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

School of Mathematical and Computing Sciences
Computer Science

Visualising Reusable Software Over The Web

Stuart Marshall, Kirk Jackson, Michael McGavin,
Matthew Duignan, Robert Biddle, Ewan Tempero

Technical Report CS-TR-01/12
Octobert 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

Visualising Reusable Software Over The Web

Stuart Marshall, Kirk Jackson, Michael McGavin,
Matthew Duignan, Robert Biddle, Ewan Tempero

Technical Report CS-TR-01/12
Octobert 2001

Abstract

This paper describes an architecture we have developed for web-based visualisation of remotely executing software. The motivation for this work is to allow users of web-based software repositories to explore existing code components and frameworks, to see what they do, and create *interactive visual documentation* of that code based on the developer's actions. This visual documentation can be used to determine what the code or framework does, how it does it, and whether it can be reused in the developer's current project. The architecture is designed to be language neutral, and supports customisable software visualisations, viewable through widely available plug-ins to standard web browsers, and does not require modification of the source code being visualised.

Author Information

The authors are postgraduate students and lecturers in Computer Science at Victoria University of Wellington, New Zealand.

Visualising Reusable Software Over The Web

Stuart Marshall **Kirk Jackson** **Michael McGavin** **Matthew Duignan**
Robert Biddle **Ewan Tempero**

School of Mathematical and Computing Sciences
Victoria University of Wellington
Wellington, New Zealand
Email: Robert.Biddle@mcs.vuw.ac.nz

Abstract

This paper describes an architecture we have developed for web-based visualisation of remotely executing software. The motivation for this work is to allow users of web-based software repositories to explore existing code components and frameworks, to see what they do, and create *interactive visual documentation* of that code based on the developer's actions. This visual documentation can be used to determine what the code or framework does, how it does it, and whether it can be reused in the developer's current project. The architecture is designed to be language neutral, and supports customisable software visualisations, viewable through widely available plug-ins to standard web browsers, and does not require modification of the source code being visualised.

Keywords: Software Visualisation, Web-based Code Repositories, Code Reuse

1 Introduction

We are investigating tool support for software reuse. This support is aimed at reducing the cost to the developer — in terms of time and effort — to make code reuse more attractive. One cost is that of understanding what a particular piece of code does, how it does it, and whether it will work in the developer's current project.

We have been developing tool support to reduce the cost of understanding code intended for reuse. The standard support for understanding code is textual documentation. Our approach has been to use *dynamic software visualisations*, which help software developers understand what the code does and how to use it, by allowing them to watch what happens when they explore it. We also envisage authors of reusable code will create their own "visual documentation" showing how they believe the code should be used. We call this "visualisation as documentation."

One of the challenges that software reuse must meet is providing access for software developers to potentially reusable code. This has led to the development of web-based software repositories (see [Eichmann et al., 1994] for example). This means we must provide some way for visualisation as documentation to work over the web. We are developing a Visualisation Architecture for REuse, *Vare* (pronounced "vahray"), which allows web-based visualisation of remotely executing software. This architecture allows potential reusers to execute code at the repository web site, while viewing visualisations of the code's behaviour locally. In this paper, we discuss our project as work-in-progress, and present our first prototype.

The paper is organised as follows. In the remainder of this section we develop the motivation for our work in more detail. In section 2, we present some of our earlier work in stand-alone visualisation systems for reusable code. Section 3 presents the new architecture, and section 4 outlines how systems with this architecture might be used. We give a summary of the relevant technologies in section 5, and present our first prototype in section 6, and we then present our conclusions.

1.1 Why Visualise Reusable Code?

Developers often reinvent code for functionality that could have been provided by existing code. One reason for this is that code reuse is not a trivially easy task, and requires — amongst other things — that developers be able to easily understand existing code so as to be able to identify reuse possibilities. Whether the reusable code be a single class or an entire framework, without understanding what the code does and how it does it, the developer will not be in a position to reuse it.

Standard forms of documentation are often difficult to use and understand, due to the limitations of text, and the fact that often it is trying to describe dynamic runtime behaviour in a static medium. Using software visualisation offers a chance to use the human ability to process complex stimuli.

1.2 Tool Support for Code Reuse

Our research looks at tools that can support developers. There are a variety of tools that can be used to support code reuse. Different tools address different costs in the code reuse process. Such costs are: finding reusable code; understanding reusable code; and inserting reusable code into a new context.

Web-based code repositories [Eichmann et al., 1994] help to reduce the first cost by collecting reusable code together in one location, that can then be stored as a database to facilitate easier searching. The usability of such repositories is very important, and needs to support the users ability to easily locate and identify suitable components [Clayton et al., 2000]. Smart cut'n'paste tools [Wallace et al., 2001] help to reduce the third cost by helping developers handle the connections between new code and reusable code, and making the two compatible. The work we present here is primarily interested in reducing the second cost, although our new tool also helps to reduce the first cost.

2 Experience Visualising Reusable Software

The Vare architecture has grown out of our earlier work done on visualising reusable software. This section summarises that work, to provide the context for the decisions we made. Recent projects are Dyno and Fire, as described below.

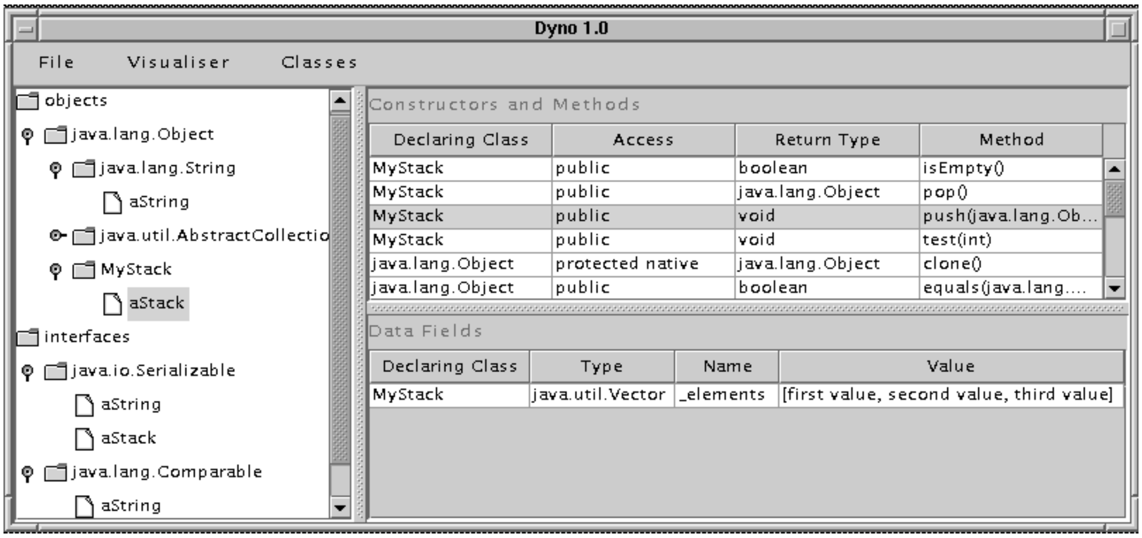

Figure 1: *The main interface of Dyno, showing the available classes and objects in the system. The component object "aStack" is currently selected and its methods and fields are shown in the tables to the right.*

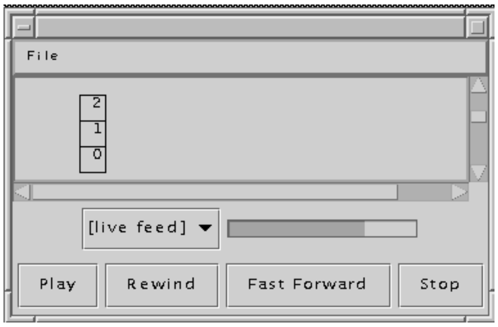

Figure 2: *A view of a stack in Dyno, about 80% of the way through its animation. The control panel allows for moving in both directions through the animation and gives the user some idea of the current position in the sequence.*

2.1 Dyno: Visualisations From Test Driving

Dyno is a prototype of a software visualisation tool that lets a developer create visual documentation of reusable code components written in Java [Marshall, 1999]. The visual documentation is derived from information gathered from events that occur during execution of the code components. The execution is driven by the developer, in an activity we refer to as *test driving* [Marshall et al., 1999].

Test driving reusable code is analogous to test driving an old car you're interested in buying. You are test driving the car to see whether it's specifications match what you want in a car, and to see how it handles. You are not (primarily) checking to see whether the car works with respect to its *own* specifications — you wouldn't take it onto the motorway if you hadn't already checked with the warrant of fitness that the brakes work. You are instead testing to see whether you like how it handles, how comfortable a fit it is, and the feel of the ride. Similarly with test driving reusable code, we are more interested in whether the code does what we need, in a way we can easily fit into our current project, and performs satisfactorily with respect to timing and resource considerations. We assume the reusable code is correct with regards to its original specifications and contains no bugs, although this may be checked during a different phase in the code reuse process.

Test driving reusable code is not a new activity. Developers commonly create small test programs (or *test harnesses*) that invoke the public methods of a component. These test harnesses are used to see what the side-effects of calling a method are, what behavioural changes occur as different parameters are used, and how objects in the component can be used in collaboration with other components. These test harnesses are usually written manually, thereby taking up time and effort, and can be a source of confusion and error. Errors in a test harness can give a misleading view of the component's work, and such errors may cause severe misunderstanding of the component.

In Dyno, software visualisations are created from events that occur as code executes during a test drive. Two different types of people may decide to use Dyno to create software visualisations 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 create visual documentation of their own components, so as to reduce the need for future developers to create it themselves.

The documentation resulting from the visualisations 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 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.

Figure 1 shows a common view of Dyno. The interface is organised with known classes (indicated by folders) and existing objects (indicated by pages) in the panel on the left (the *object browser*). 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. When a method is invoked on an object, Dyno will then prompt the user for any parameters required. The user can select objects from the object browser 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 system are converted to a form suitable for visualisation. Figure 2 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.

2.2 Fire: Visualising Frameworks

Object-oriented frameworks are bodies of reusable code that an advanced programmer can use in many situations to quickly produce a new application with less effort than starting from scratch. However, using a framework requires an intimate understanding of how it will interact with the extensions that the programmer will provide — knowledge that has to be learned. To acquire knowledge of a framework's behaviour can take a great deal of effort, and increases the costs of development the first time a framework is used. This is seen as one of the major problems with frameworks [Johnson, 1992].

Fire (Framework Interaction for REuse) is a prototype design for supporting visualisation of framework interactions, which aids the identification and understanding of the critical interactions between framework and user objects[Henderson-Sellers and Meyer, 2000] . Our prototype works with C++ frameworks, and provides diagrams based on UML. The tool works with C++ programs compiled in the usual way, and allows a description of any framework involved by use of a framework information language. The prototype then provides dynamic visualisation of the running program, highlighting the role of frameworks.

Our prototype is designed to visualise three different *views* of a running program: a static UML class diagram, a UML sequence diagram, and an instance diagram. The prototype allows simultaneous viewing of any of these, and also allows the user to enlarge or reduce their sizes as shown in figure 3. The visualisations are animated, and updated immediately when the events are received. The user can also move objects around on the screen to customise the layout.

The purpose of the diagrams is to help users understand how the frameworks are used. They provide this help by using a description of the framework (provided in a concise, machine-parsable format) to distinguish classes and objects within the frameworks, and highlight their usage guidelines. The dynamic display allows the user to follow the flow of control, especially important to understanding frameworks because of the effects of inversion of control, and the polymorphic effects of subclasses and overridden methods. To allow control flow to be followed more carefully, the prototype also allows the user to pause program execution, and to hide irrelevant classes or objects.

3 Vare Structure

In this section we present Vare, an architecture for supporting visualisation of software in a distributed environment. The design supports components in multiple languages and configurations (e.g., complete programs or just code fragments), and provides user control for all parts of the visualisation process.

Vare is a client/server architecture (see figure 4). The client side allows the user to control the various activities associated with creating and viewing a visualisation of a reusable component. The component repository interface allows a user to choose a component from the component repository, creating a component set ready for test driving.

The test drive interface allows the user to choose an engine type from the engine repository (in the case where multiple languages are used, or multiple engines for one language are available), and then control the test driving of the chosen components.

The output from an engine is a test drive report, which is stored in the test drive report repository. A report can then be fed into a transformer, which transforms the report into a form suitable for producing a visualisation. The client-side transformer repository interface allows the user to specify the transformer to use from the transformer repository, and the report it should act on.

The resulting visualisation is stored in the visualisation repository. The client side visualisation interface allows a user to specify a visualisation from the repository, and to control its presentation. We now discuss the components of the architecture in more detail.

3.1 Repositories and Processes

Vare has five repositories and two process classes. The five repositories are; component, engine, transformer, test drive report, and visualisation. The two process classes are; engine and transformer.

Component Repository The Vare component repository is intended to support uploading of new code by software developers — where such new code may represent either an entire application or a code fragment. Associated with each component is the following metadata:

- documentation from the author describing how to use the code — most likely textual in nature, e.g. Javadoc for Java code.
- keywords categorising the code
- links to other code required for execution
- the implementation language (and language version) of the code.

As software visualisations are created of the components, these visualisations will also be linked in.

As users browse through the component repository, they will begin to select a subset of the available components that they are interested in. This interest may involve a desire to create and/or view visualisations of the selected components. This set becomes their personal component set, as discussed below.

Engine Repository The engine repository creates engines to handle test drive requests from a user. The engine repository will create an engine for each user. As it is expected that the minimum number of languages supported by engine type will be one, the engine repository may need to allocate an instance of more than one type of engine per user. This will occur should the user need to test drive components that do not share a common language.

Transformer Repository The transformer repository stores all the different types of transformers. When a request to create a visualisation is made, an instance of the relevant transformer is created, pointed at a test drive report as input, and told to output to the visualisation repository under a user-supplied name.

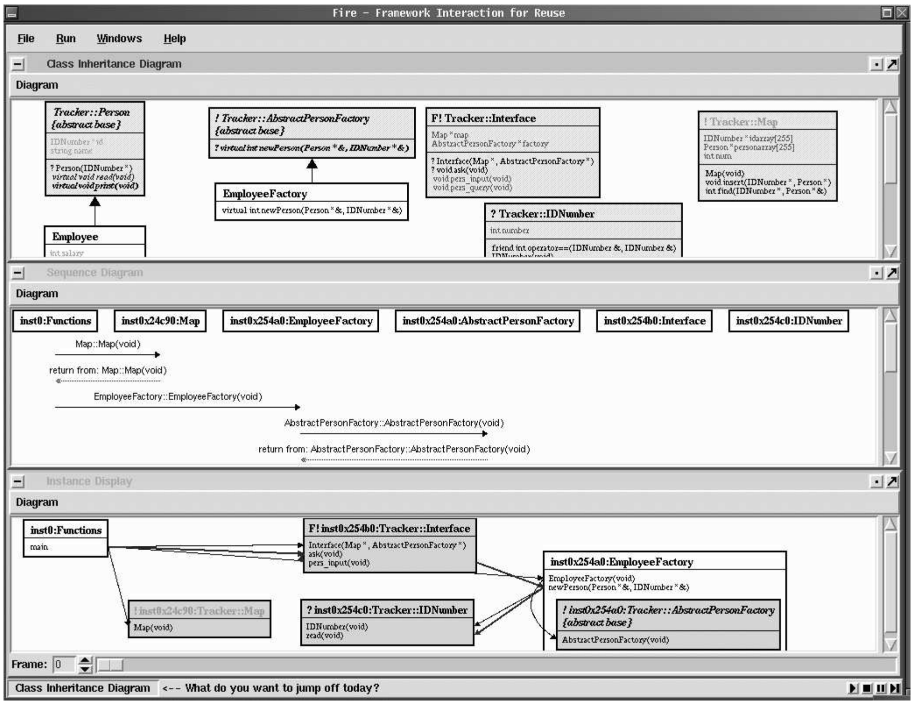

Figure 3: *The main interface of the Fire prototype, showing a class diagram, a sequence diagram, and an instance diagram.*

Test Drive Report Repository Test drive reports are stored in a repository, so that at some later point they can be used as the basis for the creation of a visualisation. The test drive reports contain all the information that may be relevant for a visualisation — such as (for OO languages) the order of object creation, method invocations (on what, with what), field accesses and field modifications. To facilitate easy look-up later on, test drive reports are stored with some identifier (supplied by the user who performed the test drive). They may also have some associated annotations to give other users an idea as to what the originating test drive did.

Visualisation Repository The visualisation repository contains all the visualisations that have been created. The visualisations are linked with the components that they describe. They also have extra metadata, such as who created the visualisation, and any ancillary notes that people have since attached to the visualisation as an extra aid to understanding. Users request visualisations from the visualisation repository, and the file is sent back to the client side for display. The user may request a visualisation that is still currently being created, so as to support real-time viewing of test drives. As visualisations are stored rather than regenerated each time, users can edit and annotate the visualisations and the changes remain for other users to see.

Engine Process Engines handle test drive requests from users. These can include such things as — in the case of OO languages — object creation, method invocation, field access and field modification. As Vare supports multiple languages, there may be more than one type of engine, with each engine supporting one or more languages. As well as this however, engines also "spy" on what happens during the test drives (e.g. who calls what, when, and with what arguments) so that at some later point in time a visualisation can be created from this information.

An engine may have two sets of input/output. The first set corresponds to:

- *input:* the components that the user wishes to test drive and the way in which they wish to test drive them (e.g. method invocations, field access/modifications).
- *output:* the results of method invocations and so forth, as well as the spy reports. The spy reports are what is specifically referred to as the test drive reports in Vare.

The second set of input/output corresponds to the input/output of the code being test driven. The code being test driven may have its own user interface, and this interface is presented to the user by linking the engine to the test drive interface in the diagram.

Transformer Process Each request by a user for a new visualisation from a test drive report requires the execution of a transformer — a piece of code that directs a particular type of visualisation. There will be a transformer for each type of visualisation, such as a "Class diagram" transformer that produces a class diagram animation, or a "Stack diagram" transformer that produces a stack animation. Some transformers may only work with certain components, while others (such as the two examples listed above) may work with a wide range of components.

The transformer takes a test drive report as input and directs the resulting visualisation to the visualisation repository. If the test drive report is still being created when the visualisation is requested, the transformer continue to poll the test drive report repository, allowing the transformer to exist in parallel with the code execution.

3.2 Component Set

The component set contains the components that the user is currently interested in. A component set is maintained for each user currently logged in. The component set acts as one input to the engine Process, for the purposes of determining what is directly available to the user for test driving.

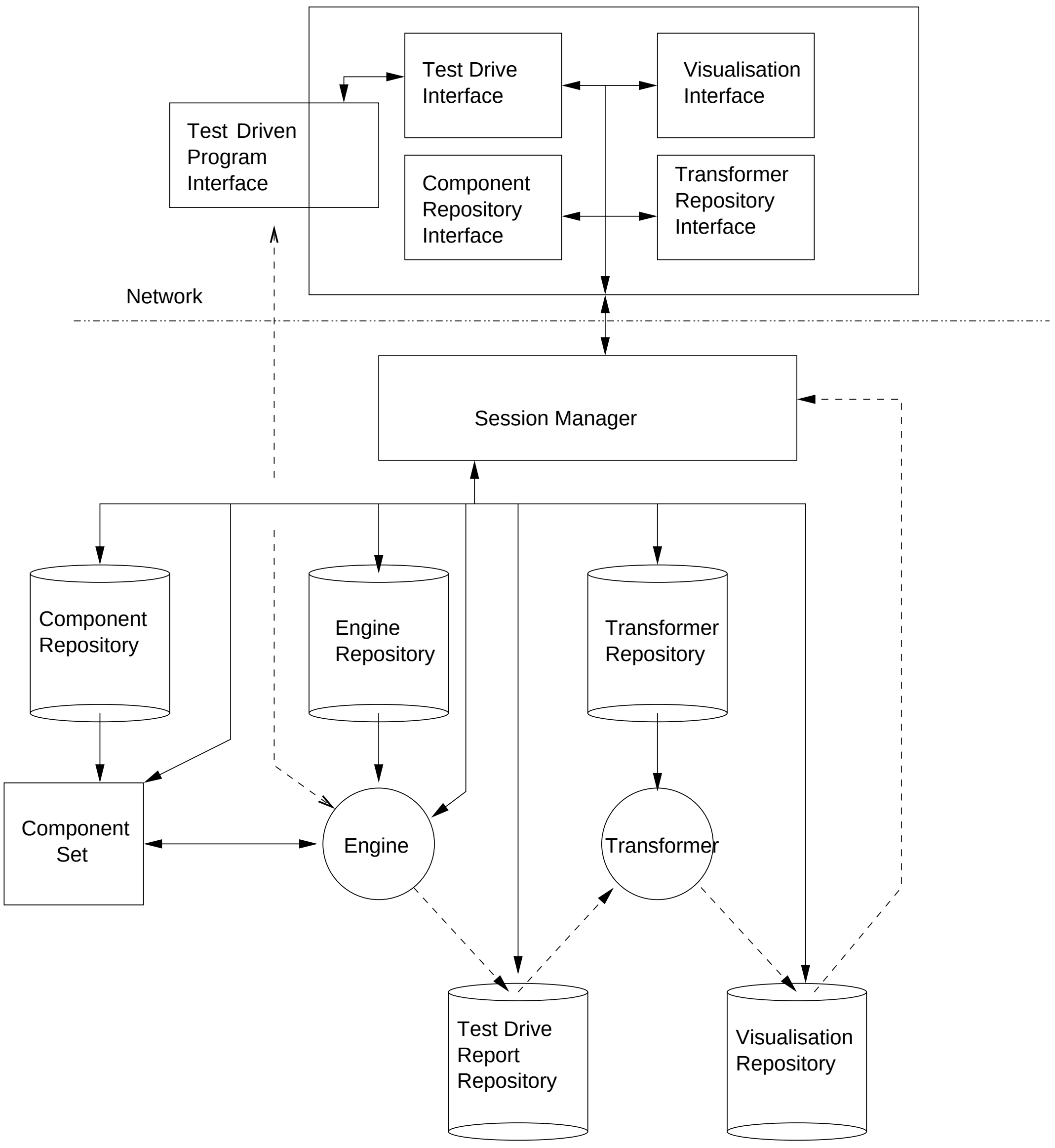

Figure 4: The Vare architecture is based on a client/server model, with the server being split into repositories and processes. Dashed lines represent test drive or visualisation input/output, while solid lines represent control, queries or responses.

3.3 Session Manager

Given that Vare supports multiple clients in a client/server model, some state information must be kept for each client. This in effect creates a session for each client. A session keeps track of what its client has done so far, and what in the server is currently associate with that client. For example, with multiple clients test driving concurrently, there may be many engines currently on the server. A session identifies which particular engine to send test drive requests from a particular user. Similarly, when a user asks to list the components they have currently displayed interest in, the session knows which component "box" (as shown in the diagram) corresponds to the querying user.

The session manager handles all the current active sessions. Our initial assumption is that a session is started by the act of logging in, and is ended by the act of logging out.

3.4 Client Side

Lastly, we discuss the client side. The client has five main elements. Two of these; component repository interface, and transformer repository interface, are primarily interfaces to the associated repository on the server side. In the latter case, the user also uses the transformer repository interface to specify what test drive report to use as input, and what name to give the output visualisation.

The test drive interface allows the user to control what executes — and when — in the engine associated with the current user and implementation language. The test drive interface is also used to specify the name to be given to the test drive report.

The visualisation interface allows the user to view a visualisation, and control how they view it (e.g. tape deck style controls for moving through a visualisation). These interfaces make use of the session manager on the server side to ensure that they are talking to the right engine instance or component set. They must therefor supply some form of ID — supplied at login — to the session manager to identify themselves.

The last part on the client side is the test driven program interface. This allows the user to, on the client side, give input and output to the interface of the program being executed on the server side. Not all programs have their own interface, so this won't always be needed. However it is available should the need be there.

4 Vare Usage

In our past work on software tools to provide visualisation of reusable software components, we have conducted usability evaluations and found that high usability is an important but difficult element of the software tool design [Marshall et al., 2002]. We expect the intended users will invest time in learning about the reusable software, but they will want the Vare software user interface to be easy to learn and use immediately. This is consistent with more general investigation of CASE tools, which also shows that poor usability often leads to avoiding use of such tools [Iivari, 1996].

Tools that support code reuse help developers by reducing the time and effort required for successful reuse. If a tool requires non-trivial time and effort to use, this will offset any savings made and render the process less appealing to developers. Developers are expected to be casual users of the tool, and are unlikely to wish to invest significant amounts of time in training. Easy learnability and intuitiveness are therefore important characteristics in a successful implementation of this tool.

With all this in mind, we are exploring a new approach to make Vare highly usable. We suggest that a developer's impression of Vare be a combination of an animation studio and a video store. The purpose of the animation studio is to animate possibly reusable code, and the purpose of the animations (visualisations) are to give a clearer idea of what the code is doing, and how it is doing it. The purpose of the video store is to store reusable components and associated animations, along with other forms of documentation.

The interface is proposed so that the flow of control for creation of software visualisations mirrors that of a typical movie creation process. The developer selects the code they are interested in (*selecting the actors or characters in your movie production*), puts them in an initial state (*determining the environment the characters will exist in*), executes the code (*creating a script of events and actions that occur*), and then selects a visualisation through which to view that execution (*selecting a director to create a visual interpretation of the script*).

The animation studio works on a movie studio analogy, and the users' navigation through the tool reflects this. The following subsections describe a typical process the user might go through in creating an animation.

Selecting the Actors

The central objects of a movie are its characters. Following the movie studio analogy, the central objects of our animations are classes and objects. When creating an animation, the user initially selects which characters (class and/or objects) they are interested in. When creating an animation from an existing application, the characters will already be decided (analogous to writing a screenplay from a book). When not using an existing application as the source, the user can load in new classes by specifying their URLs, and create new objects by calling constructors.

Once the user is satisfied they have populated the animation with the characters they want, they can then move on to the next phase.

Setting the Scene

The next phase is to put the classes and objects in a state that the user wishes to start the animation at. This can involve preliminary modification of object state. In the case of creating an animation from an existing application, the setting of the scene is handled by the application itself, so no work is necessary in this step.

Writing The Script

The script is the execution to be created by the engine component of the tool. To generate this script, the tool requires some direction from the developer as to what code is to be executed. The developer has a couple of options here. They may execute an entire application, or they may execute particular sections of the code (i.e. individual or sets of methods) in an order of their choosing.

Selecting A Director

Having created a script, the developer now chooses a director to turn their script into an actual production that can later be viewed.

Directors represent code that can translate raw scripts into a visualisation that can be viewed by the developer. There are different *types* of visualisations that are likely to be of interest to developers, such as, amongst others, specific data structure visualisations and UML diagram visualisations. To reflect this, there are different types of directors. Each director is capable of creating a specific type of software visualisation from a given script, such as a Stack Visualisation Director that can create visualisations of stacks, and a UML Class Visualisation Director that can creation UML class diagrams.

There is also the ability to *create* new directors that will be able to do new types of visualisations. This allows for customisable visualisations and does not limit developers to the select views that the tool's creators thought up, or thought worthy, at the time.

Editing The Video

Once a visualisation/video has been created, it may be worthwhile to annotate it. Should the visualisation be made available to other developers, it may be helpful to include some text at key points in the visualisation to underline important areas, features or pitfalls in the code being visualised.

A developer may decide it is worthwhile to edit out certain scenes in the visualisation as being unnecessary and possibly confusing. It is likely that a developer will usually decide to do this only if they are the author of the visualised code. Code authors are in the best position to match the visualisation to what they know about how the code is supposed to work, and be able to make such decisions in an informed manner. Developers wishing to *understand* a component by creating and looking at a visualisation may well not be in a position to correctly know when this step should be applied.

The Video Store

Vare's main interface is a web site that a developer can interact with through a standard web browser. Having identified themselves to the web site, the developer is placed in a code repository. This reflects the fact that the primary reason they are using the tool is to find appropriate reusable code.

A developer can create a personalised view of the code repository by creating their own "favourites" list. This list contains components that the developer has singled out for continuing inspection, and can also (if specified) contain the components that the developer has uploaded into the code repository.

As with many online code repositories, the reusable code is ordered by genre such as "networking", "pattern matching", or a variety of keywords.

5 Technologies

To support developers test drive and visualise code over the web, Vare implementations will need to employ a variety of technologies. Some of these technologies were used in Dyno and Fire, however a number will reflect the web-based nature of Vare.

5.1 Engines

At the heart of any Vare implementation are the engines. Each engine instance handles the executing test driven code, and is responsible for interrogating the executing code for events and information that will be used to create the software visualisations. There will be different implementations of the engine to handle different languages, with engine implementations made to be plug and play compatible.

The engine will wrap around the candidate reusable code in slightly different ways depending on the reusable code's language. With C++, the engine will wrap around the program, whereas with Java the engine will wrap around a virtual machine that contains the program. Engines will likely use similar technologies to those found in debuggers. The debugging technologies include programming libraries for event-detection and interrogation of executing code, such as the Java Virtual Machine Debugger Interface (JVMDI) and Java Debugger Interface (JDI). They also include the reuse of entire debugging tools, such as the C++ debugger GDB [Biddle et al., 1999].

5.2 Communication and Control

A Vare implementation will also need technologies for communication between different parts of the tool scattered across networked computers. For this communication, we expect technologies such as XML [World Wide Web Consortium (W3C), 2000a] and SOAP [World Wide Web Consortium (W3C), 2000b] to be useful.

XML has the benefit of being easy to construct using widely available programming libraries, and it is also relatively easy to turn the XML into other formats, useful when constructing software visualisations. SOAP is a particular XML technology for sending commands for object/class manipulation. This technology is used to convey user intentions during test driving to the engine that actually inspects and executes the code. An advantage of using SOAP is that it is largely language independent (within the OO programming paradigm) and works with all the languages we have initially targeted.

SOAP and XML might be used as the input and output for an engine. The engine might have one input channel to handle XML SOAP requests, and two output channels to handle the replies to the SOAP requests and an execution XML stream. The execution XML stream contains the information gathered by the engine, such as method calls and returns.

5.3 Visualisation

For actual visualisation output, a new and useful display technology is Scalable Vector Graphics, SVG [World Wide Web Consortium (W3C), 2001]. SVG supports the description of vector-based animations through XML, resulting in visualisations that are typically low bandwidth. Our use of SVG also has the additional advantage that it shares a common heritage with XML, and could even be converted from the raw XML output of the engine using a stylesheet if necessary.

SVG has a number of properties that make it ideal for the Vare architecture. In particular, SVG's support for animation is vital to display an important set of visualisations. However, what gives SVG its real strength is its integration with standard web-based technologies. SVG can be embedded in web pages, and interact with the web browser via Javascript and hyperlinks. This allows the user of Vare to view visualisations in their context of documentation and related visualisations. The degree of interaction that SVG provides allows the user to fine tune the visualisations as they view them. For example, in the video metaphor this could include start, stop, and speed controls. SVG visualisations would also allow users to seek or hide various additional levels of information — in the SVG itself, or as another hyperlinked web resource.

SVG also allows the user to zoom in on fine details or zoom out to see the full display for context. This may prove useful, although it is yet to be seen how well SVG can perform with the highly detailed visualisations that would make zooming valuable. Another potential advantage of SVG's interaction is that Vare's transformer may find it difficult to generate the perfect layouts for elements in dynamically created visualisations. By allowing the user to fine tune the layout of visualisation elements this problem is somewhat alleviated. Most importantly, SVG's open and XML based nature allows the dynamic creation of tailor made visualisations for efficient and rapid deployment via the web.

6 Vare Prototype

The current status of our work is that there are prototypes of the engine and client software. Together, these can generate XML streams representing the code being test driven. We are currently working on viewers that will require transforming the XML to suitable SVG, and displaying the SVG.

6.1 AT: An Engine prototype

AT (Abstraction Tool) is the working name for an engine currently in development, designed to control around the GDB, the GNU Debugger.

In the past we have created several visualisation tools that use GDB to extract information from an executing program [Biddle et al., 1999]. As GDB has been built with a primarily human interactive interface however, we have discovered from experience that it is complicated to control from another application. AT converts the GDB output to a pre-defined XML output. By doing this, it wraps the complexity of GDB in a way that can be integrated into the Vare architecture.

AT has a command line interface for testing purposes, as well as a SOAP interface (see section 5) that allows it to be controlled by another tool such as those used in the Vare architecture. Output from AT comes either through standard output, or sometimes as return values from SOAP method calls if it is being used with a SOAP interface.

On invocation, AT starts GDB in a hidden session. Having been instructed to open a program, AT uses GDB to open it and immediately extracts information about available types. This information is then made available to the client, which can decide what types it wants to follow more closely. AT sets appropriate breakpoints in GDB to allow it to extract this information later.

Once the client has indicated which classes and methods it is interested in following, AT can execute the program. As the program runs, events such as method calls, method returns, class instance creations and deletions are converted to an event form. These events are output as part of an XML stream to be interpreted and displayed by other parts of the Vare architecture.

The AT project has also addressed an important part of Vare: the representation of the information produced by the engine. By specifying the language used for this representation, we allow for separation of the engine from the rest of the system, meaning that different engines can be used within a Vare based implementation.

The language used to represent the information produced by the engine is the Process Abstraction Language (PAL). PAL is a Valid XML language, that is, there is an XML Document Type Definition for it. It has been designed to represent all the information necessary to create the kinds of visualisations we expect a Vare based system to support, and has also been design with extension in mind should new information be required [McGavin, 2001].

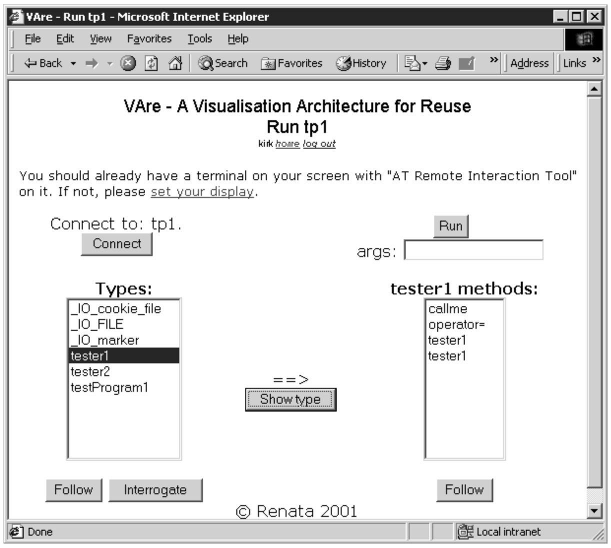

Figure 5: The prototype client allows the available types (classes) to be shown, and the methods for each type.

PAL is designed to describe object-oriented programs. This means it has elements for fully describing classes — their methods, fields, and superclasses. PAL can also describe the run-time behaviour of such programs, including objects (instances of classes), run-time representations of classes (such as meta-classes), method calls (messages) and their actual arguments and return values, and different threads of control. These elements include means to identify different instances of the same thing (for example, different calls of the same method by one object on another). PAL can also associate elements with source code (or other artifacts) if that information is known by the engine.

A PAL session represents an XML document, so the raw session can be stored in files (or a database) and read back later quite conveniently. This means it can be used in the test driver report repository, both for storage and for returning to the client. Also, being text and XML based, it is possible to edit it using a text editor or other text processing tool. This allows parts of the session to be adjusted for other uses, such as demonstrations.

6.2 Client prototype

The client prototype uses a mixture of Java Server Pages (JSP) and Javascript in a browser. The JSP provides the connection between the web server and AT. The Javascript is used to change the pages on the client side to reflect what the browser receives from AT.

Figure 5 shows the browser view of the code `tp1` (a C++ executable from a repository). There is negotiation that goes on before this view allowing the user "kirk" to identify himself to the remote repository, and the repository's presentation of available pieces of code.

Once the user has identified the code he is interested in, the server uses AT to query the code to determine what types (classes) are available (shown in the list on the left). Selecting a type results in the system again using AT to determine what methods are available for that type. The user has the ability to determine what events he is interested in. "Following" a type generates events corresponding to the creation and deletion of instances of that type, whereas "following" a method generates events corresponding to calls and returns of that method. "Interrogating" a type is a shortcut for "following" all methods in that type.

Once the user has specified what events he is interested in, he can then "run" the code. The result is a stream of events from the program described in XML.

7 Conclusions

We have described Vare, an architecture we have developed for visualisation of reusable software in a web environment, and presented an early prototype for it. The architecture supports customisable software visualisations, viewable through widely available plug-ins to standard web browsers, that do not require modification of the source code being visualised.

This project is still work in progress, and our prototype has been primarily aimed to demonstrate the feasibility of the architecture. The prototype convinces us that the architecture is feasible, however there are still some issues to be resolved.

The motivation for this work is to provide support for web-based software repositories, in particular to allow users to explore how to use code and frameworks stored, and to help understand what they do. However, we do not regard this to be the only use of Vare. We believe this architecture can provide a general architecture for software visualisation using the web.

References

[Biddle et al., 1999] Biddle, R., Marshall, S., Miller-Williams, J., and Tempero, E. (1999). Reuse of debuggers for visualization of reuse. In *Proceedings of the Symposium on Software Reusability, '99*, pages 92–100. ACM Press.

[Clayton et al., 2000] Clayton, N., Biddle, R., and Tempero, E. (2000). A study of usability of web-based software repositories. In Gray, J. and Croll, P., editors, *International Conference on Software Methods and Tools*, pages 51–58, Wollongong, Australia. IEEE Computer Society.

[Eichmann et al., 1994] Eichmann, D., McGregor, T., and Danley, D. (1994). Integrating structured databases into the web: The MORE system. In *First International Conference on the World Wide Web*, Geneva, Switzerland.

[Henderson-Sellers and Meyer, 2000] Henderson-Sellers, B. and Meyer, B., editors (2000). *Understanding Frameworks through Visualisation*, Sydney, Australia. IEEE Computer Society.

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

[Johnson, 1992] Johnson, R. (1992). Documenting frameworks using patterns. In *Proceedings of OOPSLA'92*, pages 63–76, Vancouver, B.C., Canada. ACM Press.

[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)*.

[Marshall et al., 2002] Marshall, S., Biddle, R., and Tempero, E. (2002). How (not) to help people test drive code. In *The Austrialasian User Interface Conference*.

[McGavin, 2001] McGavin, M. (2001). Extracting software re-use information for visualisation tools. Honours Report, School of Mathematical and Computing Sciences, Victoria University of Wellington, New Zealand.

[Wallace et al., 2001] Wallace, G., Biddle, R., and Tempero, E. (2001). Smarter cut-and-paste for programming text editors. In *The Austrialasian User Interface Conference*, pages 56–63, Gold Coast, Australia. IEEE Computer Society.

[World Wide Web Consortium (W3C), 2000a] World Wide Web Consortium (W3C) (2000a). Extensible markup language (xml) 1.0 (second edition). W3C Recommendation `http://www.w3.org/TR/REC-xml`.

[World Wide Web Consortium (W3C), 2000b] World Wide Web Consortium (W3C) (2000b). Simple object access protocol (soap) 1.1. W3C Note `http://www.w3.org/TR/SOAP/`.

[World Wide Web Consortium (W3C), 2001] World Wide Web Consortium (W3C) (2001). Scalable vector graphics (svg) 1.0 specification. W3C Proposed Recommendation `http://www.w3.org/TR/SVG/index.html`.