Aspects to Visualising Reusable Components

Stuart Marshall, Kirk Jackson, Craig Anslow and Robert Biddle

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

Abstract

We are interested in helping developers reuse software by providing visualisations of reusable code components. These visualisations will help determine if and how a given code component can be reused in the developer's new context. To provide these visualisations, we need both formatted information and tools. We need a format to describe the visualisations in. We need tools to create the visualisations. We need a format to describe information about the component and its runtime usage, and we need a tool to gather this information in the first place.

In this paper, we discuss our two wish-lists for the required information formats. We set this against the background of software visualisation and code reuse research. Currently we are working with components from object oriented languages, specifically Java.

Keywords: Software Visualisation, Test Driving, Code Reuse.

1 Introduction

For all the benefits that reusing code is claimed to be able to deliver, it is perceived that code reuse is not as widespread or as efficiently implemented as it could be. Certainly, code reuse does happen on some levels. A common example of this is the increasing range and availability of libraries and APIs for the Java platform that offer rich opportunities for reuse. But even where code reuse is possible, often the rewards in time and effort saved are not as great as they could be due to problems in the process of reuse.

There have been several areas of cost identified in the reuse process, where cost is measured in time, effort and financial terms. One such area is the time and effort required to understand if and how a given fragment of old code (referred to as a *component*) can be reused (Wilde 1994). Components can range in size, from individual code blocks to a suite of applications.

We are researching ways to reduce this cost by creating visualisations of the static and dynamic information present in a component (Alonso & Frakes 2000), that can then be presented to developers who are looking for an old solution to a new or old problem. While other costs of reuse do exist – notably the time to search for potential candidate components for reuse and the financial cost of purchasing reusable components – we base our discussions in this paper solely on the cost of understanding.

Our goals are to identify what information is important in deciding if and how a component can be reused, and develop tools to allow developers to create and view visualisations of this information. We also wish to to be able to easily distribute these resulting visualisations, and allow for sharing of information and easy and safe experimentation with components. To achieve these goals, we need methods of collecting, storing and transporting component information, as well as a method of converting the collected information into a visualisation format. As part of this, we have developed a wish list of what we want to see in software visualisations of reusable components, as well as a wish list for the characteristics of the intermediary format that would carry this information.

We have been working with widely-available Java debugger libraries to collect information gathered from developers' experiences of using the component. We have also been working with XML-based technologies for the encoding and transportation of this information.

2 Code Reuse

The principle reasons for wanting to reuse code are to save time and effort in both development and maintenance of quality software. This, the argument goes, is achieved as a result of the reuser not having to develop a new solution to an old problem. The reuser may also, depending on how the act of reuse is implemented, receive the benefit of accessing a common solution, used in multiple places by possibly more than one person. This benefit is demonstrated when any improvements or fixes to the algorithm or technology behind the solution are made. With a common solution, improvements can be more easily propagated to all the places the problem exists than if developers had each created their own solution that needed individually updating. This is true regardless of the size of the component being reused.

Research in the field of code reuse has been conducted for decades (McIllroy 1968), and ranges from examining how to reuse code, to what makes code reusable (Mili, Mili & Mili 1995), through to metrics to measure code reuse (Frakes & Terry 1996) (Ferri, Pratiwadi, Rivera, Shakir, Snyder, Thomas, Chen, Fowler, Krishnamurthy & Vo 1997). Software developers do reuse code, examples of which are the libraries that come with the Java Development Kit, the act of copying and pasting code from one place to another, and interacting with existing applications (e.g. databases and browsers) to create new functionality. The very act of writing code by placing algorithms inside methods, and invoking those methods from more than one place is a simple example of code reuse in action.

Code reuse can range from using code in a similar context to where it was originally used (and intended for use by the code's author), to using code in a way the code's author would never have thought of or intended. In the latter case, the code may need to be modified or extended in some way to fit the requirements of the new context, but the assumption is maintained that such modification or extension will result in less time and effort being spent than if a new solution had to be created from scratch.

3 A Brief Review of Software Visualisation

Software visualisation techniques have been developed for a variety of purposes (Ellershaw & Oudshoorn 1994). These purposes include use as pedagogical tools to teach Computer Science students how algorithms work (Byrne, Catrambone & Stasko 1999), use in visual debuggers to help correct bugs in software (Mukherjea & Stasko 1994), through to profiling large suites of applications to determine efficiency, correctness and help during maintenance.

The field of software visualisation spans research from algorithm animation (such as that demonstrated in the 1981 video "Sorting Out Sorting" shown at the SIGGRAPH conference of that year), to ways of "pretty printing" source code to make code blocks or keywords stand out more, or to make components (typically packages and classes) easier to browse. This latter approach is most commonly experienced by the majority of developers through familiarity with any reasonably modern software development environment.

Software visualisations are created from static and dynamic information. Static structures such as class descriptions, inheritance hierarchies and dependency hierarchies can be determined from analysis of the source or binary files. Dynamic information, such as method call sequences, field access/modification and multi-threading can be determined by analysis of the software's behaviour during execution (Moe & Carr 2001), (Reiss & Renieris 2001).

Many software visualisation tools have been created both in academic institutions and commercial enterprises. These tools use a number of different approaches to information retrieval and visualisation (Price, Small & Baecker 1992), and also offer a number of different approaches to the problem of allowing developers access to the creation and viewing process.

Some software visualisation tools allow for customisable visualisations, whereas others will create only one type of visualisation, or work with only one class of application (e.g. networking, sorting algorithms). Some tools require modification (called *instrumentation*) to the source code being visualised to extract dynamic information, whereas other tools can "spy" on applications executing (such as through debugger tools, or through modified execution environment) without requiring such instrumentation. Some tools allow developers to have highly interactive experiences with the visualisations and to manipulate the views presented (e.g. zoom, replay, focus, contrast two concurrent visualisations), whereas other tools only allow for straight forward viewing. Furthermore, some tools allow "real-time" live streaming of visualisations as the code executes, whereas other tools require all information to be gathered, filtered and parsed before it can be shown to the developer.

A selection of tools that highlight some of these features are Bloom (Reiss n.d.), XTango, Visor++ (Widjaja & Oudshoorn 1997) and Tarantula (Stasko n.d.).

4 Profiles of a Software Visualiser and a Code Reuser

The reason for using visualisation techniques is to further enhance understanding of software. In this respect code reusers are no different from anybody else who may have need to use software visualisations (referred to here as *software visualisers*). However there are different requirements for successful understanding depending on your motivation for being interested in the software.

4.1 Common Uses of Software Visualisation

Pedagogical-focused software visualisations for Computer Science students provide information on how the internals of an algorithm work (Byrne et al. 1999) (Naps, Bergin, Jimenez-Peris, McNally, Patino-Martinez, Proulx & Tarhio 1997) (Wiggins 1998), and may also show how certain language constructs work together. The aim of this is to teach the software visualiser how to program, and effectively how to create their own solutions (or versions of the algorithms).

Software visualisations created from large applications for the purposes of profiling, maintenance (Ball & Eick 1996) or determining correctness provide different information. Most notably, often these profiling tools work with very large data sets of static and dynamic information, and must use different graphing or abstracting techniques to show what is important in a way that won't overwhelm the software visualiser.

Research into software visualisations for understanding program traces does exist (Renieris & Reiss n.d.) (Jerding & Stasko 1994), but much of this is not focused specifically on reuse and the information required in that process, and rather mentions maintenance as the driving factor.

4.2 Differences for Code Reusers

Code reusers, while also software visualisers themselves as far as our work is concerned, do differ in some respects from the software visualisers in the first two common uses mentioned in section 4.1.

While code reusers are interested in what a component does, it will be of equal importance to understand *how* to use that component. Algorithm animations, while often showing the details of the algorithm, do not show which methods to invoke in the component to store information, or to start execution, or to extract key results. Code reusers can reasonably be expected to know the language they are developing in. Unlike students, they do not need to be shown how the component was written, but instead how the component is used. It may be more useful to a code reuser to see the order in which to invoke public methods in a Networking component's interface so as to set up a server socket, than to see the specific white box details as to how that server socket was set up.

Code reusers may also not have access to the underlying source code to a reusable component. While this can be a significant problem to a software visualiser doing profiling or maintenance (given the needs for possible corrections), a code reuser can still reuse the component as long as they have access to a compiled version for the architecture they are on.

A code reuser is also approaching the reusable component from the perspective of having it collaborate with other components that it was possibly not intended to be used with originally. This means that the component's external influences are important to visualise as well, something that is often not mentioned directly in other research projects in software visualisation.

4.3 Similarities for Code Reusers

Code reusers also share some common characteristics with other software visualisers. For example code reusers are, like software visualisers trying to understand whether a component matches its specification, interested in the side-effects and results of a component. Visualisations for this purpose describe what the component's specification is. While other software visualisers may be comparing these visualisation to a more thoroughly understood and concrete original specification (as laid down during an analysis/design phase), the code reuser will be comparing it against the requirements of their new context, and their own prepared specifications of the solution they need. While the material being compared with is different, a lot of the information and ways it can be displayed may be the same.

Likewise, code reusers may also be interested in class/package hierarchies and the methods invoked as a consequence of accessing the component's public interface. While other software visualisers profiling software for efficiencies and correctness are interested in seeing that the right method gets invoked, and that the correct number of invocations occur, code reusers are also interested in this information for the purpose of code reuse. One approach to code reuse mentioned in section 2 is to extend an existing solution through inheritance and/or overloading methods. Knowledge of when methods get called will enable a code reuser to better understand if a solution can be extended, and if so, which classes or methods need to be derived.

As well as this, the question of efficiency and resource usage may figure strongly in a component's suitability for reuse in the new context. These issues are also of importance to software visualisers doing profiling, although for slightly different reasons (i.e. correctness versus suitability).

4.4 Three Considerations for Code Reusers

Outside of the considerations of cost and of finding candidate components for reuse in the first place, we believe the main areas that a code reuser would be interested in – from a perspective of requiring understanding – are *what* the component does, *how* the component works and, if it does match the new requirements, then *how* the component can be reused and whether it needs to be modified.

5 An Architecture for Visualising Reusable Components

We are developing tools for creating and interacting with visualisations of reusable components. To create a context for our later discussions on what information we'd like to see in our visualisations, and how we'd like it to be organised, we will briefly discuss some of the key features of our visualisation tools. A simple overview of the architecture can be seen in figure 1.

5.1 Test Driver

As touched upon earlier at the end of our introduction to this paper, we are creating visualisations of developers' experiences with using a reusable component. Part of this requires that we provide a platform on which developers can *gain* that first-hand experience of using a component.

A component is often not an entire application in its own right. In our research, we are working with Java components at the class level. As many components cannot be executed in isolation (such as for the lack of a *public static void main* method in the case of Java components), we have developed a tool called *Test Driver* using the Java Reflection API. The test driver allows developers to specify a sequence of method invocation and field access/modifications, and to then execute this sequence on the component. This approach does not work for all components, with some graphical components requiring a more complicated setup before they can be used to interface with the developer. However for many non-graphical components this can be a useful method in its own right of gaining a preliminary understanding of using a component, and in what the results and side-effects of certain actions are.

5.2 SpyApp

The *SpyApp* tool is responsible for gathering much of the information required for the visualisations. While the test driver can gather some information, such as the sequence of instructions given to it by the developer and static class information, the SpyApp is responsible for gathering information such as method call traces within the component, resource usage, and most of the dynamic runtime information sought.

The SpyApp can do this while avoiding the need to instrument the reusable component source code, removing the need for the source code to be available. The static information gathered by the Test Driver can be done so from *.class* files, and by being able to gather dynamic information purely from executing *.class* files as well, we allow visualisation of reusable components where the source code has not been distributed.

SpyApp uses Java debugging libraries such as the Java Debugger Interface (JDI) to watch for events in the Java Virtual Machine. When these events occur, SpyApp then collects event information through calls to the JDI, and sends the collected information to the filesystem or a database as output. A similar approach using the native Java Virtual Machine Debugger Interface JVMDI library for Java has been used by (Reiss & Renieris 2000).

This output can then be filtered for relevancy, and the format of the information is the subject of our first wish-list mentioned at the beginning of this paper.

5.3 Transformer

The *Transformer* tool is responsible for converting the information gathered by SpyApp and Test Driver into a format that can be easily rendered by animation display tools.

We are interested in developing tools that allow for a customisable and extendible set of different visualisations, so the Transformer tool can have its conversion process configured to create these different visualisations. This configuration and conversion process does require some prior knowledge of the component, and its important features and uses (i.e. knowing what to focus on in the visualisation). This means that currently any configuration is better left to experienced users of that component who wish to create visualisations for other developers.

Transformer outputs the resulting information and stores it either in a database or directly in the filesys-

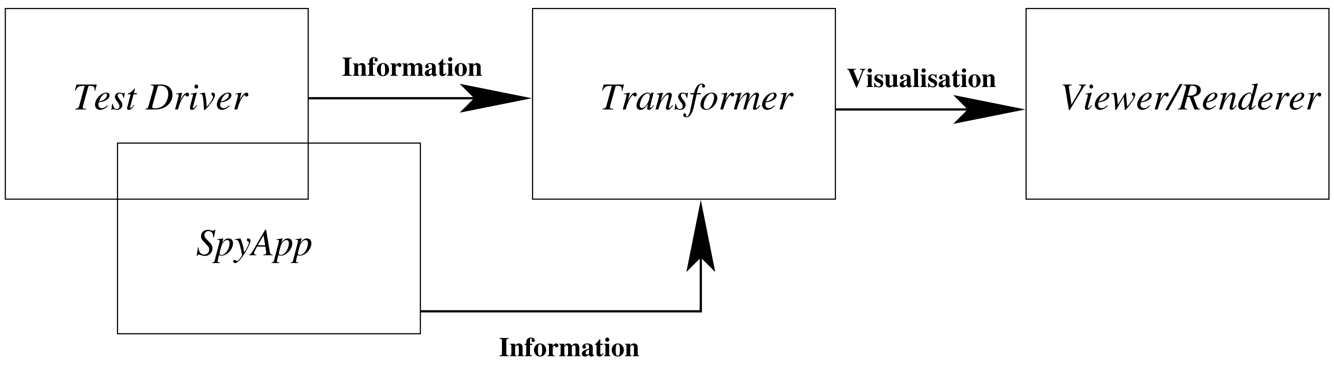

Figure 1: An architecture for visualising reusable components. The data outputted by the Test Driver and the SpyApp is the subject of this paper. We are interested in the information present in the data, and also how the data is handled.

tem. The format of the output is the subject of one of our wish lists and is now discussed in more depth.

6 The Wish list for Information To Be Visualised

We are interested in using visualisations to guide a developer's decision as to whether a component is reusable in the developer's current context. To create visualisations that are useful to developers, we must give some thought as to what should be shown. We have briefly outlined some of the intentions and characteristics of a code reuser in section 4. We will now discuss our wish list for what information we would like to see shown in software visualisations of reusable components. Not all the information would necessarily be presented in any one specific visualisation, however as we are looking at tools for creating customisable and extendible visualisations, this information should be available if required.

We have split this list into three sections, mirroring the different criteria that a code reuser may go through when deciding if a given component is appropriate for reuse, and if so, how it can be reused. These sections list the information we would like to see visualised to determine *what* a component does, *how* a component works, and *how* a component can be reused.

All of these categories could benefit from both static and dynamic visualisations. In many cases, animated visualisations could track the change in the various characteristics and performance measures during the course of execution. As some reusable components are accessed through a sequence of calls to the public interface rather than just a single call, animations could show how the different events and states evolve at key moments in the sequence.

6.1 What Does The Component Do?

To decide whether a potentially reusable component is useful in the new context, a developer must know *what* it is that the component does. We treat the component in this section as a black box, and are interested in the external side-effects and the results that occur as a consequence of interacting with the component's public interface.

We consider the following elements important in understanding what the component does:

- Author and/or user descriptions.
- Actual results and side-effects of a sequence of accesses to the public interface.
- Data sent or requested, external to the component.
- User input/output required during execution.

We shall now expand on these points in more depth.

6.1.1 Author/User Description

A component's author should know the specific details of what the component does. Any information they can provide, such as through traditional text-based documentation, will be of use in understanding what the component does. Similarly, regular users of the component will also have an understanding of the component's capabilities, and any advice or feedback they can give on the component can be useful when determining its appropriateness in the code reuser's new context.

Text based is the most common form of documentation currently available, and it should be complemented by visualisations, rather than be replaced entirely. Some visual techniques could be applied to the descriptions, to aid readability and understanding, but peoples' reports on their experiences of components remains a powerful way of sharing knowledge.

6.1.2 Results & Side-Effects

The results of executing certain sequences of method calls on a component's public interface, or the side-effects of these sequences on other components' data or the component's own state, will affect its applicability for reuse in a new context.

The results in some cases will determine whether the functionality provided by the component matches what is needed by the code reuser. The side-effects may help to show whether the component can work together with other components without compromising their functionality. They may also show whether a particular sequence of interactions leaves the component in a correct state for any future required sequences to also work. This latter point may be especially important if the component to be reused will be used multiple times. If this is the case, each interaction with the component would need to leave it in a usable state for the next interaction.

6.1.3 Sent/Requested Data

If a component is to be reused, then any information sent or requested by that component to such entities as a network, filesystem or database, needs to be handled in the new context. It is important that the code reuser understands the requirements and actions of the component with regards to the external environment.

This ensures that any components whose needs can not be met (either directly or through relatively minor modification) by the new context, can be discarded from the selection process.

This represents the results and side-effects external to the immediate application the component would be reused in, except for direct user input/output which is discussed in section 6.1.4.

6.1.4 User Input/Output

Should a component require interaction with a user to perform its functionality, then this needs to be understood by a code reuser if they are to make an informed decision as to its appropriateness in a new context. Specifically, the code reuser may need to know what information is required of the user, what information is given to the user, what the method of interaction is, and what environment (i.e. graphical, command line) is required for the interaction to take place.

6.2 How Does The Component Work?

We approached the question of *what* the component does by treating it as a black box. This may help in deciding whether the functionality available can be made to meet the requirements of the new context, but there is also the question of *how* the component works. This is important as the resource or permission requirements of operation may be prohibitive in the new context, and rule the component out as a candidate for reuse. There is also the possibility of extending or modifying the behaviour of the component to meet any new requirements. Understanding how the internals of the components work may open up opportunities for modifying its behaviour to what is required by replacing sub-components or overloading methods.

We now treat the component as a white box, and look at what internal information could be useful to visualise:

- Author and/or user descriptions.
- System permissions.
- Other software applications & libraries.
- Hardware resource usage.
- Execution traces.
- Multi-threading and synchronisation.
- Timing.

We shall now expand on these points in more depth.

6.2.1 Author/User Descriptions

Similar to the author/user descriptions mentioned in section 6.1.1, authors and users are well placed to impart valuable knowledge of how a component works. Visualisations aimed at promoting understanding how a component works should incorporate feedback from the author and users.

6.2.2 System Permissions

The system permissions required by a component affect its appropriateness for reuse in a given situation. Some environments may restrict permissions for security reasons, e.g. untrusted Java applets in browsers, and deny a component certain permissions. Describing what permissions a component requires allows the code reuser to make an informed decision regarding its usefulness. This ties in with section 6.1.3, where the *data* being sent/requested was identified as useful to visualise.

This could also identify what files are accessed or modified, or which servers and ports are accessed on the network. Other possible permissions could include such considerations as who the component must execute as, an example being a component that needs to execute as a superuser.

6.2.3 Other Software Applications & Libraries

Should a component require other software to fulfill its functionality, then visualising this information will enable a code reuser to better understand whether that component is appropriate for reuse. Visualising the information may help to identify the specifics of what software is required, why, and where. A code reuser can then investigate from a position of knowledge as to whether this other software is available and usable in the new context.

6.2.4 Hardware Resource Usage

The performance of a component may make it prohibitively expensive to reuse in a new context. If the new context requires that functionality be achieved within fixed parameters, such as in a certain time frame, or with less than a certain amount of CPU usage, or within certain boundaries of filesystem access or network traffic, then candidate reusable components should be measured against these criteria.

Visualising this information gives the code reuser a better understanding of the appropriateness of a code component within the restrictions placed by the new context.

6.2.5 Execution Traces

One possible approach to reuse is to overload certain methods of a component, or extend classes within the component, to modify the existing functionality to what is required. Visualising what methods get called, on what classes, and when, may give the code reuser a better understanding as to what methods or classes need modifying to change the behaviour. This relates primarily to the execution hidden by the public interface of the component.

Tracing the execution internal to a component involves capturing such information as method calls (Renieris & Reiss n.d.), method returns, field accesses, field modifications, object creation and object deletion. By visualising this information the code reuser can gain a better understanding of potential consequences and alternative executions that can be created by overloading or replacing certain parts of the component.

6.2.6 Multi-Threading & Synchronisation

Issues of threading, synchronisation, resource sharing and deadlock avoidance are important factors in deciding whether a component is reusable in a new context. While a component may reasonably be expected to work correctly by itself or within its original context, identifying its use of threads and any requirements or monopolising of resources may highlight problems in working with other components currently in the new context.

As well as this, a component's reliance on threads may make it unsuitable for a particular architecture because the architecture may either not support multi-threading or supports it in a fashion inconsistent with the model the component uses.

Information regarding this can be derived from analysing static source code, however dynamic information gained at runtime from viewing the threads and synchronisation can also be useful as it can show the sequencing of events and the amount of time spent holding or waiting on a resource.

Threading and synchronisation data may comprise a large amount of information. Using visualisation techniques to highlight the important parts of this information can help the code reuser make a more informed decision about the reusable component's ability to work in collaboration with other components.

6.2.7 Timing

While hardware usage may measure the component's performance with respect to its use of computer resources, the new context may also place other restrictions on what components may fit in it. One such restriction may be time, with the component needing to complete some operation within a certain time frame for the result to be useful. An example of this would be a component to be used in a real-time application.

Visualising the time line of execution can help a code reuser measure the component's performance against what is required, and against other potentially reusable components fulfilling the same functionality.

6.3 How Can The Component Be Reused?

When a developer has decided that what a component does (or can be easily modified to do) is what they need, and that how it does it is acceptable to them, they will still need to understand *how* to reuse it.

A simple example of this would be that we may know that a network component allows us to create network connections to servers, and that it does it through using the underlying native socket libraries, but we still need to know in which order to execute the various methods in the public interface to get the job done.

This touches on several of the wishes in section 6.2. However whereas that list stated what we wanted to know about the originally intended operation of the component, here we are more interested in how it can be used (or modified) in a new context that it may not specifically have been designed for.

The three categories of information we see as being important in understanding how to reuse a component are:

- Example uses of the component through it's public interface.
- Example extensions of the component through inheritance and overriding methods.
- Details of how to install any other software required by the component.

We shall now expand on these points in more depth.

6.3.1 Example Uses

Examples showing previous uses of the public interface of a component can show us how to link in the component to other code in the new context. This can also involve showing how to set up the state of objects that are required to be passed to the reusable code before it's functionality can be invoked.

6.3.2 Example Extensions

While reusing code may consist of simply plugging in the old code into a new project and interacting with it through the public interface, it may also involve extending the currently available functionality to match the new requirements. This approach could use a mechanism such as inheritance to extend certain classes or methods within the reusable component, to borrow what is already there and add on the extra capabilities that are needed. Example extensions could show which classes/methods can be extended, and the potential side effects of doing this. They could also show how extending classes whose objects are used as parameters or global variables by the component can modify the behaviour of the component, with the intention of meeting the required new functionality.

6.3.3 Installation Details

Another barrier to successful reuse is the time and effort involved in installing the component for use in the new project. For some components, it may be a comparatively easy task of downloading the component and including the file or files in the search path for the compiler or runtime environment (such as the class path/source path variables in Java). However other components may need more complex installation procedures. These could include recompilation for the local architecture, and downloading of other ancillary components that the reused component needs to work. Clearly available information on how to go about doing this would help to reduce the time and effort required in the reuse process. This is again an example where already available (and predominately text-based) documentation such as README files can be complimented and incorporated by visualisation techniques.

7 The Wish list for Transporting Information for Visualisation

Having decided what we want to see, we need to first gather the information from SpyApp and transfer it to the Transformer, so that it can be converted into a visualisation.

Some software visualisation researchers have designed their visualisation architectures so that the visualisation tool is built directly into the information gathering tool. In other cases, the visualised code is instrumented to include calls to the visualisation library. We wish to remove the tight coupling between the source and destination, by transferring the information in an independent, consistent format - so that SpyApp and the Transformer can be replaced and reused in different circumstances.

This will solve a common problem, where the captured information is abstracted too soon. The exercise of executing and exploring a component involves the expense of time and effort. If a particular visualisation (e.g. a sequence diagram) was created directly from this initial exploration, then if someone decided that they wanted another view on the component (e.g. an annotated description of the public interface, using colour and numbering to highlight important methods and access sequences), the second visualisation would either need a new exploration (and hence more time and effort), or be created from the first visualisation. Because a specific visualisation abstracts away information unnecessary in that visualisation, information relevant to other types of visualisation may be difficult to extract, or not even present.

We have the need for an intermediary format for the information gathered from a component, that can be used to generate a visualisation. Our wish list for this format is:

- Storable, and re-playable.
- Support live streaming to visualisation tools.
- Easily transportable over the Internet, using widely accepted protocols and standards.
- Filterable, so that relevant information can be extracted.
- Query-able for specific details within the overall information store.
- Programming-language independent.
- Platform-independent.
- Scalable to large components and long visualisations.

We shall now expand on these points in more depth.

7.1 Storable and Re-playable

Generating a program trace is costly, as it requires the software visualiser to spend time executing and exploring software components. At the time of exploration the software visualiser may not know what kinds of visualisations to view, or the need to view a different visualisation may become apparent at a point in the future.

It should be possible to store the SpyApp trace output on a filesystem or database, so that it can be replayed in the future to produce a different visualisation.

7.2 Live Streaming

Often, understanding software through visualisation is an explorative process, where a software visualiser will tinker with a component and view the changes that the tinkering makes to the visualisation. Therefore, the changes detected by the SpyApp may need to be transferred directly to the Transformer as they occur, rather than being sent at the close of execution.

Supporting live streaming may impose certain restrictions upon the representation of the data, forward references should be avoided, so that the Transformer always has a complete snapshot at any point in time.

7.3 Easily Transportable

As discussed in section 2, current visualisation research is trending towards the delivery of visualisations over the internet. This imposes certain restrictions on the format of the data, so a file format should be able to be transferred in a culture neutral representation, preferably text, and preferably be able to be fetched via a single HTTP request.

7.4 Filterable

Due to the potential volume of data that a program execution can produce, the data format should support an easy method of filtering, so that only relevant information need be passed to the transformer. As each transformer will have different criteria for relevance, the filtration must be sufficiently flexible to narrow the data to a sensible subset.

7.5 Query-able

The program trace information should be easily queried, so that a transformer can efficiently request subsets of information (e.g. “All types in this program”, “All methods that call method X”). It is not practical for a transformer to parse the trace each time such a request occurs, nor is it sensible for each transformer to convert the trace information into it's own database format.

7.6 Language Independence

As researchers, we work with a variety of programming languages. This research began using C++, and now uses Java. However there is sufficient similarity between many object-oriented languages (C++, Java, C#, VB.NET, Jade) that it makes sense for the trace format to represent the execution of any of these languages. Indeed, in an environment such as Microsoft's CLR, program development doesn't necessarily involve just one programming language.

Supporting multiple languages will also allow the Transformers and subsequent Visualisations to be reused as well, there is little difference between a C++ UML Sequence diagram and a Java one.

7.7 Platform Independence

Any architecture involving the internet, and platform independent languages such as Java, should transfer information in a platform-neutral manner. The trace information viewed on a Unix machine should be identical to that on an Apple Macintosh.

7.8 Scalable

Execution traces can get large very quickly, if there are a lot of method calls, object creation, or data changes. The format chosen will need to be easily used, filtered and queried, even when it scales up to hundreds of megabytes.

8 Summary

We wish to create visualisations specific to three aspects of reusable code components. These aspects are *what* does the component do, *how* does the component do it, and *how* can the component be reused. These visualisations would then be used to help foster understanding in developers as to how they could save time and effort through the process of reusing old code in new contexts.

To create visualisations we need to think about what information should be visualised. We also need to consider the details of extracting, storing and transporting this information. In this paper we have listed what we believe to be key categories for what information should be available to visualisations, and have discussed some of the characteristics of a data transport format.

While significant research has been conducted into creating software visualisations for understanding software, especially for pedagogical or profiling purposes, we believe the intentions of code reusers requires extra information for a complete understanding, and to aid in successful reuse.

Our aim is to further develop tools for gathering, storing, transporting and converting static and dynamic component information into useful visualisations.

References

Alonso, O. & Frakes, W. (2000), Visualization of reusable software assets, *in* 'Sixth International Conference on Software Reuse'.

Ball, T. & Eick, S. G. (1996), 'Software visualization in the large', *IEEE Computer* **29**(4), 33–43. *citeseer.nj.nec.com/ball

Byrne, M., Catrambone, R. & Stasko, J. (1999), 'Evaluating animations as student aids in learning computer algorithms'. *citeseer.nj.nec.com/byrne99evaluating.html

Ellershaw, S. & Oudshoorn, M. (1994), 'Program visualization - the state of the art'. *citeseer.nj.nec.com/ellershaw94program.html

Ferri, R., Pratiwadi, R., Rivera, L., Shakir, M., Snyder, J., Thomas, D., Chen, Y., Fowler, G., Krishnamurthy, B. & Vo, K. (1997), 'Software reuse metrics for an industrial project'. *citeseer.nj.nec.com/ferri97software.html

Frakes, W. & Terry, C. (1996), 'Software reuse: Metrics and models', *ACM Computing Surveys* **28**(2), 415–435. *citeseer.nj.nec.com/frakes96software.html

Jerding, D. F. & Stasko, J. T. (1994), Using visualization to foster object-oriented program understanding, Technical Report GIT-GVU-94-33, Atlanta, GA, USA. *citeseer.nj.nec.com/jerding94using.html

McIllroy, M. D. (1968), Mass produced software components, *in* P. Naur & B. Randell, eds, 'Report on a Conference of the NATO Science Committee', pp. 138–150.

Mili, H., Mili, F. & Mili, A. (1995), 'Reusing software: Issues and research directions', *Software Engineering* **21**(6), 528–562. *citeseer.nj.nec.com/mili95reusing.html

Moe, J. & Carr, D. A. (2001), Understanding distributed systems via execution trace data, *in* 'Proceedings of the Ninth International Workshop on Program Comprehension'. "citeseer.nj.nec.com/moe01understanding.html".

Mukherjea, S. & Stasko, J. T. (1994), 'Toward visual debugging: Integrating algorithm animation capabilities within a source level debugger', *ACM Transactions on Computer-Human Interaction* **1**(3), 215–244.

Naps, T., Bergin, J., Jimenez-Peris, R., McNally, M., Patino-Martinez, M., Proulx, V. & Tarhio, J. (1997), Using the www as the delivery mechanism for interactive, visualization-based instructional modules, *in* 'Proc. of ACM ITiCSE'97'. *citeseer.nj.nec.com/naps97using.html

Price, B. A., Small, I. S. & Baecker, R. M. (1992), A taxonomy of software visualization, *in* 'Proc. 25th Hawaii Int. Conf. System Sciences'. *citeseer.nj.nec.com/price92taxonomy.html

Reiss, S. P. (n.d.), 'Website - bloom', http://www.cs.brown.edu/ spr/.

Reiss, S. P. & Renieris, M. (2000), Generating java trace data, *in* 'Java Grande', pp. 71–77. *citeseer.nj.nec.com/reiss00generating.html

Reiss, S. P. & Renieris, M. (2001), Encoding program executions, *in* 'International Conference on Software Engineering', pp. 221–230. *citeseer.nj.nec.com/reiss01encoding.html

Renieris, M. & Reiss, S. P. (n.d.), ALMOST: Exploring program traces, pp. 70–77. *citeseer.nj.nec.com/renieris99almost.html

Stasko, J. T. (n.d.), 'Website - tarantula: Fault localization via visualization', http://www.cc.gatech.edu/aristotle/Tools/tarantula/.

Widjaja, H. & Oudshoorn, M. (1997), 'Concurrent object oriented programming — a visualisation challenge'. *citeseer.nj.nec.com/widjaja97concurrent.html

Wiggins, M. (1998), An overview of program visualization tools and systems, *in* 'ACM Southeast Regional Conference', ACM Press, pp. 194–200.

Wilde, N. (1994), 'Faster reuse and maintenance using software reconnaissance'. *citeseer.nj.nec.com/wilde94faster.html