

A Web User Interface For An Interactive Software Repository

Stuart Marshall, Robert Biddle & James Noble
stuart@mcs.vuw.ac.nz

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

Abstract

Using tools aimed at promoting the reuse of existing components costs the user in the time and effort needed to install and understand the tool. These costs could counteract or subsume the benefits of reuse argued for by reuse practitioners, rendering the activity worthless. One approach to reducing these costs is to deploy the tools in an environment that the user is already familiar with, and has easy access to. We have chosen the web as just such an environment, and this choice can have a significant impact on the usability and utility of the tool. This paper discusses the difficulties that arise from our use of the web, and the manner in which we have partly overcome these difficulties.

Keywords: component reuse, test driving, www.

1 Introduction

We are interested in tool support for developers wishing to reuse components. Specifically, we are interested in tool support that reduces the cost of understanding what a component does, and how it does it. The cost of understanding can be categorised in terms of the time and effort spent by the developer determining whether a particular candidate component can be made to work in the developer's new context. In exploring tool support to reduce this cost, we are mindful of making the tool support inexpensive to use with respect to these same cost categories.

The tools we are developing focus on two main techniques for improving understanding. The first technique is to give developers first hand experience of being able to manipulate instances of a component through its public interface. We call this technique *test driving*. The second technique is to show developers software visualisations of runtime information captured during the test drives. Together, these techniques will allow developers considering components to "try before they buy".

We have developed a visualisation architecture for reuse (*VARE*), and are currently developing a prototype tool called *Spider*. We have discussed the architecture in an earlier paper (Marshall, Jackson, McGavin, Duignan, Biddle & Tempero 2001), and a diagrammatic form of it can be seen in figure 1. *Spider* currently supports Java components (Sun Microsystems 2003b), with each Java component being a set of Java classes.

The goal of the tool is to ultimately make component reuse more attractive by helping reduce the cost of reuse to the point that it is cheaper to reuse than reinvent. The usability of the tool must not have a high associated cost, otherwise the overall cost of reuse will not be reduced. We have tried to enhance the usability of the tool by presenting it in an environment, and over a delivery mechanism, that developers would commonly already have access to and exposure to. Specifically, we have chosen to present the tool inside a W3C standards-compliant web browser, and delivered in a typical client/server model over the web.

The choice of environment and delivery mechanism impacts on what the tool can achieve. In this paper we explore the impact of using the web on how we can now support the developer.

We will first provide some context as to how we envision developers test driving components and viewing visualisations. We then identify the core utility features that would need to be available in the user interface of a tool supporting these activities. We will then demonstrate how our choice of the web as a means of improving the usability of the tool in turn impacts on the utility.

The use of the web presents challenges in dealing with component user interfaces, and in dealing with the event notification architecture that many languages use. While still in development, we use *Spider* as the subject of this paper, and we discuss how *Spider* meets some of the challenges posed by our choice of the web.

2 Motivation

Component-based development is developing into a vibrant activity within the field of software engineering (Ravichandran & Rothenberger 2003). The practice aims to deliver the benefits that reuse has promised (and delivered in some areas with qualified success) for decades.

Proponents argue that by reusing existing solutions (i.e. reusable components) developers win in three ways. Developers end up with software that was less costly to produce; that is of a higher quality; and that may be less costly to maintain. Firstly, the resulting software is less costly to produce as time and effort has not been spent re-designing, re-implementing, and re-testing what has already been done by the reusable components. Secondly, the resulting software is of a higher quality because the reusable components have presumably been tested and used in other contexts. For commonly reused components, any flaws have a high chance of having been discovered and fixed. This can be seen in the philosophy of many eyes making bugs shallow that is

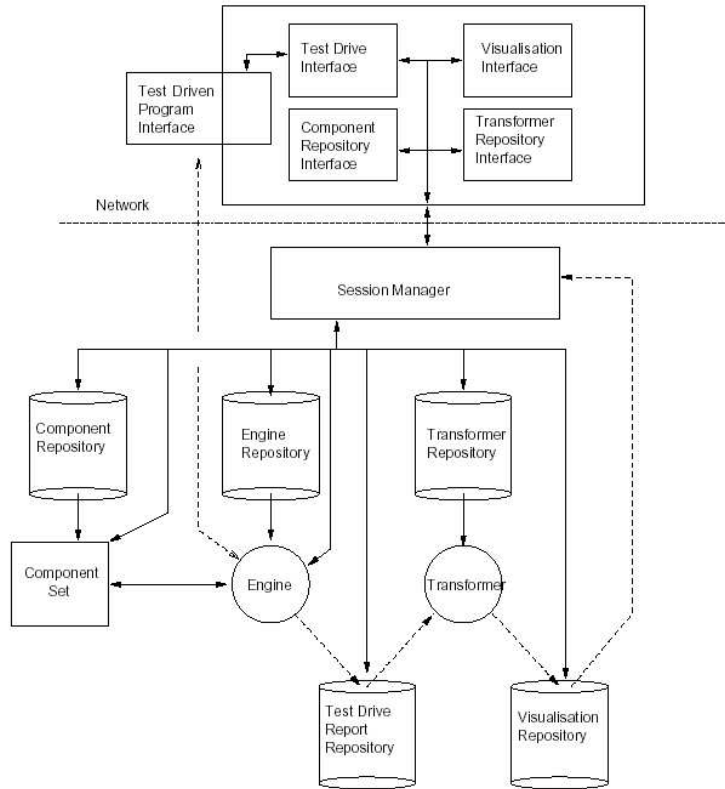


Figure 1: The Vare architecture is based on a client/server model, with the server being split into repositories and processes. Dashed lines represent test drive or visualisation input/output, while solid lines represent control, queries, or responses. The transformers mentioned in the figure refer to programs that can turn execution traces into a visualisation format.

followed by the open source community. Thirdly, enhancements or corrections to a piece of functionality across a range of projects can be done by modifying in one place and propagating the change to the interested parties.

There are counter-arguments to the statement that reuse lowers the cost of software development. One counter-argument is that while the cost of re-designing, re-implementing and re-testing is removed from the cost of software development, the *activity of reuse* is potentially more expensive to perform, thereby *increasing* the cost of software development.

The costs of reuse can be broken down into finding reusable components, understanding reusable components, adapting reusable components, and linking in reusable components. The costs can be measured in time, effort, financial outlay, and legal restriction.

We are specifically interested in reducing the cost of understanding reusable components. We seek to reduce this cost by providing easily available tools that allow developers to actively engage with a component, and also view rich visualisations of the component. For tools to reduce the cost of an activity, they must in turn be cheap to use in those same cost categories.

Many tools that may be used to gain a better understanding of software behaviour are too costly to use. Some tools are not costly to use, but the amount of extra understanding gained is reduced by how the developer is restricted with interacting with the component, or by restrictions on the method of displaying information to the developer.

In the former case of the tool being too costly to use, this can manifest itself in the need to install, configure and maintain a tool on the developer's worksta-

tion; the need to learn how to use the tool; the need to work with the tool when it is external to the developer's environment; and the need to migrate to a new tool should the developer move to an architecture not supported by a previously used tool.

3 Test Driving

We have earlier defined the term *test driving* as manipulating instances of the component through its public interface. The purpose of test driving is to gain first hand experience of using the component. We have previously created a stand alone prototype called Dyno (Marshall, Biddle & Tempero 1999b)(Marshall, Biddle & Tempero 1999a) that can be used for test driving and visualising Java classes. We have also previously performed usability tests on Dyno (Marshall, Biddle & Tempero 2002).

We now discuss in more detail our model for how test driving is done. We then discuss how test driving differs from testing, before discussing how software visualisations tie into our test driving, and then who test drives (and how).

3.1 What Test Driving Is

Test driving is performed by writing instructions to play with a component. An instruction is a call to a public operation, where the instruction is written in the component's language. A list of ordered instructions combined to achieve a particular task is called a *test drive*.

A developer is presented with the public interface of a component. The developer then needs to instantiate part or all of the component, so that there is some-

thing for the developer to manipulate. Depending on the complexity of the component, and the number of parts (e.g. classes) that it has, this may involve one or more invocations of constructor or factory operations.

Once a collection of instances has been created, the developer can begin to invoke their behaviour. Invoking operations may result in values being returned. These values are stored and indexed just as they would be in source code, and can be recalled for use as parameters in future invocations of the current test drive. These values may be stored for the duration of one test drive, or the developer may decide to carry these values across to future test drives as well.

3.2 Test Driving vs Testing

The developer wishes to understand what a component does; how the component can be used; and how the component can fit into the new context.

In the first case, the developer needs to know what the performance characteristics of a component's behaviour is, and what the side-effects are of requesting a particular piece of behaviour (i.e. sending a sequence of messages to an instance of the component). There currently exist many tools for supporting unit testing of what components do (Cheon & Leavens 2002). The objective of these tools is to allow a component's author or maintainer (or someone closely associated with either) to compare the component's observed behaviour with the required correct behaviour. Any differences between the observed and required behaviour can then be eliminated. An assumption required for using these tools is that the developer is intimately familiar with the required behaviour and can correctly identify the differences. Another assumption is that the developer knows how the component should be used, so can easily specify the correct sequence of messages.

A developer test driving a component also has a required behaviour to compare against the component's observed behaviour. This required behaviour is the functionality currently missing from the developer's new project. However, the developer may not have had any prior link with the component being explored, and may have no prior experience of its use. This links in with the second case from the list described at the top of this section: how the component can be used.

3.3 Visualising Test Drives

The cheapest use of a component is that which does not require any change to the component's source code, assuming that the source code is even available to change. Use of the component's behaviour can then be achieved by a sequence of messages to the component's public interface. Extension of the component's behaviour (thereby reusing the component as the base behaviour) could also be achieved through similar mechanisms as to those used in whitebox and blackbox frameworks.

For any moderately complex component, both of these uses of existing behaviour requires at least a basic understanding of the component's public interface, and an appreciation of the importance of various parameters to operations in that public interface.

It is here that the software visualisations created from the developer test driving the component come in useful. The visualisations show the runtime execution of the component, and can give clues as to how the parameters are utilised in the operations invoked by the messages. If a parameter was an object for instance, they can show what messages are passed to

that parameter object. This would then provide a clue as to how sub-typing the parameter could result in the behaviour of the component being extended to perform the required functionality, with the component's "framework" being reused.

3.4 Who Test Drives

There are three audiences for our tool, of which we are currently focusing on two. The first audience is the developer interested in reusing an existing component. Their motivation for using the tool would be they do not as yet have the level of understanding necessary to properly determine if and how the reusable component can be reused in their particular new context. The aim would be for the necessary understanding to be provided by first hand experience of using the component. This understanding would then be reinforced by the subsequent viewing of customisable visualisations created from the test drives' execution traces.

The second audience is the developer of a reusable component, who wants to create visualisations showing how the components can be reused.

A third potential audience outside the scope of this paper is visualisation designers who create customisable mappings from execution traces to visualisations, for use by the first two audiences.

In both of the first two core audiences, we have users who are technically capable, but who also have significant pressures placed upon their time and resources by the need to be continually creating or maintaining new functionality — or new combinations of old functionality. The first audience in particular would be sensitive to any usability costs associated with our tool, and any lack of utility imposed by our delivery method. The second audience may be more forgiving of needing to learn how to use the tool, and any resource constraints imposed, but their win from performing this activity is directly connected to the frequency with which the first audience is willing to use the tool.

4 Test Driving over the Web

Many developers already use the web (or at least an HTML browser) when reusing existing components. One of the most frequent uses is in the reading of online documentation or developer forums associated with a particular technology.

An example from the programming language Java is the HTML documentation files created by the JavaDoc tool (Sun Microsystems 2003a). A standards-compliant HTML browser can be used to navigate through static descriptions of a component's interface, along with certain structural details of its implementation. The descriptions also include comments made by the component's author. These comments can further clarify the purpose of the component, and its usage.

As Java developers are already familiar with using the web as a mechanism for understanding existing Java components, we decided to also use the web as the delivery mechanism for our tool for creating dynamic documentation of a component's execution.

As web browsers are also widely available amongst developers, we decided to implement as much of our prototype using standards that would not require any extra configuration or installation on behalf of the users. This has the effect of moving much of the execution on to a server that the user connects to. This emphasis on server-side work benefits us also by reducing the impact of using the tool on the user's

machine, and in an increased sense of trust that may eventuate as the user is not having to compromise their own security environment to run currently unknown/untrusted components. By shifting as much of the execution as possible to the server, and transmitting only W3C standard formats, we also seek to make the tool effectively platform independent — and reduces the possibility of different results or performance being seen by different users based on their own unique environment.

Another effect of shifting component execution to the server-side is that the component does not need to be downloaded on to the client machine. The benefit of this is that if the component was a commercial product whose use resulted in a financial cost, the component could be offered “on trial” via the repository, and the developer could decide before purchasing it as to whether it was what they needed.

There are two distinct user interfaces that need to be handled by a test driving tool. The first interface is the tool’s interface to the developer, through which actions are specified to select components, and then query and manipulate those selected. The second interface is that which a component may produce to gather more information, or display results.

As well as utilising a commonly available environment, the use of the web as a delivery mechanism enables the association of our prototype with code repositories. Public repositories of reusable components already exist, and the addition of our architecture’s functionality to the repositories’ library functionality could be highly beneficial.

5 Tool Requirements

Providing the utility for test driving and software visualisation places certain requirements on the tool’s user interface. In any implementation of the tool, these requirements must be addressed.

5.1 Interacting with Component Public Interfaces

A test driving tool needs to allow interaction with a component’s user interface in two ways. Firstly the tool needs to provide a view of the component’s public interface for the developer. Secondly the tool needs to provide a way of invoking the operations within that public interface.

5.1.1 Displaying the Public Interface

Test driving a component primarily means providing a sequence of messages to operations listed in the component’s public interface. To facilitate this, the tool’s user interface must display the component’s public interface to the developer. A component is a set of types. Each type could contain tens of public operations. This does not even include the protected operations that it might be useful to understand, as they may allow reuse through subtyping. If a component was to consist of more than a few relatively complex types, the amount of space needed to view the public interface of the component would be greater than that available on the developer’s screen. The tool’s user interface needs to handle components as they scale up, by providing some query feature, or means of filtering specific categories of operations (or types). Those categories could be automatically created by primitive assumptions made based on the nomenclature used in the names of the types and operations.

5.1.2 Invoking Operations

Having been shown the public interface, the developer may well then wish to see what happens when certain parts of it are invoked. The tool’s user interface needs a mechanism for allowing developer’s to specify which operations they wish to invoke, and in which order.

One key task in invoking an operation is passing parameter values. It may be necessary to use results from earlier invocations as parameters to a new invocation. Therefore, the tool needs to allow the storage and display of stored results as the test drive progresses.

5.2 Timeline Navigation

A single test drive of a component is a sequence of message calls to an instance of that component, where it is assumed that an earlier message call may affect the environment that a later message call executes in.

During the use of the tool, a developer may need to run multiple test drives. There should be the ability for the developer to specify whether a test drive starts from the end state left by the previous test drive, or whether it starts in a “clean environment”. Starting from a clean environment may mean performing such tasks as reverting any static type-scope data back to their initial values, or rolling back the contents (or existence) of files or streams. The user interface could even support the ability to re-execute earlier test drives — or parts of test drives — to put a component in a particular state for new exploration.

5.3 Handling Component I/O

The test driving model we have described describes a developer (in the role of the tool’s user) interacting with a component through the component’s public interface. This represents the model of interaction expected when the developer creates software applications using the component. However the component may well have a second interface: the user interface presented to the user of any software application the component ultimately forms part of.

This user interface needs to be displayed during the test driving of the component, and any input and output properly handled. As the software visualisations are documenting the test drives, the software visualisations should ideally have some means of separately identifying any interaction that was occurred with the component’s user interface.

The component may well have more interfaces: designed to be used in collaboration with a filesystem, a network, or with other software applications external to that which the component forms part of. This raises security issues as the behaviour of the component is not entirely known to the developer at the time of test driving. The component may therefore affect the integrity of the developer’s system.

5.4 Handling Component UI Context

A component is not necessarily an executable in its own right. The component’s user interface may only represent a portion of what is meant to be shown in a larger tool. A tool to support test driving of components with user interface “parts” may need to create a mock framework in which the component user interface can be run. The framework should not modify the behaviour of the component’s user interface (or its default appearance) as this would give a potentially misleading account of the component’s behaviour to the developer doing the test driving.

Another requirement stemming from the need to display the component's user interface is the requirement to provide the user interface with a frame it can exist in.

5.5 Handling Component Event Notification

Components may be designed to work in a model/view/controller architecture, akin to the Observer pattern (Gamma, Helm, Johnson & Vlissides 1994). Side-effects of components may therefore not be visible by analysing values returned by messages sent to the public interface, or through a supplied user interface.

Instead, behaviour may be discovered by registering listeners for key events that the component advertises. For a component that advertises more than a few events, the manual creation and registration of listeners for these events would be burdensome on the developer. The tool needs to register default implementations of listeners that direct feedback from any component events for displaying in the tools interface. The information gathered from these events must also constitute part of the runtime information later visualised.

6 The Spider Prototype

We now briefly describe our current prototype. The prototype is written in Java and supports the test driving of Java classes and packages over the web. Our earlier prototype, Dyno, was a stand-alone application requiring that the Java classes be downloaded and executed on the developer's machine. A representative screenshot of Dyno can be seen in figure 2. A screenshot of the current Spider interface (still in the early stages of development) can be seen in figure 3.

6.1 Test Driving

Spider supports storing components in a web-based repository, and the test driving of those components.

Developers can add components to a repository. This step requires using an HTML form displayed in a web browser, and includes specifying the component's author, name, version, and the collection of URLs that point to the component's class and jar files. Once this information is submitted, Spider creates an XML file representing the static component data (e.g. method signatures, field signatures) and stores this on the server. Spider then includes this in a list of known components that is presented to the developer.

The developer can then select a particular component from that list. Spider displays component information in the web browser. The information shown is based on the static component data stored in the earlier step.

As before, the component information is displayed using HTML, along with an HTML form that developers can use to type in the instructions they wish to use in their test drives. The instructions can include parameters that are either literal values, or are names earlier assigned to values retrieved in the same test drive, or in previous test drives.

The form can then be sent to the web server, which then executes the instructions in a "clean" environment. We define a clean environment to be a Java virtual machine in which only the test driving framework and any previous related test drives (if any) have been executed. The results from each of the instructions in

the test drive are then displayed in the resulting web page.

Any values returned by the instructions and that have been assigned a name are then also displayed in the page so that developers can reuse them in future test drives.

6.2 Technology

Spider uses W3C standards to present a web browser-based user interface to developers. The two standards used on the client side are the HTML and Cascading Style Sheets (CSS). Spider also uses the W3C recommended format Scalable Vector Graphics (SVG) (Duignan, Biddle & Tempero 2003) for the software visualisations.

6.3 Creating Visualisations

One of the intended benefits in performing the test driving is that component-specific visualisations can be created to act as documentation as to how the component can be used (and extended). These visualisations are created from XML-based execution traces captured from the executing components – driven by the invocations requested by the user.

The tool has a spy capability built into it that can detect events occurring within the executing environment of the component, and extract relevant runtime information about that event. That information gets saved to the server's filesystem as the afore-mentioned execution traces. With this information now saved, the user needs to specify a visualisation template through which the a desired execution trace will be interpreted. Once this template is selected (from a list of available templates previously created by our tool's third audience from section 3.4), and once the relevant execution trace has also been specified, then the tool transforms the execution trace content into visualisation content in the SVG format.

This SVG file is then sent to the developer's machine for viewing.

7 Component Interaction

Using our tool over the web presents challenges when interacting with component user interfaces.

We shall discuss these challenges with respect to the three general requirements documented in sections 5.3, 5.4, and 5.5. In doing so, we make reference to how we are meeting these challenges in our Spider prototype. Some of these challenges are similar to challenges faced by web-based simulation tools (Kapadia, Fortes & Lundstrom 2000).

7.1 Component I/O via the Web

The first challenge arises when you consider that the developer's machine is no longer the display environment for the test driven component. As a component is actually executing on the server-side, driven by instructions sent over http, any user interface that the component tries to display will display on a peripheral attached to the server.

This user interface needs to be sent back for display in the developer's web browser. The expressiveness of forms-enabled, CSS-enabled HTML is not really sufficient to mock-up a wide-range of possible component user interfaces. Wrapping the component in an applet embedded in a webpage breaks one key benefit of using a client/server model: that of being able to test drive components without risking the integrity of the developer's system or resources.

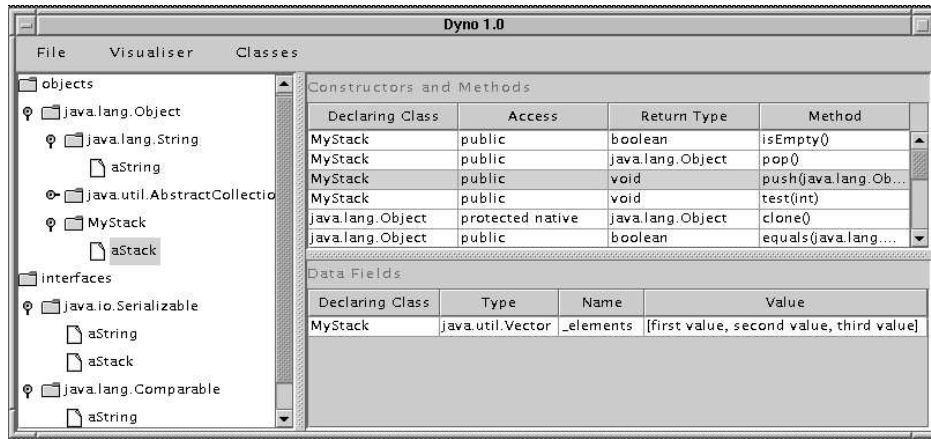


Figure 2: A browser in Dyno displaying loaded Java classes. The tree view in the left hand side shows a hierarchical inheritance view of the loaded Java classes. The component selected in the tree view has its methods listed in the top right hand side table, and its fields shown in the bottom right hand side table. Test driving is initiated by clicking on the table rows.

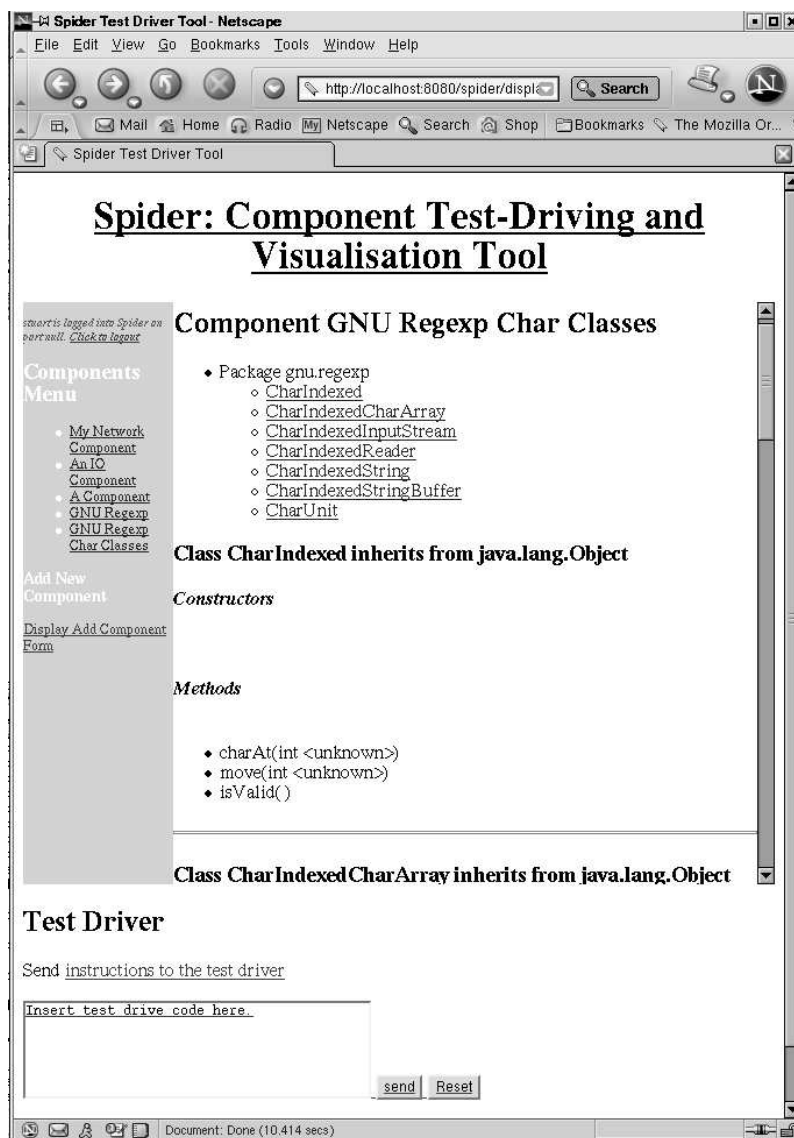


Figure 3: An early version of the Spider prototype. Spider uses HTML and CSS to display a component repository, and to display information relating to specific components. The left hand column contains a list of components currently available on the server. The top right hand side of the page shows the top of the listing for the GNU Regexp Char Classes component. Spider currently uses an HTML form in the lower half of the browser screen as the input device for test driver instructions.

Our solution to this first challenge is to give the developer a window onto the server's executing environment within their web browser. Applications such as VNC (VNC 2003) provide this functionality. VNC servers can be executing along side the components, with the VNC viewer applets embedded into web pages sent back to the developer.

There are two main costs associated in using this solution. The first cost is that of server resources. As each display needs to be kept uncontaminated by the test drives of concurrent users of the server, it is necessary to run a VNC server per user. This can pose a barrier to scalability and the effectiveness of the tool for sharing components and visualisations between large groups. The second cost is that the use of VNC from a web browser may require modification to any firewall that the developer is behind. Also, there is the need for the developer to remember and use an additional password to securely connect to the VNC server.

7.2 Component UI Context via the Web

The second challenge follows from the first. As the server's architecture may be different from the architecture of the developer's machine, code executing on the server may perform differently than it will when incorporated into the developer's new software.

We are currently prototyping in Java. While applications in Java do not need to be recompiled to execute on different architectures, subtle differences in the virtual machine may cause performance differences, such as the threading scheduler used. User interface components may adopt a default look and feel on the server environment than they would in the developer's environment. This could potentially confuse the developer as to the suitability of a component's appearance.

One solution to these problems would be to have multiple servers representing the common architectures that are currently being developed on. On the basis of browser information sent to the web server, the web server would execute the test drive on a similar architecture. While there would not be a duplication in the system resources available on the client's side (e.g. memory, cpu cycles), this would remove some of the more immediately obvious differences.

7.3 Component Event Notification via the Web

The third challenge is that of dealing with the event notification model that some components use. When a component's state changes, it may fire off event notifications to dependent observers (also known as listeners).

Given the client/server model that Spider uses, it becomes difficult for the server to prompt the client's browser to update the display to show events as they occur during the test drive. In effect, there is no way for the client's browser to "register" with the server without needing to continually poll the server. If this continuing polling were to be implemented, it would tie up a server port. The status of the browser continually requesting information from the server might also confuse a developer into believing the tool was faulty.

Our simple solution to this challenge is to then leave the event notification information for the visualisations that are viewed after the test drive is complete.

8 Tool Interaction

Using our tool over the web also presents challenges when meeting the tool's required utility.

8.1 Real Time Software Visualisations

Prompt and useful feedback from the test drive is important for the exercise to be of any use to the developer. The tool could easily display the values returned by the individual messages in the test drive sequence. However, this shows only one part of the side-effects resulting from a component's behaviour. A goal of the tool is to be able to create customisable visualisations from information captured by the system during execution.

Delivery of these visualisations would ideally be closely tied to the timing and the interaction mechanism of the test driver.

Ideally, these visualisations could be viewed as created (effectively a live stream) by the tool. Unfortunately such live streaming is not easily implementable in the standard formats that we sought to use. As well as this, there is currently no universally available animation format built into standards-compliant browsers. The W3C is currently advocating for the acceptance of the XML-based SVG, while Macromedia's Flash has a wide installation base amongst web users.

8.2 Timeline Navigation via the Web

The tool allows the repeating of a test drive (or test drives) from a clean environment. With this in mind, the the "back" navigation mechanism in a standard web browser might be made to work as a way of "re-pealing" state or action.

8.3 Creating Message Sequences

In earlier standalone prototypes of our tool the developer created component instances, and sent messages by clicking on the appropriately named operation in a table based on the active class at the time. Were the web-based prototype to require that a communication between the client and server occur for adding a message to a test drive, the resulting network communication and constant refreshing of the developer's page would become wasteful.

9 Summary

We are interested in enhancing understanding of candidate reusable components by providing developers with tool support. We are developing an architecture and a prototype tool to do this by providing the opportunity to gain first hand experience in querying and manipulating components, and then viewing visualisations created. The visualisations are created by transforming execution traces representing automatically captured runtime information. We are interested in reducing the resource cost in using the tool by tying the user interface into a mechanism already commonly available to developers. In this paper, we looked at the requirements that our prototype's user interface should meet, and then reviewed our proposed use of the web against these requirements.

The choice of using the web as the mechanism over which developers would interact with the tool was made for a number of reasons. These reasons were that it reduced the cost of installing and configuring custom tools on the developer's machine, and that the

client/server model that this in turn enforced leant itself to reducing the security risks to the developer.

However our choice of using the web created a number of challenges. These challenges had to do both with the method of controlling the test drive and the resultant output, and the method of handling extra user interaction required by the component beyond that given in the test drive commands.

These challenges could potentially add to the cost of using the tool, either in reduced functionality or in additional support required at the developer's machine. This in turn could reduce the impact that test driving and visualisation could have in making reuse a more desirable activity in lowering the costs associated with software development.

We have created a prototype tool that addresses some of these challenges and presents solutions.

References

- Cheon, Y. & Leavens, G. T. (2002), A simple and practical approach to unit testing: The JML and JUnit way, *in* 'ECOOP', Lecture Notes in Computer Science.
- Duignan, M., Biddle, R. & Tempero, E. (2003), Evaluating scalable vector graphics for use in software visualisation, *in* T. Pattison & B. Thomas, eds, 'Proceedings of the Australasian Symposium on Information Visualisation, Conferences in Research and Practice in Information Technology', Vol. 24, Australian Computer Society.
- Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1994), *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional Computing Series, 13th edn, Addison-Wesley.
- Kapadia, N. H., Fortes, J. A. B. & Lundstrom, M. S. (2000), 'The Purdue University network-computing hubs: running unmodified simulation tools via the WWW', *ACM Transactions on Modeling and Computer Simulation (TOMACS)* **10**(1), 39–57.
- Marshall, S., Biddle, R. & Tempero, E. (1999a), Dyno: A tool for dynamic interactive documentation., *in* 'First Symposium on Constructing Software Engineering Tools (CoSET'99)'.
- Marshall, S., Biddle, R. & Tempero, E. (1999b), Reuse of debuggers for visualization of reuse., *in* 'Proceedings of the Symposium on Software Reusability'.
- Marshall, S., Biddle, R. & Tempero, E. (2002), How (not) to help people test drive code, *in* J. Grundy & P. Calder, eds, 'User Interfaces 2002', Australian Computer Society.
- Marshall, S., Jackson, K., McGavin, M., Duignan, M., Biddle, R. & Tempero, E. (2001), Visualising reusable software over the web, *in* P. Eades & T. Pattison, eds, 'Information Visualisation 2001', Australian Computer Society.
- Ravichandran, T. & Rothenberger, M. A. (2003), 'Software reuse strategies and component markets', *Communications of the ACM* **46**(8), 109–114.
- Sun Microsystems (2003a), 'JavaDoc tool', <http://java.sun.com/j2se/1.4.2/docs/tooldocs/javadoc/>.

Sun Microsystems (2003b), 'The source for Java technology', <http://java.sun.com>.

VNC (2003), 'Realvnc web site', <http://www.realvnc.org>.