VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui

School of Mathematical and Computing Sciences
Computer Science

How (Not) To Help People Test Drive Code

Stuart Marshall, Robert Biddle, and Ewan Tempero

Technical Report CS-TR-01/5
August 2001

School of Mathematical and Computing Sciences
Victoria University
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

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

How (Not) To Help People Test Drive Code

Stuart Marshall, Robert Biddle, and Ewan Tempero

Technical Report CS-TR-01/5
August 2001

Abstract

This paper discusses the results of usability testing on the interface of Dyno. Dyno is a tool designed to support code reuse by helping software developers understand what a specified piece of code does. The tool does this by allowing a programmer to undertake a process we call *test driving*. This paper looks at the deficiencies uncovered in the initial interface, and the implications this has for a tool aimed at helping software developers better understand code fragments so as to be able to reuse them.

Author Information

The first author is a PhD candidate, and the other authors are lecturers, all in Computer Science at Victoria University of Wellington, New Zealand.

How (Not) To Help People Test Drive Code

Stuart Marshall **Robert Biddle** **Ewan Tempero**

School of Mathematical and Computing Sciences
Victoria University of Wellington
PO Box 600, Wellington, New Zealand
Email: stuart.marshall@vuw.ac.nz,
robert.biddle@vuw.ac.nz,
ewan.tempero@vuw.ac.nz

Abstract

This paper discusses the results of usability testing on the interface of Dyno. Dyno is a tool designed to support code reuse by helping software developers understand what a specified piece of code does. The tool does this by allowing a programmer to undertake a process we call *test driving*. This paper looks at the deficiencies uncovered in the initial interface, and the implications this has for a tool aimed at helping software developers better understand code fragments so as to be able to reuse them.

Keywords: Usability testing, code reuse, test driving, tool support.

1 Introduction

Our research looks at the questions of if and how code reuse can reduce the amount of time and effort software developers spend on new projects. We are concentrating on one of the problems often encountered by developers trying to reuse code: understanding what the code does, if it can be reused, and if so then how.

We have built a prototype of a tool to help a developer better understand reusable code. The aim of the prototype is that a developer using it should be able to better understand code through being able to *test drive* the code. We use the term test driving to describe the process of creating and using a *test harness* – or test code – to see what target code does and how it can be used. Manual creation and use of test harnesses is already a common activity amongst software developers. Our prototype seeks to make the use of test harnesses – and thereby the process of test driving – easier and more reliable.

Because such tools fill a similar role with respect to code reuse that CASE tools do with respect to analysis and design, and as usability is an important factor in the success of CASE tools [Iivari, 1996], we feel it appropriate to perform usability testing on our prototype. This paper discusses the process and results of our usability testing.

2 Test Driving Test Harnesses

You walk into a second-hand car dealership. You want a cheap, reliable car, and know roughly what you want in a car, but you're not sure how the different cars in the lot match up with what you want. So you take some of the cars for a test drive. The same principles are at work in searching for candidate code for reuse amongst a collection of existing code. You don't want to know if the code is correct with regards to its own specifications, just as when you test drive a car from a dealership you're not test driving it on the motorway to check if the brake pads still work. This means that test driving is not the same as testing. Instead, you are checking if the code's specifications match your specifications for what you need in the new project. We call a tool that supports this a *test driver*.

The concept of playing with existing code to see whether what it does is what is needed is not new. Many developers currently write code that does this for them, such as invoking library methods, passing in different parameters, and then printing the results. We call these test harnesses. Writing test harnesses can be laborious however, and a potential source of error and subsequent misunderstanding. We see test drivers as a means of supplying an easy to use and reliable tool for the creation and use of test harnesses, thereby replacing the current time-consuming and error-prone means of test harness creation.

3 Dyno

Dyno is a prototype showing how we initially thought a test driver might look and feel. Dyno was developed using Java, and can be used on Java classes [Marshall et al., 1999]. With Dyno a user can:

- load Java classes.
- create instances of that class (given the presence of constructors).
- invoke methods on a class or object.
- access and modify fields.
- create visualisations documenting the user's test driving (this is not discussed here at all, see [Marshall, 1999]).

As can be seen in figure 1 the interface is split into two main components: the *Object Cache Browser* and the *Entity Information Browser*. In the following discussion, we use the term *entity* to refer to a particular class, interface, object or array that the user has loaded, or otherwise created.

3.1 Object Cache Browser

Entities are stored in the *Object Cache*. The user uses the object cache browser to select the entity from the object cache that they wish to inspect. The object cache browser is organised as a tree-like hierarchical structure. The three subtrees at the top level represent the three categories for entities: Objects and Classes, Interfaces, and Arrays.

A class can be loaded into Dyno from a Java class file or from a previously saved object cache read from a file. Instances of a class result from invoking constructors and from method return values. The first subtree is ordered hierarchically, using the Java Inheritance model for the entities, and ensuring that all

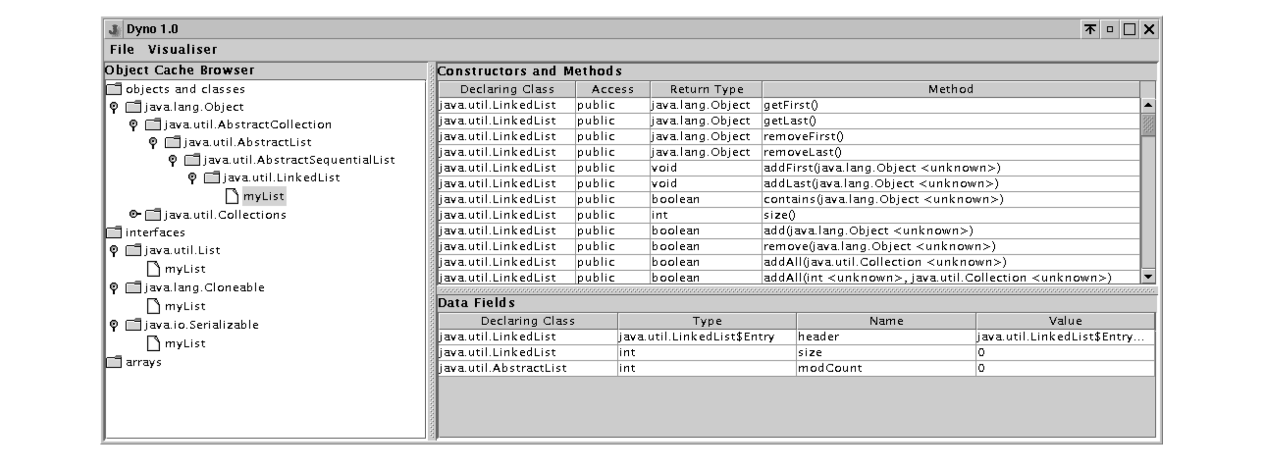

Figure 1: The main interface of Dyno, showing a few entities in the Object Cache Browser, and the methods and fields of a LinkedList class in the Entity Information Browser. Note that the "myList" object entity (a child of the LinkedList class entity) is highlighted in the Object Cache Browser. Also note that LinkedList is a child of java.util.AbstractSequentialList as LinkedList inherits directly from there.

instances of an class are shown as child nodes of the node representing that class.

A single entity can exist in more than one of the three subtrees. This occurs if an entity is an instance of an class, in which case it will appear in the *objects and classes* subtree, and also conforms to a particular Java interface, in which case it will also appear in the *interfaces* subtree.

3.2 Entity Information Browser

The right hand side of the main screen contains the *Entity Information Browser* and shows specific information for the entity currently selected in the object cache browser. The entity information browser is split in half horizontally. The top half contains a table that lists any methods or constructors that can be performed. The bottom half contains a table listing the fields. If the entity selected is a class, then only the static methods and static fields will be shown. If the entity selected is an instance, then only the non-static methods and non-static fields will be shown.

Clicking on the methods in the table causes them to be invoked (asking for any parameters if needed, see figure 2), while clicking on the fields brings up a window that shows the data field in more depth (should it be of a non-primitive type).

4 Usability Testing

We have performed cognitive walkthroughs [Wharton et al., 1994] of Dyno on ourselves [Marshall, 1999]. However we now need to see whether other developers can also use Dyno for the purposes intended, and we are doing usability testing for this reason. We now discuss the people involved, and the goal, methodology and results of this usability testing.

In this initial study, we did not expect to uncover *all* of the usability problems in Dyno. It is expected that there will be undiscovered problems with regards the scalability of the tool, however it seems reasonable to assume any problems discovered here will only worsen as examples scale up, so addressing these early errors will be to the advantage of later prototypes.

4.1 Intended Users and Test Subjects

The intended users of Dyno are Java developers of all experience levels, who will use the tool on a casual basis and will therefore not want to spend a great deal of time learning how to use the tool. This makes learnability a key ingredient of the usability of the tool [Nielsen, 1993].

We have started usability testing on Dyno using students at Victoria University as test subjects. All the test subjects used have had some experience in writing Java applications — with some having only used Java in undergraduate course work, while others are using Java in larger graduate projects.

4.2 Goal

Our goal is to find usability problems in Dyno's user interface. We will get an insight into how the users' interaction with the tool matches how we thought users would interact. The tests will show what users expect to see, to be able to do, and where they expect to be able to do it. Dyno can not be useful if it is not usable. It is not our aim to prove the usefulness of the tool, rather the areas in which it can be made more usable.

4.3 Method

Usability testing was performed under the guidelines of the Victoria University Human Ethics Committee, with regards to advising test subjects of the purpose and nature of the experiment.

The usability testing method we used was to observe test subjects step through a list of tasks that we asked them to do. The test subjects were told that they were welcome to "think aloud" at all times. We coupled this with a basic questionnaire (both pre- and post- test session) to ask them to identify the benefits and weaknesses of Dyno's interface. These techniques are typically seen as key components in the process of discount usability engineering [Nielsen, 1993].

The test subjects were sat down in front of a standard PC running NetBSD — an environment they are familiar with from course work — with the Dyno application already running. The test subjects had previously been given a document describing *what* Dyno is for, but that contained no information on *how* to use Dyno. This was done deliberately so as to be able to gain some feeling for how intuitive interaction with Dyno would be for the test subjects. We then observed the test subjects, making written notes on the way in which they tried to interact with the tool, and on any verbal comments they made on what they expected to happen and why they expected it.

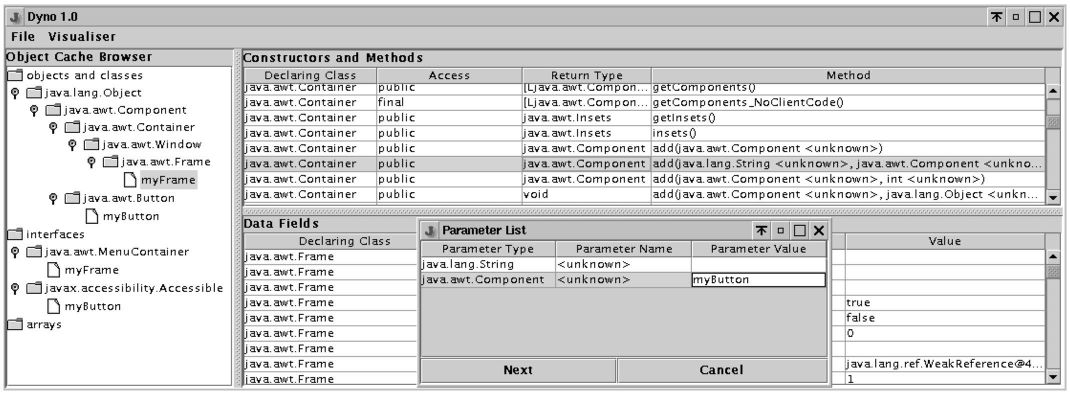

Figure 2: This figure shows a method call being invoked on the java.awt.Frame class. The method, highlighted in the entity information browser, is the *add* method, and takes two arguments. Note that there is little information present on what the arguments represent, although there has been a reasonable assumption made that the component argument is the component to add. "myButton", a Button object in the object cache browser has been named as the value to pass through. The user does not know what to do about the String argument however.

The experiment consists of three tasks.

In the first task the test subjects are asked to use two classes from the JDK to solve a collection of problems. The classes are *java.util.LinkedList* and *java.util.Collections*. Some of the problems require the use of just one class, while others require using classes in collaboration. In both cases the test subject needs to derive knowledge of what the classes can do, and how they can be used. The test subjects are observed as to if and how they derive this knowledge from using Dyno.

In the second task the test subjects are asked to use common user interface classes, again from the JDK. The classes used are *java.awt.Frame* and *java.awt.Button*. Method invocations on the user interface classes have side-effects that are external to the Dyno interface, such as the creation, deletion, or modification of other windows on the user's screen. Use of the classes also requires using two classes in collaboration to solve a simple problem.

In the third task the test subjects are asked to use classes from *gnu's* Java regular expression API[GNU, 2001]. They are asked to solve a problem using the regular expression class.

All three tasks require the test subject to be able to find methods, invoke methods, and pass arguments to methods. All three also require the test subject to understand that the results of method calls can be used in subsequent method calls.

4.4 Results

Having performed usability tests we now discuss some of the preliminary results, and the observations we have made.

The test subjects identified four main areas of concern for the interface: knowing that the tool could do; finding the correct order of method calls; determining what should be used as a parameter; and information overload.

4.4.1 Knowing What The Tool Can Do

Test driving tools need to be intuitive and easy to use, so as to cut down on the amount of time and effort needed to work them. If they take too long, or require too much thought, the software developer will win nothing over having rewritten the needed functionality in the first place. The test subjects were divided on the intuitiveness of Dyno's interface.

Firstly, a number of test subjects were initially unsure of what exactly the object cache browser represented, and did not realise that some of the elements were objects, while others were classes — represented by the *file* and *folder* icons respectively.

Secondly, while most test subjects immediately tried to invoke methods by double-clicking the method name in the entity information browser (the correct course of action), few made the connection that you could do the same with the field names to look in more depth at the data structures.

Thirdly, with respect to passing parameter values, none of the test subjects realised early on that you could pass existing objects through by supplying their name, and needed to put quotes around string literals. This approach was used to mimic the style with which you pass parameter values around in ordinary code, however it took trial and error for test subjects to realise this. While all the test subjects worked this out in the end, it was not as intuitive as we had expected.

4.4.2 Which Method First?

A common problem that test subjects faced was that they often didn't know which method had to be invoked first. To get a particular piece of functionality out of a class, it is typically necessary to invoke a certain set of methods in a given order. An example of this is with the java AWT classes, where if you add something to a frame, you must also invoke the "pack" and "show" methods in that order to get anything on screen.

Sometimes method signatures alone are not enough to work out the correct ordering for their execution, especially when there are a lot of them. A suggested alternative is to combine the test driving tool with the traditional forms of documentation (best supplied by the class's author). This would make the test driving tool complement the current means of understanding code, instead of trying to replace those means.

4.4.3 Difficulty With Parameters

The most striking observation that test subjects made was the difficulty that they had in knowing what a parameter was for. Consider that the test subject is presented with a list of method signatures and asked to execute some of them in a particular order to achieve a certain task. The method name is always present in the method signature, and is therefore a textual clue as to its purpose. However parameter names are not present, unless the class was compiled with the debugging switch on. This means that on occasions the test subject is presented with a method that takes two integers and a string, but no real idea what each represents. Figure 2 shows similar confusion over the *add* method in the Frame class, that was to be used in the second section of the experiment.

The test subject can experiment with different combinations of values, but won't necessarily know where to look easily to find the side-effects those different values may have resulted in.

The common observation was that while executing the methods was a useful function of the tool, there also needed to be the traditional text documentation (such as that produced by JavaDoc [Sun, 2001]) along side it to provide some context for the parameters, the methods and the class.

This reinforces the view that a test driving tool such as this should complement, rather than replace, traditional forms of understanding code.

4.4.4 Information Overload

An observation that we were expecting to see, and subsequently saw, was that test subjects often had difficulty navigating through the larger AWT classes that we asked them to use in the second section of the experiment.

Both java.awt.Frame and java.awt.Button have a large number of methods. The current layout of methods has two orderings. Firstly they are ordered by the class that defined them, so all superclass methods are listed after the methods defined specifically for the class under inspection. Likewise, all methods in the immediate superclass appear in the list before those defined in the next superclass up the hierarchy. Secondly, all methods defined in the same class are ordered alphabetically with respect to each other.

This works well in some situations, although it assumes that the user will know a method's name (or likely name) when they are looking for a particular piece of functionality. Given that the class's author has total flexibility in the nomenclature of the methods, and the user is using Dyno because they *don't* understand the class, the user is unlikely to know exactly what name they are looking for.

Alternative orderings that were suggested by the test subjects were to use an ordering suggested by the class's author (perhaps derived from any accompanying JavaDoc files), or have a search engine that allows the user to search by parameter types. The latter approach was suggested on the basis that while they may not know the *name* of the method, they may be able to make some guess as what type of information it would need. Indeed, with the second part of the experiment, where it was necessary to combine a couple of interface components together, the test subjects were uniformly searching for a method that took a java.awt.Component as an argument, instead of explicitly searching for a method called *add*.

5 Summary

We have developed a prototype for a Java code test driver. We have done this with the intention of helping software developers to better understand how a particular piece of Java code works, and thereby empower them to be able to reuse that code if appropriate. Having created this prototype, we have now performed usability testing to find any problems in its user interface.

The usability testing on Dyno has been useful in throwing up a number of usability problems. The ordering of the entities in the object cache browser needs to be made clearer, and there is a lack of context for the entity information, such as what the parameter names and ordering mean. There is no easy way of finding a method in the methods table (and to a lesser extent – a field in the fields table), and there should be multiple possible orderings that can be selected by the user. The interface for passing values through to method calls isn't as intuitive as expected, and there is often too much information on the screen for the user to easily understand.

The users were eventually able to complete the tasks asked of them, however there is room for improvement in the usability of the tool. Since its usefulness is heavily affected by its usability, future prototypes need to take into account these test findings to better promote the reuse of code. Usability testing has been useful for uncovering problems in the tool that had previously gone undetected.

6 Future Work

Dyno is our old idea of how a test driver may look. We are currently developing a web-based code repository that will combine a test driver and a software visualiser to create animations of the test drives.

Dyno needs further usability tests using larger classes (and collections of classes), along with more software developers, some preferably with experience on large projects. This will help find more usability issues as scale becomes more of a problem.

The usability tests performed on Dyno highlight the mistakes to avoid in the new web-browser interface, and also help in identifying where additional information needs to be included. An example of the latter is the suggestion of using JavaDoc to add some context to what the methods do, and the purposes of the individual parameters.

As usable tools are developed to support software developers understand code, so to may the appeal of trying to find code to reuse.

References

[GNU, 2001] GNU (2001). Java regular expression library. "http://www.cacas.org/java/gnu/regexp/".

[Iivari, 1996] Iivari, J. (1996). Why are case tools not used. *Communications of the ACM*, 39(10):94–102.

[Marshall, 1999] Marshall, S. (1999). Understanding code for reuse. Master's thesis, School of Mathematical and Computing Sciences, Victoria University of Wellington, New Zealand.

[Marshall et al., 1999] Marshall, S., Biddle, R., and Tempero, E. (1999). Dyno: A tool for dynamic interactive documentation. In *First Symposium on Constructing Software Engineering Tools (CoSET)*.

[Nielsen, 1993] Nielsen, J. (1993). *Usability Engineering*. AP Professional.

[Sun, 2001] Sun (2001). Javadoc homepage. "http://java.sun.com/j2se/javadoc/index.html".

[Wharton et al., 1994] Wharton, C., Rieman, J., Lewis, C., and Polson, P. (1994). *Usability Inspection Methods*, chapter 5, pages 105–140. John Wiley & Sons Inc.