

Using and Visualizing Reusable Code

Position Paper for Software Visualization Workshop

Stuart Marshall

School of Mathematical and Computing Sciences
Victoria University of Wellington
Wellington, New Zealand

stuart.marshall@vuw.ac.nz

ABSTRACT

This paper describes a software visualization tool for helping a developer reuse existing Java code. The tool supports the creation and viewing of visual documentation of reusable code based on a developer's experiences of using that code. The visual documentation, in essence software visualisations, can be used by the developer to better understand what the code does, and how it does it. We have sought to create a tool that can create customizable software visualizations of Java code with minimal modifications to the code itself. This paper looks at both our first prototype, a stand alone Java application called Dyno, as well as at our second prototype called Vare. Vare expands on Dyno by working over a network and also acting as a code repository. We discuss the issues that have arisen so far in our development of these prototypes.

Keywords

Software Visualization, Test Driving, Code Repositories, Java

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 to explicitly support the reuse process, 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.

In this paper we introduce our first prototype tool, called Dyno [2]. Dyno allows developers to "test-drive" a Java component, and can be used to explore the behaviour of a reusable Java component interactively, aided by dynamic

visualization. This will provide the developer with a deeper understanding of what the component does, and how it does it. This in turn helps the developer decide if and how the component can be reused.

We also discuss the current development of our second prototype called Vare. Vare will include a code repository and provide software visualizations over a network using standard browsers and publicly available plug-ins.

2. SOFTWARE VISUALIZATIONS IN DYN0

Our tool uses software visualizations to create dynamic documentation of executing code, so as to help a developer understand what a component does and how it does it. The visualizations may be created by a developer wishing to understand how a component works, or by a component writer who wishes to include it as documentation to help future developers.

In many cases of code reuse, only the compiled code is available and the developer can not inspect the source. To support this scenario, a goal of our tool is to get all the necessary runtime information without having to modify, or instrument, the source code.

Another feature of code reuse is that the reusable code is not typically an entire application in of itself. Unlike other software visualization tools therefore, our tool works on components that make up fragments of applications. In the case of Dyno, the components are individual — or groups of — Java classes.

The software visualizations are customizable and created from information gathered at runtime from executing code. This information includes such things as:

- what methods were called when, on who, by who, and with what arguments,
- what methods returned when, and with what return values,
- what fields were accessed, and when,
- what fields were modified, when, what they were modified to, from, and by who,
- what exceptions were thrown/caught and when.

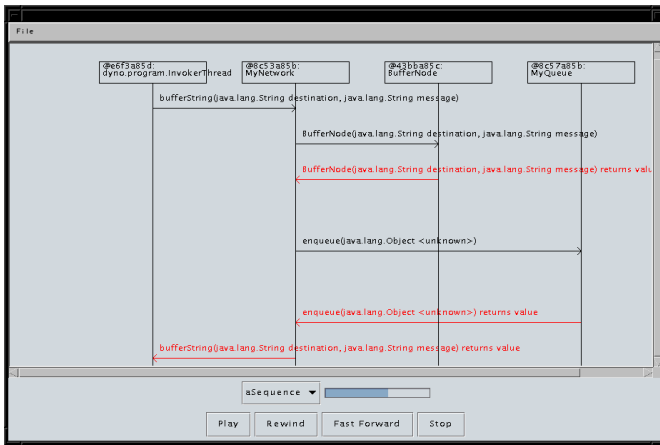


Figure 1: A simple sequence diagram visualization. This sequence diagram is for a component called *MyNetwork*, that uses two other classes called *BufferNode* and *MyQueue*.

Our tool is not limited to any particular set of pre-determined visualizations and can visualize anything from UML diagrams to animated data structures. Each type of software visualization (such as a UML sequence diagram, see figure 1, or a Binary Tree diagram, see figure 2) requires a visualization template that the tool uses to determine what to draw, and when to draw it. The templates then use the information gathered at runtime to flesh out concrete visualizations.

In Dyno, visualization templates are written in Java, and anyone proficient in Java can write their own template and use it, making a wide variety of custom visualizations possible. Java component writers may even write visualization templates that work specifically with their component, rather than use general purpose templates such as UML diagrams. A network component writer may create a template to show how the backlog queues work for their new socket class, a template which will not work if it is used on any other component.

We have discussed the characteristics of our tool’s software visualizations, and also looked at abstract concepts of what appears in our visualizations — e.g. classes, method calls and data structures. We have not yet discussed how our visualizations are populated with concrete information — e.g. what classes, which method calls, which data structures? We now discuss the driving force behind the creation of our software visualizations, an activity we call *test driving*, and then look in more depth at how software visualizations are created from test driving.

3. TEST DRIVING IN DYNO

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 expecta-

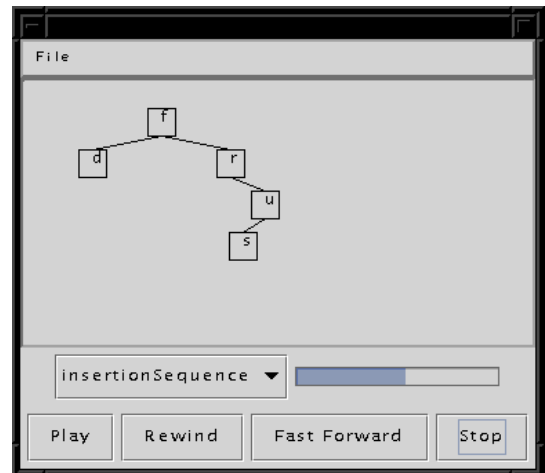


Figure 2: A simple binary tree visualization. Colour, flowing motion and sound are not used here, although they can be incorporated into visualizations. The viewer works on a tape deck metaphor. Playing the visualization shows the tree grow as new nodes are added.

tions. 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. We use the term test driving to describe the process of using a test harness.

Manual test harness are also a potential source of error and confusion. An incorrectly coded manual test harness may give a false impression of what the component does, and errors in the manual test harness may be falsely attributed to the component.

As our goal is to make code reuse more appealing, our tool allows a developer to test drive a component without having to manually create a test harness. In Dyno, the developer can load Java classes, create and store objects, invoke methods, pass parameters, store returned objects, and access and modify fields. This is done through the user interface (see figure 3), and does not requires any coding on the part of the developer.

Tool support for test harnesses also removes the potential source of error from incorrect manual test harnesses, and may make the process of using test harnesses easier, quicker and safer.

4. CREATING SOFTWARE VISUALIZATIONS FROM TEST DRIVING

Software visualizations are created from events that occur as code executes during a test drive. Two different types of users may decide to use Dyno to create software visualizations from test drives. The first group are developers who wish to understand what a Java component does. The second group are Java component writers who wish to cre-

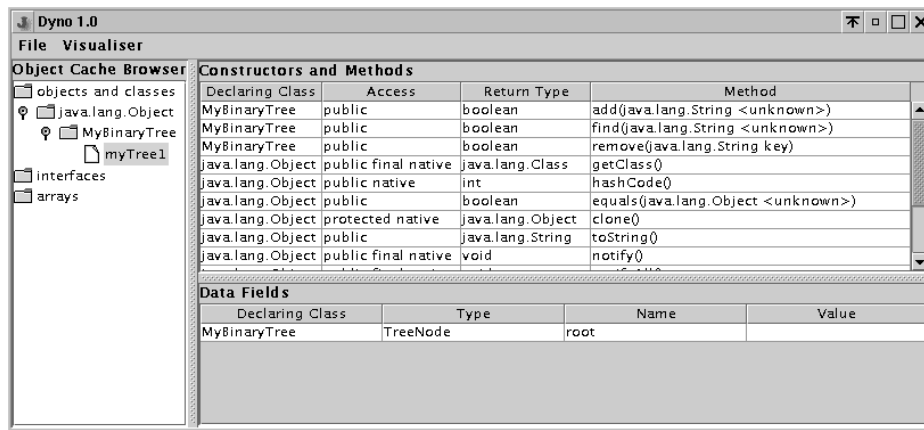


Figure 3: The test driver interface, with an object called “myTree1” of type “MyBinaryTree” currently selected. The left hand side of the interface shows what objects and classes are currently available. The right hand side has two tables, showing the available methods and fields that can be executed on the selected object or class.

ate visual documentation of their own components, so as to reduce the need for future developers to create it themselves.

The documentation resulting from the visualizations is different in nature for the two different types of users. The first group do not fully understand the component, otherwise they would not need to use our tool. This means that their test drives present how they *think* a component might be used, which might differ from how it should *actually* be used. The resulting documentation from the second group is therefore a description of what they have done, and may give them insight into whether what they did was correct. For the second group, the test drives represent how the component *should* be used, as the component writers should be expert in the use of their own code. The resulting documentation therefore fits more in line with traditional forms of documentation, that describe the correct usage of a component.

Regardless of whether the test driving is done by the developer wishing to reuse the component, or by the Java component writer, the steps taken are the same even if the end result may differ in interpretation.

We now look at the technologies used for creating software visualizations, and how components are mapped to them.

4.1 Technologies for Creating Software Visualizations

The software visualizations can be created by using debugger technologies such as the Java Virtual Machine Debugger Interface (JVMDI [3]) to “spy” on the executing code, and detect when certain events occur, such as method invocations, method returns, field accesses, field modifications, thrown exceptions and caught exceptions. The debugger technologies can then be used to derive information about these events, that can then be used to map to a frame or sequence of frames in the visualization.

For example, a developer might be test driving a Java class that represents a binary tree. A call to the “addToTree” method, with a String parameter, could trigger the creation of a new sequence in a visualization where a new node containing the String appears on a visual representation of the binary tree. The entire visualization might contain many such sequences, as well as sequences triggered by calls to the “deleteFromTree” method, or “findInTree” method.

Typically, the visualization templates describe the type of information they display in a manner that is available to users of the template. For example, class diagrams state that they are interested in classes that exist at any given event, whereas sequence diagrams state they are interested in method calls, who called the method, on whom, and with what parameters. It is then a question of filtering the information gained from the debugger technologies to locate only that information that is needed for the visualization. To determine the correct filtering, some input from the user is required.

4.2 Mapping Components to Software Visualizations

While we tried to minimise the amount of mapping information that the user needed to supply for Dyno to successfully map events in the code to visualization sequences, there was the need for at least some mapping information. This is due to the fact that the component writer and the visualization template writer are not necessarily the same person, and each writer has total control over the nomenclature used in their code. Using the earlier example, a method for adding into a binary tree could be called “addToTree”, “add”, “addNewNode” or indeed anything the binary tree component writer feels like calling it. Likewise a sequence in a template for showing a node being added to a tree could be called “addNewNode”, “createNewNode” or, again, anything the visualization template writer feels like. Also, it might be the case, such as in a UML sequence diagram, where an event *type*, such as all method invocations, maps

to a particular sequence, rather than a particular method mapping to it.

It is bordering on the impossible for a tool to be able to automatically create mappings from one arbitrary name to another arbitrary name, so it is necessary for the developer who is trying to understand the binary tree component to say which method in the component maps to which sequence. Note that this can be a one-to-one mapping, or a many-to-one mapping.

This sometimes creates a problem as the purpose of the test driving is to understand the component, and a developer may not know enough to know which methods *should* map to which sequence.

4.3 Technology Problems

One problem that has arisen in our choice of technologies is that return values are not accessible using the JVMDI without modifying the source code to include flags that the debugger can detect. This is a problem since reusable code may only be available in compiled form. We have created a work-around that involves parsing the class file when a Java class is loaded into Dyno, and automatically inserting the necessary calls. This is not an optimal solution, and the developer must use the unmodified class when actually reusing it in their new context.

5. VARE: THE NEXT PHASE

Dyno is our first prototype of a tool that helps a developer reuse code through combining test driving with software visualization. We have begun usability testing [4] on Dyno to determine what usability problems the tool has, and we are now at the stage of developing a new prototype, Vare, that goes beyond the specifications of the first.

Whereas Dyno is a stand-alone Java application for visualization of Java code, Vare will be accessible over a network. It will combine the functionality of test driving and software visualization with a code repository. Developers will interact with Vare by using standard web browsers to access a web site (either on the LAN, or the Internet) where they will be able to import Java classes, browse through classes already loaded, test drive them as before, and create software visualizations that can then be streamed over the network. Vare will also provide support for attaching traditional forms of documentation (such as JavaDoc [1]) with classes, so that the software visualizations complement rather than replace current methods of understanding components.

5.1 New Technologies

Vare will use several technologies that were not used in Dyno. These include:

- SOAP to handle control communication between web browsers and the test driver engine on the server.
- XML to store the information gathered from debugger technologies regarding the events in the executing code.

- SVG to specify and display the visualizations. Dyno stored the visualizations as Java objects, and visualizations were therefore limited to being shown on viewer applications that understand the structure we had enforced in a visualization file. Viewers for SVG are widely available, and is becoming increasingly widely used for presenting visualizations over the Internet.

5.2 Interface Metaphor

Vare will also present an entirely different kind of interface to users, to combat the weaknesses that our usability testing of Dyno have highlighted. We are looking at using a "studio film production" metaphor for the creation of the visualizations, to further help casual users quickly understand the order in which things need to be done to create software visualizations. In this metaphor, loading and browsing classes is similar to selecting your characters, test driving is similar to writing a script, and creating the software visualization is similar to directing the script.

6. SUMMARY

We are developing prototypes for a tool that will help developers better understand Java components so as to know if and how the components can be reused. Developers will be provided with visual documentation of the component being used (test driven) either by themselves or the component's author. This visual documentation will be in the form of dynamic customizable software visualizations, that (ideally) do not require any modification to source code. We are seeking to minimize the amount of information needed from the developer to successfully map a component to a particular software visualization.

We are using debugger technologies such as the JVMDI to get information from executing code needed to be able to populate our visualizations.

We have already developed our first prototype, called Dyno, and are now developing a new prototype, Vare, that will include a code repository and work over a network.

7. REFERENCES

- [1] Javadoc - Java API Documentation Generator. The Javadoc Homepage
<http://java.sun.com/products/jdk/javadoc/index.html>.
- [2] S. Marshall. Understanding code for reuse. Master's thesis, School of Mathematical and Computing Sciences, Victoria University of Wellington, New Zealand, 1999.
- [3] S. Microsystems. Jvmdi - Java Virtual Machine Debugger Interface. The JVMDI Homepage
<http://java.sun.com/products/jdk/jvmdi/index.html>.
- [4] J. Nielsen. *Usability Engineering*. AP Professional, 1993.