmer to modify the component being tested, for example to give access to the state of the object, which would otherwise be encapsulated, so as to enable the programmer to determine if the state is correct at any one time. Methods not available in the public interface may also need to be made available so the programmer can make sure that they are working correctly and that potential problems are not being masked by the public methods that use them. Encapsulation protects against bad design habits by not allowing programmers to become overly dependent on the implementation of a component, but it also acts as a barrier to thorough testing and exploration.

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

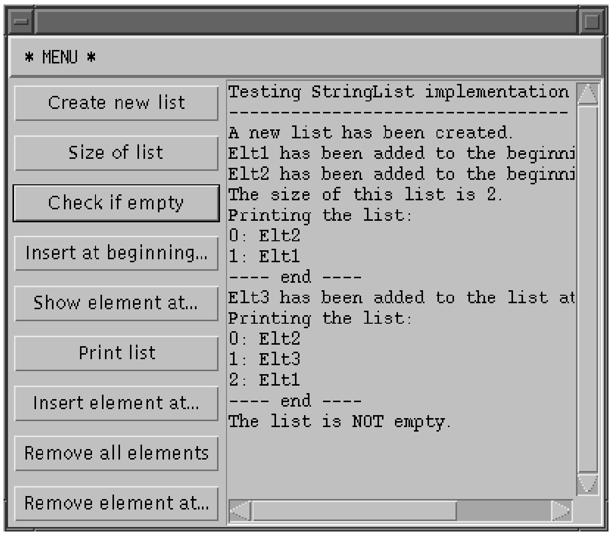

Figure 1: *Custom-built test-harness for a List class*

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. For the reasons given above, we also want the test-harness to be able to break encapsulation. 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 re-used and where such re-use 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.

Visualisation has typically been used to demonstrate how certain types of programs or algorithms work, and the motivation has been pedagogical (see, for example, [4]). We have also explored in this way, creating visualisations to demonstrate how certain data structures work; figure 2 shows our AVL-tree demonstrator in action. From our experience with these visualisations, we believe that the kind of understanding that students gain about the data structures is similar to the kind of understanding that programmers need to better understand software reuse.

Our early work in this area concentrated on how visualisation might help when developing *for* reuse. This work considered development of C++ programs, and used Gdb, the Gnu source level debugger, and Expectk [5], a combination of Expect and the Tk toolkit, to create generic visualisations and running programs [6, 2]. We were especially interested in how debuggers could be used to provide the necessary observability and controllability for visualisation [2]. Figure 3 shows one of the views the tool, Dy-re, provides. A Dy-re visualisation consists of icons for each C++ class, with arrows between the icons representing flow of control. The current call is shown using animation, and the current stack by different coloured arrows. Classes can be expanded to show calls made by individual member functions

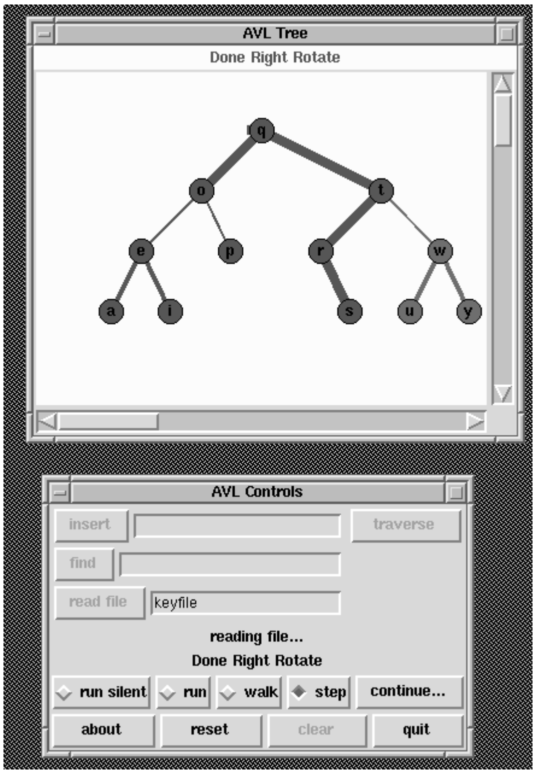

Figure 2: *Custom-built program visualisation for AVL-trees*

(on the left of the figure). We hoped Dy-re would assist programmers identify reusable patterns in their programs.

To support development *with* reuse, and the test-drive idea, a different approach is required. We will want visualisation of a component, rather than a whole program, and we will want visualisations tailored to the nature of the component, rather than general-purpose visualisations used in Dy-re. However, we are aware of the costs involved with creating custom visualisations, and would like to share visualisations and sub-visualisations between components, where reasonable.

Having established what we want, we should also be clear about what we do not want. We do not propose building a general purpose visualisation system. Building effective visualisation systems is still a significant problem, and not the main focus of our research. Instead, we believe we can get the benefits by providing a limited form of support for visualisation. Also, we are not proposing building a general purpose test support tool, although we do expect the system we're proposing will be useful for unit testing. Finally, at this stage of our research, we want to develop this system as stand-alone tool, and get experience with it, before considering using it in an integrated environment.

3 Requirements

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.

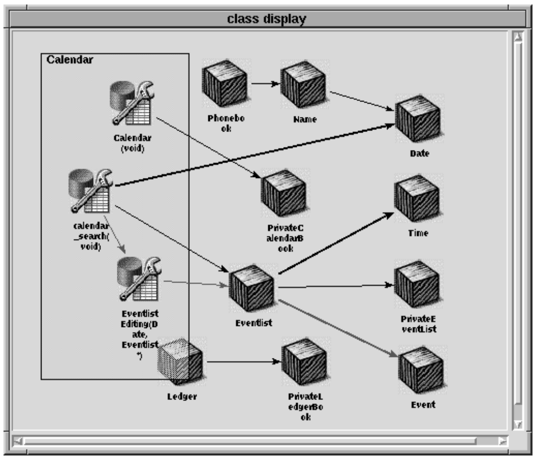

Figure 3: *Generic visualisations of C++ programs as provided by Dy-re.*

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.

The main test-drive functionality required is the ability to select a component, allow the programmer to inspect constructors, methods, and fields, and to call constructors and methods, allowing entry of parameters and inspection of the results. It should also be possible to maintain an object store, so that objects created for previous tests, or as the result of method calls, may be saved for use in further tests.

For the test-drive functionality to be really worthwhile, Dyno must allow a programmer to run a component through its paces in an easy and thorough manner. A hard to use interface that required complicated actions could result in more time being spent using Dyno than would have been invested in writing the old style test-harness, so negating much of the intended benefit of using Dyno.

As well as being easy to use, Dyno must allow for the programmer to test-drive all parts of the component, and observe the complete results and state. This may imply the need to break encapsulation, especially by allowing a programmer to directly invoke non-public methods and to observe non-public data members. It may even be useful to allow the programmer to modify non-public data members, so as to be able to put an object in a specific state prior to a test-drive.

Dyno should be resilient as possible to errors in component code. Uncaught run-time exceptions in a loaded component should not affect the running status of Dyno, and a programmer should also be able to interrupt execution.

The main visualisation functionality required is the provision of pictures that helpfully display the state and behaviour of a component as it is used. As we observed above, general-purpose visualisations are not sufficient for providing good documentation. This means that Dyno must provide some way to associate particular visualisations with particular components, allowing both custom visualisations where necessary, and also allowing choice from a library of useful visualisations. Ideally, useful visualisations could be added to such a library over time, making it easy for component providers to reuse an existing visualisation, rather than creating a custom visualisation.

Rather than needing components and visualisations to be associated too closely, there should be a facility allowing mapping between the details of component state and behaviour and the details of visualisation. However, there is also usability issue here. The information that must be provided to map from a component to a visualisation should be minimal and relatively easy to input via the interface. The harder this mapping is, the less likely any component provider will make the effort.

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.

4 Design and implementation

The design of Dyno addresses both the test-drive requirements and the visualisation requirements. The design links the two aspects together, by the test-drive facility also acting 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.

4.1 Test-driving

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.

In JDK 1.2, the Reflection API security policy was modified to allow access to private methods and fields, which would ordinarily be suppressed as being in violation of the access control given to such fields and methods. This enables Dyno to give the user direct access to private features, which would prove useful if they needed to test (for example) helper methods which were not part of the public interface, but needed to be tested in isolation from the public methods. This also allows the user to modify the internals of objects directly, to see how they behave with different states, without having to go through or be restricted by the public interface to the class.

4.2 Visualisation

The architecture for the visualisation part of Dyno is based on the PMV model created by Stasko and Roman and Cox, which describes an general architecture for program visualisation [9]. We are especially influenced by the extended version developed by Noble [8]. The PMV model is named after its three components: the program component, the mapping component, and the visualisation component.

Program component

The Program Component 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, such as:

- 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 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 (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 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++.

Mapping component

A visualisation 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 visualisation. The data model maps objects in the execution to elements in the visualisation.

The event filter is necessary because not all events will be important to a visualisation. The user may want to create a visualisation of a stack being used in an algorithm, therefore operations on the stack will have to be forwarded onto the visualisation 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 visualisation 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 Visualisation component. More than one event filter may exist at any one time inside Dyno, listening for the different events for the different views.

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 `Vector` field and will also map "size" to the return value of the size() method in the `Vector` class (determined dynamically whenever the snapshot is taken). When this data model is used by the Visualisation component, the stack view can then retrieve values for its "elements" and "size" and use them to display a visualisation 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.

Visualisation component

The Visualisation 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 Visualisation component may consist of multiple views at any one time.

A single view in the Visualisation component reads in the information it needs for the attributes from a stream 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.

The views are written as Java classes that extend a Java class present in the Dyno distribution called *View*. *View* itself inherits from the Abstract Window Toolkit class *Canvas* and the programmer of the view would typically overload the *paint* method from *Canvas* and write a simple method for each operation they want their animation to be able to handle. Dyno provides an interface to load view classes, and the Mapping layer provides an interface to connect events to methods of a view class.

The information required by a view can either be delivered directly by the Mapping component, as a "live feed", or that information can be stored in a file for later review. The serialisation API was used for this purpose. Serialisation supports the reading and writing of objects and primitive values over a stream, of which a file is an example.

An object, when serialised, will serialise all of its fields creating an easy way to store and retrieve objects. A downfall of using this approach is that because of the security implications of being able to modify an object during transmission or storage, not all classes are serialisable. Only those that implement the `Serializable` interface may be written to and read from a stream.

Not all classes implement `Serializable`, but the significance of this problem is reduced because of the mapping process. By the time the information gets this far, it has likely been mapped to a format that is useful in visualisation, such as strings, primitive types or container classes, all of which are serialisable. The mapping of non-serialisable classes to the

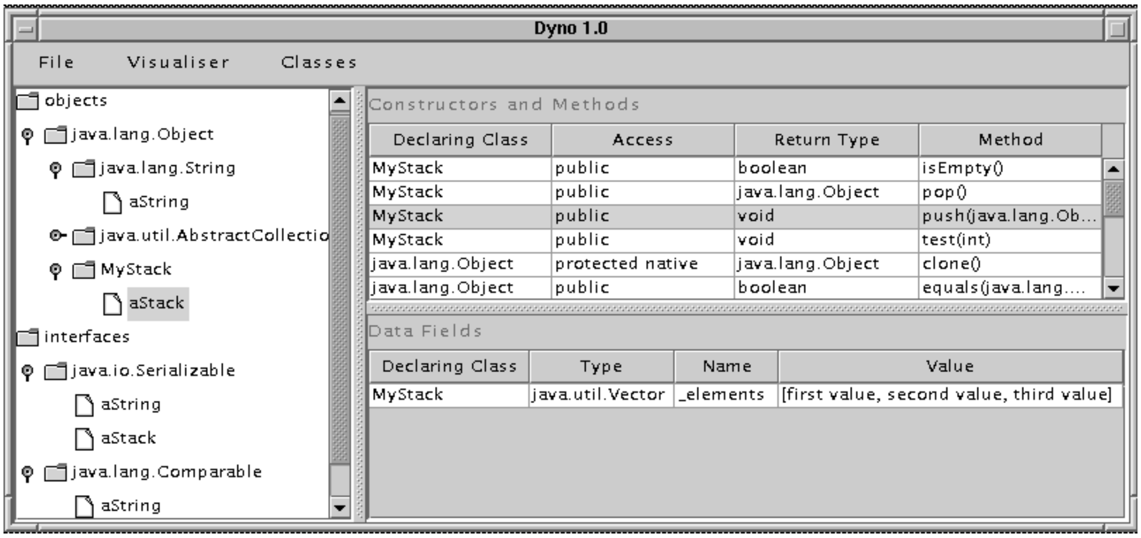

Figure 4: *The main interface of Dyno, showing the available classes and objects in the system. The "aStack" object is currently selected and its methods and fields are shown in the tables to the right.*

form that the Visualiser uses is performed directly before serialisation, so the non-serialisable classes can still be use safely as they will be converted before a problem arises.

5 Dyno in action

The main interface for Dyno is shown in figure 4. The classes that have been loaded into Dyno are shown in the hierarchical browser, known as the "object browser", in the panel to the left. Selecting any loaded class causes the constructors, static methods, and static fields for the class to be shown in the panel on the right. The methods can then be called, or the fields changed. If a constructor is called, then the resulting object will be named by the user and placed within its class in the browser. Figure 4 has two objects visible: `aString` and `aStack`. An instance of `java.util.Vector` also exists, but is not visible because the `java.util.AbstractCollection` node in the object browser has not been expanded.

The loaded classes and created objects are organised by the inheritance hierarchy used by Java so that when an instance of a certain type (such as an `AbstractCollection`, which `Vector` inherits from via `AbstractList`) is needed, the user can easily see which instances already exist that satisfy the type requirements. Any relevant interfaces also get shown in the object browser, and any object that belongs to a class that implements those interfaces will be listed under the appropriate interface, as well as under the class. Arrays are organised separately because they are treated specially in Java.

If an object is selected in the object browser, the methods and fields for that object are displayed in the panel to the right. Methods and fields for any superclasses are also shown. As with the static methods and fields, methods can be called and field values changed. Figure 4 shows the result of selecting the `aStack` object. When a method is invoked on an object, Dyno will then prompt the user for any parameters required. Figure 5 shows the `push` method of `aStack` being invoked. The user can select objects from the object browser, in this case `aString`, to be passed to the method invocation.

Once any required parameters have been provided, execution of the method proceeds. As it does so, events detected by the program component are passed to any mapping components that have registered an interest (see below). The events are filtered by the mapping

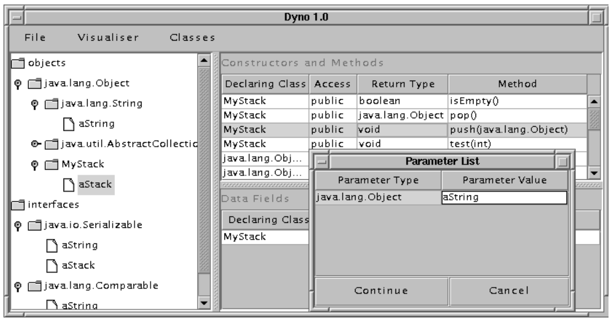

Figure 5: *When a method that requires is invoked, the user supplies values from the object browser.*

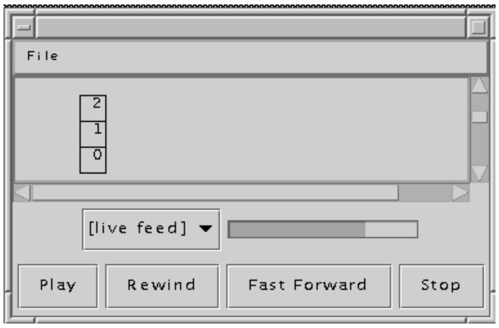

Figure 6: *A view of a stack, about 80% of the way through its animation. The control panel allows for moving in both directions through the animation and gives the user some idea of the current position in the sequence.*

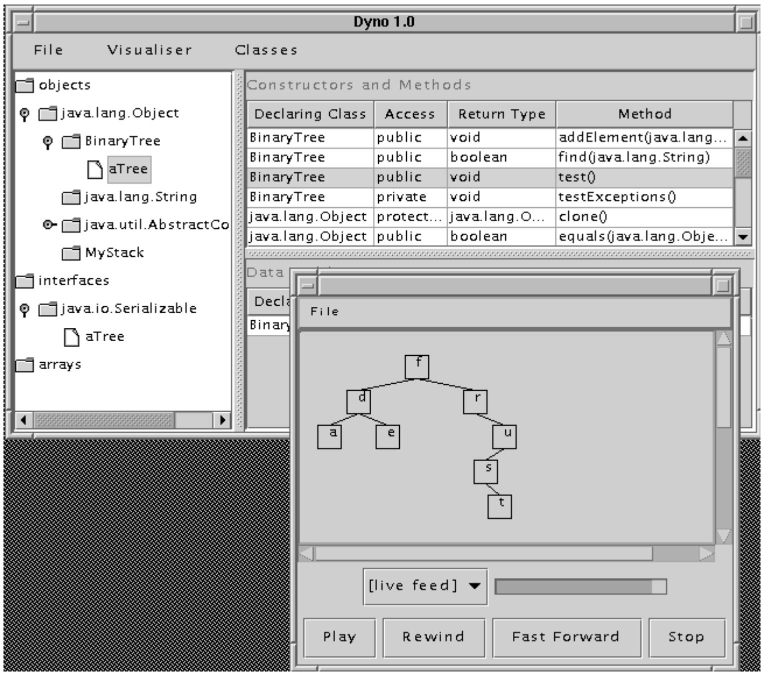

Figure 7: *A view of a binary tree, towards the end of the animation.*

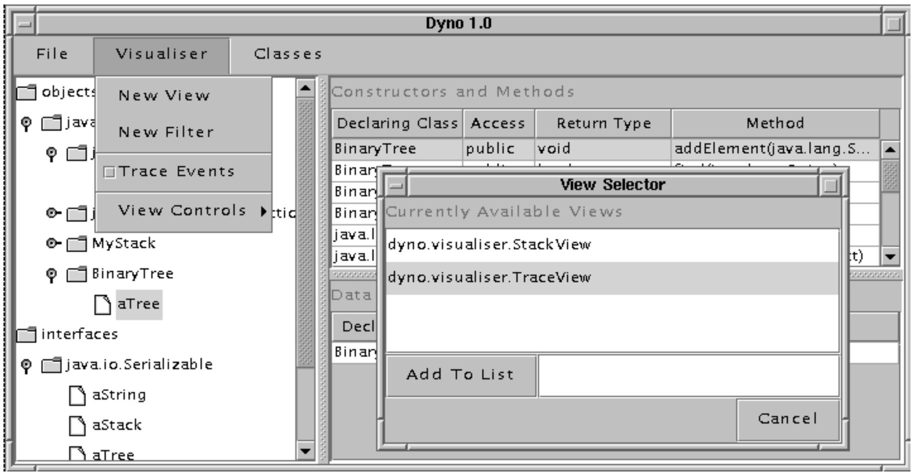

Figure 8: *Choosing a view.*

component to be in a form usable by views in the Visualisation component. Figure 6 shows a simple view for a Stack in action. The common user interface for all views is based on a tape deck, with facilities to *play*, *rewind*, *fast-forward* and *stop* the current animation. As mentioned above, a view may get the events via a live feed, or from file, and so the user interface provides a mechanism to specify this.

Figure 7 gives another example of a Dyno visualisation. In this case, it shows a simple method being called on a binary tree. This example demonstrates that Dyno can handle recursive structures.

Connecting a view to a component requires specifying the mapping between the view implementation and the component. The first step is to specify the view implementation to be used. Figure 8 shows the two views that are currently available (the user may load more views on demand). All views specify operations and attributes that must be mapped. The window on the right of figure 9 shows that the `TraceView` view chosen in figure 8 has one operation `new_event` and one attribute `events`. Mapping an operation involves specifying which methods in which component result in that operation being executed. The top window on the left of figure 9 specifies one of the objects in the object browser, `aTree`, while the bottom window on the left indicates that all methods of `aTree` will cause `new_event` to be executed. The user can also specify that only some methods have this effect. The user may specify that the operation gets called on method execution, method return, or both, as is shown in the figure. Attributes are mapped similarly.

Figure 10 shows the `TraceView` that results from the mapping shown in the figure 9. This view traces all calls and returns from methods in the chosen component (`aTree` in this case) and shows them in a UML-like sequence diagram. This is an example of a view that could be attached to any component.

6 Discussion

As shown in the previous section, Dyno meets the basic aim of allowing a user to test-drive a class, and instances of a class. It provides a complete test-harness, giving the user control over what methods get called. Any objects resulting from any method call are placed in an object store, which makes it convenient for the user to reuse them (for example as parameters) in further exploration of the class. Any exceptions generated by a method call are trapped and the user is notified.

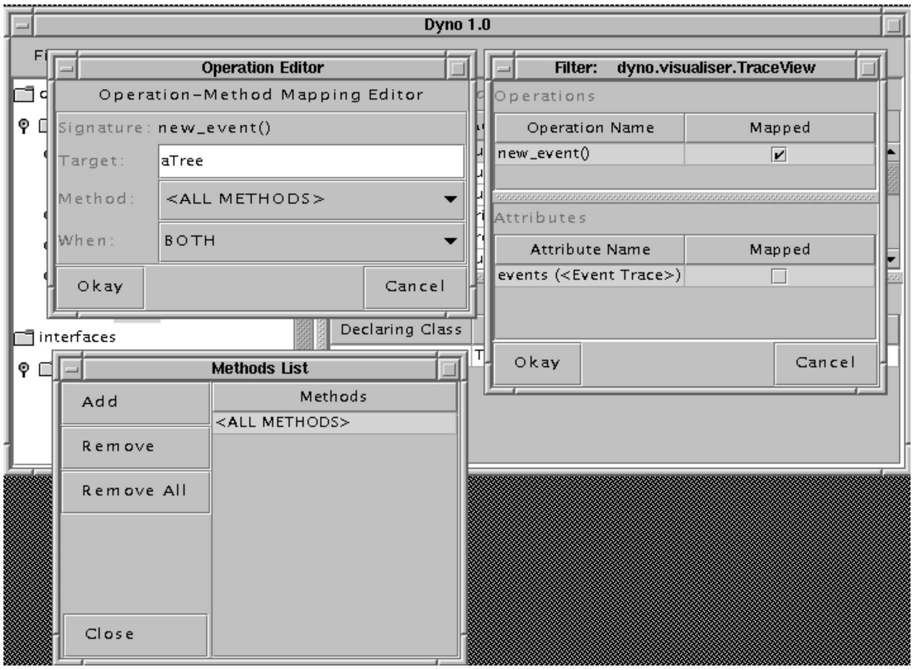

Figure 9: *The interface for specifying the mappings between methods and operations, fields and attributes.*

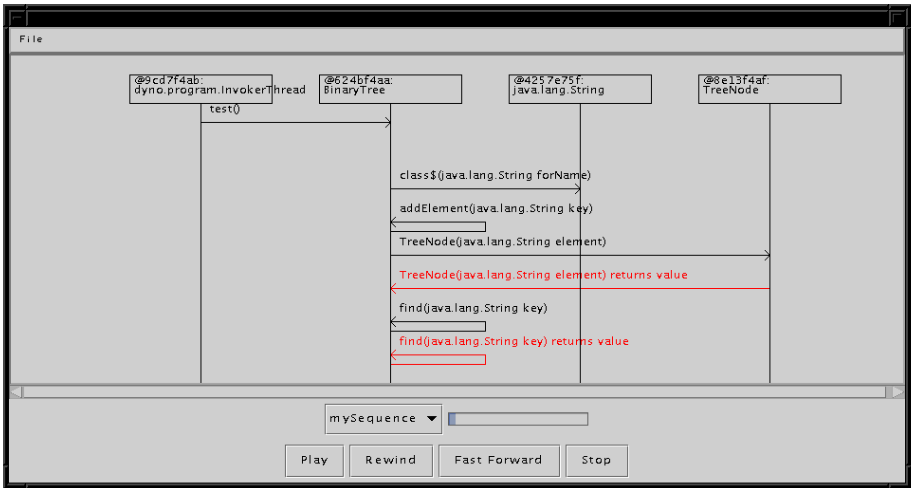

Figure 10: *A "trace view", which shows all method calls and returns using a UML-like sequence diagram.*

Dyno allows the user to examine any part of a class or its instances, that is, it can break encapsulation. This allows the user to put an object in an invalid state, but only when the object is in the test harness. Outside of the harness, the encapsulation boundaries are maintained, since there has been no actual change to the class.

The fact that the class being examined is unaffected by Dyno is due to the availability of the JVMDI and the Reflection API. The Reflection API provides access to the static description of the class, and the available methods and fields. The JVMDI gives access to the actual run-time behaviour of the class.

Because the class that is test-driven is identical to the class that will be (re)used, the user can be confident that the behaviour shown by Dyno is valid behaviour for the component. This will have important implications for support for class libraries. A user of the library can choose a class and examine it with Dyno, and then use exactly the same class in their application.

Dyno allows visualisations to be attached to classes at any point during a test-drive of a component. The views have to be implemented, but the implementation does not have to be directly tied to a specific component. We believe that simple visualisations can be fairly quickly developed. So far, we have only developed views for fairly simple components. There is still a question as to how effective this approach will be for more complex components.

We also believe that some visualisations will be appropriate for many components, for example the trace view can be attached to any component. There is a tradeoff here between the usefulness of visualisations, and their reusability. We expect custom visualisations to be very useful, and we expect general visualisations to be very reusable. The question is whether reusable visualisations will provide sufficiently effective assistance to the user.

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.

To consider how useful a tool such as Dyno might likely be, it seems reasonable to explore how Dyno relates to other tools. However, it is not immediately clear which category of tools is relevant. Dyno is intended to facilitate a prospective user test-driving a component. There are many tools designed to assist software component testing, and test-suite management. However, test-driving is not testing: testing is an effort to determine whether the component meets a specification, but test-driving is simply trial use of a component.

Dyno does feature program visualisation to assist understanding, and there has been much work on tools, such as that of Brown's Balsa [4] or Stasko's Tango [10], that use visualisation to assist understanding of algorithms. Such approaches typically require the programmer to directly annotate the source code with calls that enable visualisation. Dyno, however, does not require access to the source code. Instead, Dyno adopts the approach of Tarraingím[8], and uses object structure itself, and the patterns of method calls and returns, to drive the visualisation. In this way, visualisations can be attached to components without any special annotation or access being necessary.

There is also work on visual debuggers, such as Lens [7], but this is typically aimed at helping the developers of the software, whereas Dyno is aimed at later users of the software. Moreover, such approaches to visualisation typically require access to component source code or special run-time environments, and Dyno is intended to work with library components where these would be serious drawbacks.

The approach that Dyno takes is well suited to its task: to help users of a component better understand how to use the component. This role is in fact similar to that of a documentation system. For example, JavaDoc [1] 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. There are drawbacks. The first is that Dyno relies heavily on the Java run-time environment. Given the current popularity of Java as an implementation language, this may not seem a significant problem, but more importantly it illustrates the benefits that can come from a run-time environment with features such as the Reflection API, and the JVMDI.

The second drawback involved in the approach is the cost of development of the visualisations. In the simplest and cheapest case, visualisations can be excluded, and Dyno would still allow the test-drive capability. While we anticipate this will be useful, we expect the use of visualisations to make the dynamic documentation more effective. In our experience, simple helpful view classes, such as shown in section 5, can be written in 20–50 lines of Java code.

More complex visualisations will require greater investment, and we are aware that complex visualisations are also difficult to design, as well as implement. However, we do not intend Dyno as a general visualisation system. Instead, we hope that Dyno facilitates creation and use of visualisations that are reasonably straightforward but still useful. We speculate that such visualisations might achieve much of the benefit of more complicated visualisations, without requiring difficult design and implementation.

7 Conclusions

In this paper we have described Dyno, a tool that allows its users to test-drive software components, and also allows visualisations to be attached to component behaviour in order to greater assist user understanding. We outlined how our prototype implementation of Dyno facilitates test-driving Java components, and how it does so by making use of the Java Reflection API and the Java Virtual Machine Debugger Interface.

Our motivation in creating Dyno was to explore tool support for reuse, and in particular to attempt to support programming *with* reuse by making it easier for programmers to understand a library component they are considering. In this way, the role of Dyno is similar to that of documentation, and Dyno allows a form of dynamic interactive documentation.

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

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

[2] Robert Biddle, Stuart Marshall, John Miller-Williams, and Ewan Tempero. Reuse of debuggers for visualization of reuse. In *Proceedings of the Symposium on Software Reusability*, 1999.

[3] 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.

[4] M. H. Brown. *Algorithm Animation.* MIT Press, 1988.

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

[6] 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.

[7] Sougata Mukherjea and John T Stasko. Toward visual debugging: Integrating algorithm animation capabilities within a source level debugger. *ACM Transactions on Computer-Human Interaction*, 1(3):215–244, September 1994.

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

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

[10] John T. Stasko. Tango: A framework and system for algorithm animation. *IEEE Computer*, 23(9):27–39, September 1990.