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

School of Mathematical and Computing Sciences

Understanding Code For Reuse

by

Stuart Marshall

A thesis

submitted to the Victoria University of Wellington
in fulfilment of the requirements
for the degree of
Master of Science
in Computer Science

Victoria University of Wellington

November 3, 1999

Abstract

The software industry currently has problems meeting the high demand for software. One proposed solution is to reuse code from old projects, however this does not seem to be working. One reason could be the time and effort required to reuse. We identify the understanding of code as a key component of the time and effort spent during reuse, and look at tools that can help programmers to better understand code. Tools can utilise two approaches to support understanding: allowing the programmer to first hand experience of executing the code, and creating visualisations from these executions. We design, implement, and evaluate a tool that uses these two approaches.

VICTORIA UNIVERSITY OF WELLINGTON

Acknowledgements

There are many people to whom I am indebted for their help during the completion of this thesis, but none more so than my supervisors, Dr Robert Biddle and Dr Ewan Tempero. Their patience and guidance in the face of my initial, unique, interpretations of the principles of grammar, consistency and logic is much appreciated. Also, in the spirit of award ceremonies everywhere, I'd like to give thanks to my parents, for continually asking why I wasn't finished yet and greeting all my responses with polite disbelief, thus spurring me on to completion. My thanks also extend to the students and staff of the Department of Computer Science here at Victoria University, for a fun and enjoyable working environment during these past seventeen months. I'd also like to acknowledge the cast and crew of "Ferris Bueller's Day Off", for creating one of the best movies of the 1980's and for reminding me that " life moves pretty fast. If you don't stop and look around once in awhile, you could miss it", and also to the re-emergence of swing music, that helped to while away the evening hours.

Contents

Chapter 1

Introduction

Over the past few decades, computer technology has advanced at an impressive pace. One effect of this is that the population at large expects more and more from computer technology, and society has seen a significant increase in the use of computers throughout commercial, government and military sectors, as well as in home use. The result of this is that computer technology can now be found in nearly all the critical areas of society; maintaining health and legal systems, supporting and logging economic transactions, complimenting national defence, furthering scientific discovery, and underpinning the global communication networks, amongst other things. At the heart of computer technology is software, consequently there is a high demand for software that must be met by a limited supply of software engineers.

Development of large and complicated applications can take a long time even when developed by large corporations with vast amounts of resources to throw at the problem. In a competitive industry, meeting deadlines is a priority so as to avoid the risk of a product being released too late into a rapidly moving market, losing market share and being seen as yesterday's technology. Not only are programmers required to write high quality software in a world where errors are not well tolerated, and may even result in serious social or financial consequences, such as loss of life or business, but they must also maintain a high output.

Research in software engineering has looked at finding ways to help programmers write more code faster, either through improved processes and management, or through tool support for analysing, designing, implementing and testing. Examples of the latter include CASE tools that can automate some parts of the lifecycle, such as consistency checking in analysis and design diagrams, or the automatic translation of detailed design diagrams into source code. The use of these tools has achieved wide-spread acceptance throughout the software industry, yet the problem remains that supply is not keeping up with demand. These pro-

cesses, enhanced management practices and tools do not seem to offer a complete solution to the problem. For example, rigorous testing procedures help satisfy the quality issues but not the quantity issues, especially if an increase in the quantity of code leads to the time spent on testing regimes affecting commercial viability.

One answer to similar problems in other industries has been to reuse the products of past efforts, because it is typically easier, and cheaper, to recycle old ideas and products than come up with new ideas and products, as long as the old ideas and products properly serve the new purpose. Therefore, the answer to meeting the high demand for quality software may not be in finding faster ways of writing new code, but rather in findings way to reuse existing code. Without reusing the products and byproducts of past intellectual and physical endeavours, society would be in a constant cycle of reinvention. Unfortunately, computer programmers can often find themselves in this very cycle. Programmers frequently write code to implement functionality that has already been provided by existing code in past projects. A programmer need not be limited to only reusing their own existing code, but may be able to reuse another programmer's existing code. Writing new code does not necessarily improve on the work done by old code, and can often serve in merely duplicating the previous effort.

Reusing appropriate code can save effort in the analysis, design, implementation, testing and maintenance phases of a new project by building on the effort put into these phases in past projects. This saving of effort may then translate into reduced time and cost, that in turn may help the software industry to better meet the demands place on it.

As well as saving effort in the software lifecycle, reusing code also acts as a way of passing on knowledge and experience from the top programmers of one generation to the programmers of the next generation [Mey97]. If quality code implemented by good programmers can be reused in appropriate places, then the new software can be built around tried and true solutions and there are fewer opportunities to introduce faults. Programmers might also learn good programming techniques by reusing and seeing code from respected practitioners of the past. This is similar to how people often learn in other professions, by studying the work of those who have gone before them.

Code reuse does not appear to be prevalent in software engineering practice and therefore seems to have failed. There have been several reasons advanced by others to explain this. Firstly, current software engineering management practices do not seem to encourage or support reuse. Secondly, there is the issue of the amount of code that is made available to reusers. Much of the code written by a company remains within that company and is

kept out of the public domain. This reduces the quantity of code that a reuser can search through and reduces the probability of finding code that could be successfully reused in a new context. Thirdly, there is the issue of quality. The available code, while there for reuse, may not be reusable in its current form due to its design or implementation. Assumptions that it makes regarding the original context might make it difficult to fit into the new context, even though it achieves the desired functionality. There should be a heightened emphasis on writing reusable code for tomorrow's reusers. Fourthly, reuse can, while trying to save time and effort, itself require non-trivial amounts of time and effort.

There are several reasons why reuse can be costly. Firstly, a programmer must expend effort in searching for viable code for reuse. Secondly, when code has been selected it may require some modification to fit into the new context. Even without modification, there may be some work required to import the reused code into the system under development. Both of these tasks require that the programmer have formed a good understanding of the code. Searching for viable code requires that the programmer be able to identify what code is relevant to the functionality required for the new project. Given the potentially large quantities of reusable code that could be available, the ability to quickly separate the relevant from irrelevant requires the programmer to quickly understand what a particular piece of code does, how it does it, and how it can be reused. Once this understanding has been gained, the programmer is in a better position to determine that particular code's applicability. This understanding also helps a reuser know how to reuse the code, in terms of what parts of the code to call, in what order different parts should be called, and what data, such as parameters, that the code needs.

If reuse requires a reuser to understand code, then this raises the question as to how code is understood. Programmers typically understand code by referring to a variety of sources. The majority of these sources fall into one of the following three categories:

- programmer documentation
- previous uses of the code
- hands on control of the code

Documentation is a byproduct of the software lifecycle and is well established in contempary software engineering practices. The types of documentation that are of interest in the code reuse process are those documents created for software engineers to help them understand the analysis, design or implementation, rather than the user manuals distributed with

software applications. Documentation for software engineers is well established in practice because it is the primary input and output of phases in the widely used models of the software lifecycle. Documentation will usually describe the project at all the major phases, from analysis through to implementation and testing. Documentation is created in the analysis phase to be used in the design phase, and so on down the line. This documentation exists both as something internal to a project prior to completion, and also for future maintainers to use as a source to understand how the project works. In its crudest form, this documentation is the source code. More usefully it can be a text description or come in the form of modelling diagrams, such as flowcharts or UML class diagrams. These are not necessarily created with reuse in mind, but just as with reusing code for unforeseen situations, documentation can be utilised for new purposes. The documentation can give a programmer a better understanding of what a piece of code does, how it does it, and how it is invoked. With this knowledge, the programmer can then determine if a piece of code implements functionality required in a new project, and if so, then whether it can be reused, and how much effort might be involved in hooking the reused code into the new project. In short, the documentation is supporting understanding for code reuse.

Lately, with an increase in the power of workstations, people are looking at using visualisations to document and understand projects. These visualisations tend to make better use of the human capability for visual interpretation than plain text or ordinary source code, and better use of visual interpretation can lead to more information being conveyed in an easier to understand manner. Visualisation research is looking at a variety of techniques, such as pretty printing source code to make it more readable and navigable, to dynamic animations of the code executing at some level of abstraction. An example of the former is *Javadoc*, that takes Java API source code and creates HTML web pages for all of the classes, interfaces, methods, constructors and fields. Javadoc[Jav] is used to create the web documentation pages that come with the standard Java Development Kit (JDK) distributions.

A programmer's understanding of a piece of code can be also be enhanced by looking at previous uses of the code. Two common forms of such uses are pedagogical examples and *real use* examples. Pedagogical examples are those written by the original programming team to demonstrate their code in action. An example of this is the *Swingset* application that demonstrates Java's Swing API [Sun]. Swingset is a demo program bundled with recent releases of the JDK that shows the capabilities of each of the classes in the Swing API. The application itself aims to achieve no task other than to demonstrate the new types of

interaction and display possible in the Swing API. The code for Swingset is also available and a reuser can look at how the code relates to what appears on screen, thereby gaining an understanding of the Swing API not just by seeing what it can produce, but also by seeing how it produces it. Real use examples could come from previous projects where the code has been used to solve some problem or fulfill some functionality. Seeing how code worked and fitted in previous situations can give the reuser an appreciation for whether that code can be reused in the new situation, and if so, how.

Another approach is to have hands-on control of the code. This implies that the reuser tries to get first hand experience of what it does and how it's used. This is often achieved by writing some small piece of code that drives the code to be reused, invoking its functionality and exploring the results and effects of different actions. We call this activity *test driving* and refer to the fragments of code written for test driving as *test harnesses.* Test driving code is analogous to how a prospective buyer would test drive a car to ensure it met their requirements before actually purchasing it. Code is test driven to ensure it performs the desired tasks, returns the desired results, and possibly determine the speed and memory requirements, as well as how much effort would be required in successfully hooking it in with the other parts of the new project.

If reuse is failing then how can we turn this around. In other areas of software engineering there has been significant work into creating tool support with the aim of easing the technical burden. Prominent examples of this support is the existence of CASE tools and the advancement in programming environments, integrating software engineers' tools and automating some tasks. Although not necessarily offering a complete solution to the problem of why people do not reuse as much as they could, having worked in other areas of software engineering, tool support should be able to promote the application of reuse principles. Tools could support all parts of the code reuse process, including the finding of code to reuse, the understanding of found code so as to judge its applicability to the new project, and then inclusion of that code into a new project. None of these tasks are trivial. Firstly, finding code to reuse may involve searching through large collections, and tools could automate some of this process, similar to the principles that search engines work on. Secondly, understanding other programmers' code can be difficult due to such things as personal coding stylisms, the latitude that programmers have in implementation of essentially similar solutions, and some of the failings of standard forms of documentation. Tools could help focus attention on the relevant parts of reusable code, and support programmers in gaining hands-on experience of

the code's execution. Thirdly, inserting a piece of reused code may require modification of the reused code, that in turn can lead to problems. Tools could, amongst other things, help ensure consistency between the reused code and the new code in a project. Overall, tool support could make it easier and more desirable to reuse code, by reducing the associated time and effort.

To re-cap, code reuse can help to solve the problems of meeting the overwhelming demand for software. That it is failing to do this is in part due to the fact that the cost of reuse is high. Anything that brings this cost down is therefore good for reuse, and one of the costs of reuse is in understanding potentially reusable code. This understanding is usually gained through experience of its use, or by referring to documentation. Promoting understanding of code by allowing reusers to gain experience in using it, or by allowing them to create documentation for it, would seem a good way to reduce its cost. Reducing the cost of understanding should lead to a reduction in the cost of code reuse, and therefore lead to better and more wide-ranging applications of its principles.

The thesis of this research is that the understanding of code may be promoted in two ways. Firstly, by affording the programmer the opportunity to test drive a code so as to see what it does. Secondly, by documenting the runtime behaviour of code using animations. Tool support for both of these tasks is required, and the development of such a tool has been undertaken as part of this research. The tool will be a proof of concept and will be measured against requirements developed from the research in this thesis.

Chapter 2

Reusing Code

The introduction to this thesis argues that reusing existing code may solve or at least alleviate the supply problems faced by the software industry. These supply problems have risen from demands from an increasingly computer-aware, computer-literate and computer-dependent society. This chapter will introduce code reuse in more depth. We discuss a model for reuse developed by Biddle and Tempero[BT96], and use it and the accompanying terminology to formalise our discussions of reuse in this and later chapters. Although we claim in chapter 1 that code reuse is failing at the present time, it is at least being attempted and we look at some of the ways that code reuse already exists in software engineering and is supported in the languages and tools that programmers can use. We outline some of the steps typically undertaken during the reuse process and identify understanding as a key player in these steps, leading onto chapter 3 where there will be further discussion on understanding for reuse.

2.1 Reuse in Society

Reuse is the application of existing concepts or things in the creation of new concepts or things. Often the concept or thing to be reused would not have been originally thought of or designed with the new purpose in mind. Reuse has been a common activity throughout the course of human events. Industry reuses processes (defined sequences of activities) from old product lines and services to aid in the manufacturing of new kinds of goods or in the delivery of new services. It can be cheaper to recycle than to buy new resources or finance new research. One need look no further than the entertainment industry and Hollywood to see an entire industry repeatedly reusing the same plots and characters to great success. Reasons for reuse are not restricted solely to commercial motivations either. To alleviate

environmental concerns, society is looking to reuse products and by-products so as to protect dwindling resources, or to reduce the amount of waste that would otherwise need to be disposed of. Reuse is pervasive throughout society, and in a world of finite resources, finite research funding and finite labour, reuse looks likely to continue being pervasive for some time to come.

2.1.1 Software Reuse

Reuse has been present in the relatively young field of Computer Science from the beginning, although often in a somewhat unsophisticated manner that hasn't seen its full potential realised. Nearly all languages currently in use support the logical grouping of code into blocks, even if the terminology differs from language to language. This has resulted in much work into, amongst other things, functional decomposition and object oriented (OO) design, whereby a project is split up into logical sections. A section can be reused if its functionality is required in more than one place in the overall design. Intra-project code reuse is reasonably well established in programmers' psyches, however the same cannot be said of inter-project code reuse.

The general practice of reuse in the software engineering lifecycle is known as *software reuse*, a term that is further defined by a number of practitioners. For example, Biggerstaff and Perlis write:

> "Software reuse is the reapplication of a variety of kinds of knowledge about one system to another system in order to reduce the effort of development and maintenance of that other system" [BP89]

Software reuse can itself be categorised, commonly as *building block reuse* and *generative reuse* [MMM95]. Building block reuse is the reuse of products created during the software lifecycle, such as code or documentation. Generative reuse is the reuse of the processes involved in the software lifecycle, whereby the term process we mean a sequence of activities undertaken by software engineers rather than a process running on a CPU. Mili, Mili and Mili refer to both products and processes as *reusable assets* [MMM95].

All kinds of assets can be useful to a reuser. Documentation from an old project can be applied to a new and similar project, perhaps with some modification. This can save on the time required to write documentation for the new project, time that could be spent elsewhere or on moving the completion date forward. Similarly, reusing processes can result in a better understanding of how these processes can be done more efficiently, causing a streamlining

effect that further reduces the development time and improves productivity. The specific area our research concentrates on is the reuse of source code.

2.1.2 Code Reuse

Code reuse implies taking code written for some previous purpose and applying it in a new situation. The code may or may not have been originally designed with the intention of being reused and the code may require some small amount of fine-tuning to get it to fit. Some practitioners [Tra95] require that code reuse be the reuse of code specifically designed with future reuse in mind. We do not draw this distinction.

Code reuse is not new. Most programmers have copy and pasted code, or invoked a block of code from several locations, whether those blocks are called procedures, methods, messages, functions or something else. Doing so avoids the need to rewrite the code every time its functionality is needed. Successful, widely-used languages tend to build up standard collections of code written by the language's practitioners or creators. One of the advantages to using Java or C/C++ is the extensive libraries of code that come with their standard distributions. This spares the programmer from, amongst other things, having to build their own hashtables or vectors, window structures or buttons. Java's standard packages and C++'s Standard Template Library serve as a source of existing code for users of the language to reuse.

We coin the term *reuser* to refer to someone who uses existing code in solving a new problem.

2.2 Reuse vs Reusability

The research contained within this thesis is aimed at helping someone to reuse code once that code has been written. Another line of research closely tied to this looks at helping a programmer to write code that may be reused at a later stage. The latter line of research is looking at the *reusability* of code.

When code is reused, it is often reused in situations that would not have been thought of at the time of the code's design and implementation. Yet the quality and nature of the design and implementation affect how much work is required to reuse the code in a new situation, or if it can be reused at all. Using reusability principles during design and implementation can help to increase the codes applicability to new situations, and reduce the amount of effort needed to import it into these new situations. This in turn makes code reuse easier and

more attractive. Reusability therefore supports reuse, and reuse acts as the main motivation for reusability. The benefit of writing reusable code is lost if reusers do not know how to reuse the resulting code, or do not reuse it as frequently as is possible. Both lines of research compliment and rely on each other.

The tool support we look at in this research does not actively support reusability. During the process of reusing code, a reuser may be able to identify certain characteristics that make code more reusable. They could then try to incorporate these characteristics into their own code, however this would rely on the reuser gathering implicit information from the structure of the reused code, and is not explicitly supported.

2.3 Benefits of Code Reuse

We now look at some of the benefits software engineers gain by reusing code. Later in this chapter, we will discuss some of the obstacles that are preventing code reuse from becoming widespread, therefore limiting the impact of these benefits.

2.3.1 Reduced Development Time

Reusing code reduces the amount of time required during development by removing the time spent designing, writing and testing code for functionality that is served by code already written.

Designing and implementing take up a significant part of a development team's time. If past design work immortalised in code can be reused then the creative time needed to come up with the same or a new, but unimproved, solution can be better spent elsewhere. The act of reusing code also saves time that would otherwise need to be spent transcribing the design into source code.

It is rare for new code of any substantial size to be free of faults even when written by expert programmers. Time spent on debugging code can be a substantial portion of the overall development process. Existing code will most likely have already been through the testing process so assuming this testing process was carried out properly, then the reuser should require less time in the testing phase or the new project. Frequently reused code can offer some guarantees of quality to a reuser. If the code were not of a high quality then it is unlikely that it would have gained any popularity in performing functionality for previous reusers. Any faults that may have existed are likely to have been found and ironed out by previous reusers. The reused code will still need to be tested with regard to its interactions

with other code in the system, but one level of testing has been removed.

2.3.2 Promoting Easier Understanding

A programmer's understanding of what and how a system works can be helped by code reuse. This comes from the following observations. Firstly, a programmer may already be familiar with an assembly being reused. The programmer may have first hand experience of using it, or have seen it being used in other projects. Secondly, the use and reuse of the same assembly better highlights areas of commonality within or between projects. It is easier to see that the same functionality is met in ten different locations if the same assembly is referenced in those ten places than if there were ten slightly different implementations that all did the same thing. With these two observations in mind, the programmer may borrow from understanding of past projects in their understanding of the current project. A programmer, seeing an assembly previously encountered and understood, gets a memory cue that may give an insight into understanding the overall design and purpose.

2.3.3 Promoting Easier Maintenance

Code reuse can have benefits beyond the design, implementation and testing stages in the software lifecycle. Time and effort may be saved in the maintenance stage if functionality common to multiple places in an application is handled by the same assembly. Maintenance can be split into three types: corrective, adaptive and perfective [Pre92].

Corrective maintenance is the correction of faults in the code. If assemblies are reused then there is less actual code required in the application, and less code should lead to a lower number of faults. Adaptive maintenance is the updating of code to meet with changes to the working environment. A reused assembly can be modified once and the changes are automatically updated in all the context in that it is invoked. Logic dictates that changing ten implementations of the same algorithm to do something slightly differently would be more time consuming and error-prone than changing just one implementation reused ten times. It is also easier to find one implementation than ten. This argument also applies to perfective maintenance, where the application is modified to meet new demands and new ideas.

An example of adaptive and perfective maintenance would be an application that needs to use a sorting algorithm in several places throughout the code. At the moment it uses the quicksort algorithm. It is then decided to change from the quicksort algorithm to the mergesort algorithm. If only one assembly is used for all parts of the code that need a

sorting algorithm, then this task would involve changing a quicksort assembly to a mergesort assembly with possibly some extra work to cover any changes in the sorting algorithm's interface. However, if the sorting was done by different implementations of quicksort, then these quicksort algorithms must be found, a possibly non-trivial task that could see the maintainer miss an instance, and then the mergesort algorithm must be implemented multiple times. There is a heightened chance of making a mistake implementing something ten times than there is when doing it just once, and it would certainly take more time and effort in terms of searching for the different quicksort implementations, writing multiple mergesort algorithms and debugging the same.

2.4 Modelling Reuse

In discussing code reuse it is useful to have some form of conceptual model to act as a framework of classifying and predicting advances in the field and also fix the vocabulary we use. Biddle and Tempero[BT96] developed a conceptual model based around the dependencies that may exist in fragments of code.

2.4.1 Assemblies and Dependencies

The model created by Biddle and Tempero is based around looking at the connections between pieces of code. These connections are called *dependencies*. A unit of code is called an *assembly*, and is also considered to be the unit of reuse in this model. Code fragments of varying sizes qualify as assemblies, from simple methods through to entire subsystems. Dependencies exist in and between assemblies.

The dependencies that exist between assemblies can take a variety of forms. An assembly may require parameters be passed through to them that fulfill some type requirement or they may rely on global variables being set to a value within a valid range. Another assembly may depend on a value being modified by an assembly it uses in a meaningful way. All of these dependencies restrict the situations where the assembly may be reused.

We will use the term assembly as the unit of reuse in subsequent chapters.

2.4.2 Context and Components

The model splits code reuse into two varieties, context reuse and component reuse. Figures 2.1 and 2.2 show context reuse and component reuse respectively. Contexts and components are both uses of an assembly, and an assembly can be both a context and component at the same

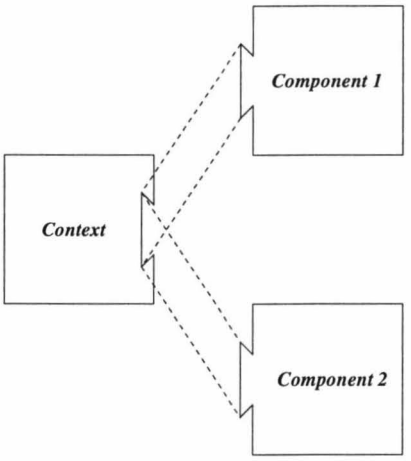

Figure 2.1: Context Reuse.

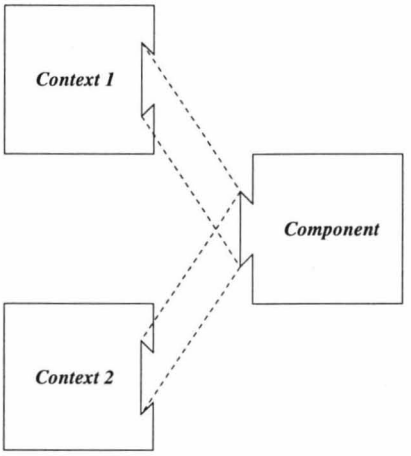

Figure 2.2: Component Reuse.

time. A context uses a component to perform some required functionality. That same context may itself be performing some functionality (and is using the previously mentioned component to fulfill some part of this) for another assembly further up the hierarchical structure of the program. The most familiar form of reuse is that of component reuse. Using a method in multiple places counts as component reuse. Using a class to typify and instantiate multiple objects is another.

We will use the terms component reuse and context reuse in later discussions on the requirements and evaluation of our tool.

2.5 Languages and Code Reuse

Most languages have some form of support for code reuse. The use of procedures and functions is a form of code reuse, as is macro expansion. Functionality can be tied up in one of these

constructs and then parameters passed through to allow some a degree of fine-tuning to its behaviour to match the current context it is being used in.

OO models the code based on the concepts in the domain that is is supporting. This is touted as being natural and easier to understand, both to the programmer and to any other people who may have a role to play in software development (clients for example). Whether these claims by OO proponents are accurate or inaccurate is not the focus of this research, however we will at this point summarise some of the features that support reuse and that are present in many languages subscribing to the OO philosophy. This is not intended to be an exhaustive list of features. The terms discussed in this section will be used in subsequent chapters and we will also discuss how some of these features can aid or prohibit understanding, and their role in tool support.

2.5.1 Class

The concept of the class is supported in most languages within the OO paradigm. A class is set of objects that share a common type of data structure and a common set of operations that can be performed on that data structure. These classes have an interface and implementation. Programmers create individual objects (referred to in most of the literature as *instances*) by instantiating the class. At this point, the specification typified by the class is being reused. Any modification to the algorithms in the class, or to the underlying data structure, will automatically apply to all the objects that belong to that class. During the course of this thesis, we will design and implement a tool that will use the class as the limiting size of the permitted assemblies.

2.5.2 Encapsulation

Encapsulation aids reuse by reducing the number of dependencies that an assembly may have with regards to another assembly. A properly encapsulated class will limit all outside interaction to having to go through the interface. By disallowing an external assembly to have direct access to the internal implementation of the class, the external assembly can not rely on features of that implementation. For example, it is common for an encapsulated class to prohibit external assemblies from accessing the data structure directly and forcing them to go through interface methods instead. This is referred to in the literature as *information hiding.* This frees up the implementation to be modified for reasons of maintenance and extendibility, and as long as such modifications do not affect the interface then assemblies using this class

can continue to use it. The contracted functionality between the reuser and the reused asset should not be broken by the modifications, otherwise it may become infeasible to continue using the asset. However, this last matter is unaffected by whether or not encapsulation is employed.

2.5.3 Composition

Composition is the creation of assemblies out of other assemblies. In effect, an assembly is being reused as part of a larger assembly, removing the need to duplicate functionality in the larger assembly that is already supplied by the smaller assembly. An example of this would be a class that has instances of other classes as data members. Composition is a common form of reuse. Building systems out of subsystems is a form of composition and fits in well with the real-world analogy that "things are made out of other things".

2.5.4 The Effect of Inheritance & Polymorphism on Understanding

Inheritance and polymorphism are features of most OO languages. Inheritance allows a new class to be created by having it extend and reuse the features of an old class, including both the interface and implementation. The new class offers different behaviour from the old class by extending some of the behaviour of the old class or over-riding it completely. An important benefit is gained from inheritance: in most OO languages that support inheritance, the interface of the newly created class will *conform* with the interface for the old class. This means that the new class can be used wherever the old class may have been used, and in this manner the context that the old class has been used in can be reused for the new class. This is an example of context reuse, that we introduced in section 2.4.2.

Conformance of interfaces for different classes leads to polymorphism. Here, a context can be reused by implementing with respect to a common interface, by using runtime substitution of classes with the same interface but different implementations. While this aids context reuse, it does have an effect on a programmer's ability to understand code. Inheritance and polymorphism can obscure exactly what implementation is being referred to at a particular point in the code [JS94a]. This can lead to the programmer being confused over exactly what the code does at a certain point. It may be impossible to decipher by looking at source code which implementation is chosen to run as this would be be decided at runtime.

2.6 Tool Support

Tool support already exists for code reuse. This section will cover some of these tools to give an appreciation of where our tool will fit. This is not an exhaustive list of possible tool support for code reuse, however they are a reasonably representative spread of the major tasks of the code reuse process, i.e. searching, incorporating and understanding.

2.6.1 Code Repositories and Search Engines

Reusers need to find useful assemblies. An approach is to group reusable assemblies together. This can facilitate easier searching, and such pools of reusable assemblies are called *code repositories*. Reusers can search through these repositories, and also store their reusable code for future use by others. Such pools would need search engines as the volume of code would be too great for reusers to sort alone the relevant from the irrelevant. Without search engines, the amount of time required for finding useful assemblies could overtake that required to design, implement and test new code. Search engines would require some specification of the problem from the reuser and would need to intelligently match them against specifications submitted with stored assemblies. The search engines could then return any matches for further inspection by the reuser. Refining the pool of useful assemblies returned by the search engine requires a reuser's understanding of these assemblies. This understanding could be supported by tools, such as the tool that we develop later on in this thesis.

2.6.2 Smart Cut'n'Paste

Even when candidate assemblies are found for a problem, the reuser may need to modify the code to fit into the new situation. For example, when cutting and pasting code it is often necessary to change variable names, and these kind of problems could exist on a greater scale. The assembly may offer an algorithm that may differ slightly from what is required. Variable names may differ or parameters may be in a different order, so the reuser needs to do some fine-tuning. Tool support for this kind of modification would be desirable to reduce the chance for errors being introduced that would further waster reusers time.

2.6.3 Assembly Documentation

Understanding code is important to its reuse. Static documentation of code can be informative and useful, however it is limited in how it can properly express the dynamic nature of runtime behaviour. A user can gain a better understanding of dynamic runtime behaviour by

using dynamic documentation. Tool support to better document assemblies can give reusers a better idea of what issues lie behind the reuse of a particular assembly and what that assembly may need to be reused. Documentation may also lead to identifying what parts may need modification and in what ways.

We discuss dynamic documentation as a form of tool support more closely in chapters 3 and 4, and this will form a major component of the tool support we develop in later chapters.

2.7 Summary

In this chapter, we have introduced code reuse as one approach to reducing the overall time and effort needed to develop software. We have outlined the benefits of code reuse, as well as some of the obstacles that are currently blocking successful code reuse. A model for reuse, created by Biddle and Tempero, has been introduced and will be used in subsequent chapters for formalising our discussions. Lastly, some of the features of modern OO languages that support code reuse have been identified, and we have listed a small selection of possible tools that could support code reuse.

Chapter 3

Supporting Understanding for Reuse

In the introduction to this thesis, we claim that code reuse can help software engineers meet the demands made on them. We also claim that code reuse is failing at the present time, and one of the reasons for this failure is the high cost of code reuse. If understanding assemblies is a key part of successful code reuse, then anything that promotes understanding and helps to reduce its cost, should be good for code reuse. In this chapter we further explore the role understanding plays in code reuse and look at ways of helping the reuser to understand.

3.1 Reuse Requires Understanding

To reuse an assembly, the reuser must have an understanding of what it does and how it does it. Without this understanding the reuser will neither be able to determine whether it is applicable to a new situation, nor how to modify it if modification is necessary, nor how to use it properly.

We split the reuse process into a *search phase* and *inclusion phase*; understanding assemblies is relevant in both. During the search phase, understanding is important so as to be able separate suitable assemblies from unsuitable assemblies. This requirement for understanding can lead to problems if the available code repositories are large, as the time and effort required to understand large quantities of assemblies will erode some of the benefits of code reuse.

Search engines can be employed to help with finding suitable assemblies. The reuser can supply the search engine with some form of specification of what they need. The search

engine can then match the reuser-supplied specification against specifications stored with the individual assemblies. The specification language can be plain natural language text or something more formal. The result of this is that the search engine is able to pick out a smaller pool of more relevant assemblies. This reduces the amount of assemblies a reuser must understand, but it does not completely remove the need to understand.

Whether or not a search engine is employed, a reuser is typically left with the task of picking the most appropriate of the plausible assemblies. This selection would be made on the grounds of performance, conformance, or some other this issue. To choose the most appropriate assembly the reuser should have an understanding of what the assemblies do and how they work. The tool we discuss and implement in this thesis serves to support understanding of what assemblies do and how they work.

The need for understanding does not stop at the end of the search phase. The assembly selected for reuse might require some modifications or fine-tuning for it to fit into the new project. Safely modifying an assembly's interface or implementation requires an understanding of the current interface and implementation. Without an understanding of the assembly, the reuser may introduce faults and inconsistencies and create problems further down the line. If the assembly does not need modification then there may still be work to be done before it can be reused. The assembly may need some form of configuring to be used in certain situations (parameters be passed through, variables be set, methods invoked in certain orders, and so on and so forth) and again, this requires understanding on the part of the user of how that assembly works.

3.2 Understanding Assemblies via Test Driving

One of the best ways to understand something is to use it. Children learn about the environment around them by playing. Programmers often learn programming languages not just by sitting down with a book and reading for a long time but rather by writing programs in the new language, trying out the features and seeing the effects of certain actions.

Programmers often attempt to play with assemblies in some form or another to properly comprehend what they do. It is our experience of using the Java Layout Manager *GridBagLayout* from the Java Abstract Window Toolkit (AWT) that writing exploration code can be beneficial. The exploration code uses GridBagLayout in some way and then the user manipulates the new code to see how different methods and arguments have different results. This approach compliments, rather than replaces, the act of reading the documentation sup-

plied for GridBagLayout. Often the code used to test GridBagLayout is not part of the final product.

Part of this thesis is that helping a reuser to gain first hand experience of an assembly can further promote their understanding of that assembly.

3.2.1 Test Driving Assemblies

Reusers can gain an understanding of an assembly by using it. When a reuser uses an assembly for the express purpose of understanding what it does and how it does it, then the reuser is said to have *test driven* the assembly.

In the context of this thesis, we use the term test drive to mean something similar to what we would expect when test driving a car. The aim of test driving a car is not to reach a particular location, rather the car is being checked to make sure it fulfils the driver's requirements. Having met these requirements, the car can later be successfully used to take the driver places. Likewise with test driving assemblies, the goal is not to solve some problem with the assembly, but to make sure it fulfils the reuser's requirements. Having met these requirements, the assembly can later be successfully used to solve problems. Borrowing terms from the OO languages mentioned in chapter 2, test driving an assembly might involve reading a public interface, instantiation of a class, invoking methods, and access and modification of state. These activities allow us to see what is available, what the effects of actions are, how a sequences of actions can affect later results and also how good the performance of the assembly is. Performance can be an important criterion for successful reuse and, unless explicitly mentioned, speed and memory requirements can often be hard to judge from the usual forms of documentation or source code alone.

3.2.2 Test Harnesses

Programmers often write fragments of code for test driving a method, class, subsystem or system. In the introduction to this thesis, we coined the term *test harness* to describe these code fragments. Test harnesses can serve a dual purpose. Firstly, they can be used in the testing phase of the software lifecycle. Here they would serve to check out that there are no faults that need to be ironed out and typically the tester would already have a good understanding of the code. Secondly, they can be used to determine what the code does, by checking the results and side-effects. This thesis primarily concerns itself with the latter use of test harnesses, although checking for faults in assemblies can also be important to reusers.

Writing test harness code can be a tedious and time-consuming exercise, and the code is not typically included in the final project, further reducing their appeal. Test harnesses, like any other source code, may have faults in them that result in the assembly being incorrectly used. Such faults may go undetected and any irregularities in the results may be erroneously attributed to the assembly. This would in turn cause a reuser to misunderstand the assembly and negate the purpose of having the test harness. Tool support for test driving assemblies would remove the need for manually writing test harnesses. Assemblies could be plugged in and played with via the tool's interface. This reduces the cost of understanding to the reuser by reducing the time and effort required. As long as the tool offered the same range of interaction with an assembly that a test harness would do then the reuser would still gain the same degree of understanding.

3.3 Understanding Assemblies via Documentation

If the reuser cannot test drive an assembly, then they may still be able to form an understanding by referring to any accompanying documentation. Documentation compliments understanding even when test driving is possible, the use of one should not limit the use of the other. Documentation can come in different forms, at different levels of effectiveness and abstractness.

Source code is the crudest form of documentation. A reuser trying to understand an assembly by reading the source code would have to contend with the original programmer's potentially different stylisms. This can easily lead to confusion and misunderstanding. This in turn can lead to a misapplication of an unsuitable assembly or the missing of an opportunity for reuse.

Tool support for pretty printing source code does exist and can help to alleviate some of the concerns. The execution path of an assembly can be better traced by using code browsers that facilitate navigation through code. These tools can help the reuser to quickly find referenced blocks of code without having to manually search potentially large tracts of code.

Internal documentation (i.e. commenting) describes the code at a higher level of abstraction. It can give an overview of the algorithms used, or state the the purpose of the code, or more clearly outline the inputs and outputs. Internal documentation can be complimented by the external documentation that normally results from the software lifecycle. Analysis and design documents help to focus the reuser on the important features of the assembly.

Unfortunately these latter forms of documentation are not always available to a reuser, and even if they are available may be of varying standards of quality. Typical documentation tends to be in static media, such as printed text. With static media it can be difficult to find the expressive power needed to describe the dynamic nature of runtime behaviour.

Static documentation also runs into problems with some of the features of the OO language paradigm as mentioned in chapter 2. Notably, inheritance and polymorphism, along with dynamic binding and overloading methods, can hide exactly what piece of code is being called at a particular moment [JS94a]. Since these decisions are often made at runtime, class and code browsers would be unable to navigate properly as they typically do not function during runtime. Text documentation would become cumbersome if it had to describe every single possible execution path.

3.3.1 Animation as Dynamic Documentation

Trying to model dynamic behaviour in static media often disappoints. The static media has a limiting effect on the expressiveness and the amount of information that can be usefully put across to the reader, and new language features have complicated predictions of the execution path prior to runtime. Understanding would be better served by complimenting the static documentation with dynamic documentation that follows the dynamic runtime behaviour of an assembly. Animation is one such form of dynamic documentation.

In many real-world examples, people find it easier to comprehend something (that they can't yet do themselves for whatever reasons) if they can watch either somebody else experiment with it or watch a simulation of this happening. With the planned expeditions to other planets in the solar systems, NASA scientists use simulated animations to test out landings of modules on planetary surfaces, or use animations of maps created from overhead surveillance satellites to get a feel for the layout of the planetary or lunar surface. Similarly, military operators can plan a strike using simulators that show animations of their actions and possible consequences.

By creating animations of the runtime behaviour of assemblies, the reuser will have a better feel for how the assembly develops and what it does as time goes on. The animations can tap into human abilities to comprehend complex visual stimuli and help the viewer form a clearer mental model of what is going on.

3.3.2 Using Program Visualisation to Create Dynamic Documentation

If animations are a good source of dynamic documentation that could promote understanding, in turn promoting code reuse, then how can dynamic documentation be supported? Program Visualisation has been a field of research for many years now and has looked at this exact topic. Program visualisation will be explained in greater depth in chapter 4. It does not stop at just using animations, and incorporates a more expansive range of techniques aimed at presenting information to the user. However the use of sophisticated graphics (given the increased power of today's computing systems) is a common topic of discussion in this particular field. By using the work done in visualisation research, we can look at tool support for creating animations of the runtime behaviour of assemblies. These animations can then be used as documentation and give the reuser a greater understanding of what is going on and how.

The tool support undertaken as part of this thesis uses visualisation technology to create animations to document assemblies. A reuser's test driving of an assembly can drive the dynamic documentation created by the tool. This way the reuser gets to determine what the runtime behaviour of the assembly should be, and then gets to see what they did. In this manner, the reuser can determine what features of the assembly should be documented, and it what order the features should be documented.

Chapter 4 will now cover program visualisation in more depth.

Chapter 4

Program Visualisation

This thesis has claimed that understanding plays an important role in code reuse, and that at the moment understanding is usually gained via documentation, example or experience of the assembly in question.

This chapter looks more closely at research into visualisation. Visualisation is ideally suited to documenting an assembly's dynamic behaviour, and is a key part of the tool we have developed to support understanding. This chapter serves as a background to visualisation, and introduces the model and terminology that will be used in subsequent chapters. We are particularly interested in the field of algorithm animation, and how these animations can be derived from the test driving mentioned in chapter 3.

4.1 What is Visualisation

When we visualise something, we are forming a mental image (or model). This image or model helps in understanding that thing and can form the basis for how we interact with it. It is important that we interact with things correctly, and since visualisations guide how we interact, it is important that we form the correct visualisations. Research supporting the formation of visualisations has been active for many years. When forming a visualisation, the visualiser is not limited to just using visual information, and may use any available information, including that received by other senses. Since computers do not easily support touch, taste or smell, computer based systems are typically visual, or sometimes audio.

One result of visualisation research has been a refinement of presentation techniques. In mathematics and statistics such things as pie charts, bar graphs and histograms are often used to give a better representation than is afforded by the raw data itself. Books often use

pictures to show the reader something that would otherwise be difficult to convey using text alone. An electronic encyclopedia could use schematics to show how the internal combustion engine works, giving the reader a better appreciation of how the different components interact, and where the inputs and outputs are. Similarly, the encyclopedia could use audio cues to give a listener a better understanding as to how the sound varies amongst different musical instruments of the same taxonomic family (such as all reed instruments, for example). Tool support for visualisation supports users by presenting relevant information in a form that is easily digested. Not all visualisations are inherently useful however, and visualisation for the sake of visualisation is little better than text for the sake of text, and there must be some thought into the relevancy of what is shown to avoid cluttering the screen with trivialities.

Visualisation already plays a role in computer science. The art of creating good human-computer interfaces relies in part on a variety of presentational techniques to express information to the user in a clear and concise manner, so as to avoid confusing the user or distracting them with irrelevancies. In this project, we concern ourselves more with the visualisations of executing code rather than with HCI. When working with code, programmers form visualisations to understand it. The area of visualisation research that supports this is referred to in the literature as program visualisation, or software visualisation [1].

4.2 Why Use Program Visualisation

The standard forms of documentation, such as annotated source code and written manuals, are textual in nature and can often run into problems succinctly expressing the large amount of information present in a project [OWE96], as well as the dynamic behaviour at runtime. Understanding can be enhanced by making more efficient use of the human ability to comprehend complex visual stimuli. The human eye can register up to 150 M-bits/second. This compares well with the amount of information that is present in such interfaces as computer screens (8-24M-bits/second)[Glo94]. Program visualisation is increasingly useful given the recent increases in the power of graphical workstations. Roman & Cox [RC93] define program visualisation as a "mapping of a program to a graphical representation". The graphical representation helps the reuser to better understand a piece of code. Visualisations can make use of animations that are dynamic in nature, and these dynamic animations are better suited to documenting the dynamic runtime behaviour of assemblies than the more traditional styles

[1] Some practitioners refer to program visualisation as being a subset of software visualisation. [PBS94]

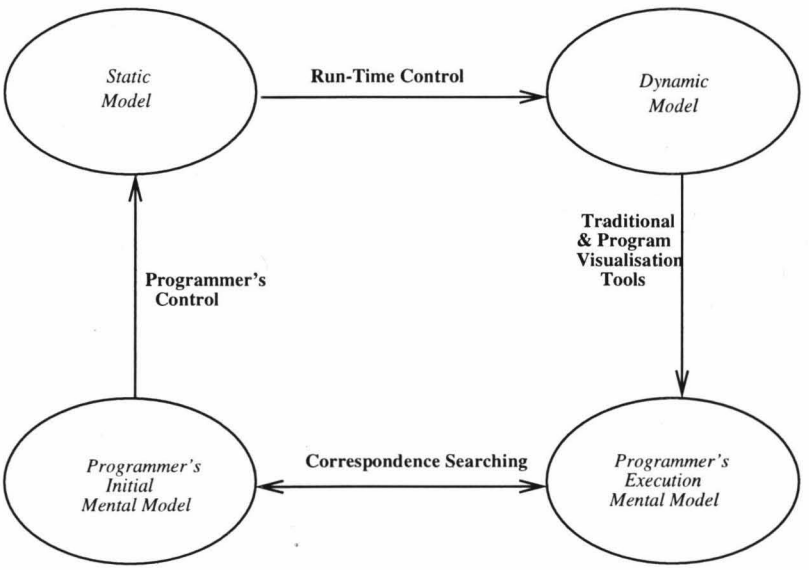

Figure 4.1: Oudshoorn, Widjaja & Ellershaw's model for the program development lifecycle.

of documentation. The possibilities for animations continue to increase with advances in the graphical capabilities of modern computers.

Widjaja and Oudshoorn have developed a model for how programmers form mental models of code during the development lifecycle. This model is shown in figure 4.1. It shows the the programmer's initial mental model of the program is turned into a static model, typically source code. The execution of this source code in turn creates a dynamic model. Program visualisation tools support the process of turning this dynamic runtime model into a mental model that the programmer can understand. Comparisons between the programmers initial mental model and the resulting execution model allow for the detection of inconsistencies and faults. Without these program visualisation tools, the mapping of the dynamic model to the programmer's execution mental model becomes more difficult [OW96]. An incorrectly formed mental model would then lead to confusion and error in judgement when relating it back to the initial mental model. Widjaja & Oudshoorn talk primarily about how comparison between the execution mental model and initial mental model is of use in the testing and maintenance phase of the software lifecycle, however much the same kinds of understanding are required for reusing other programmers' code than is needed for maintenance of that same code.

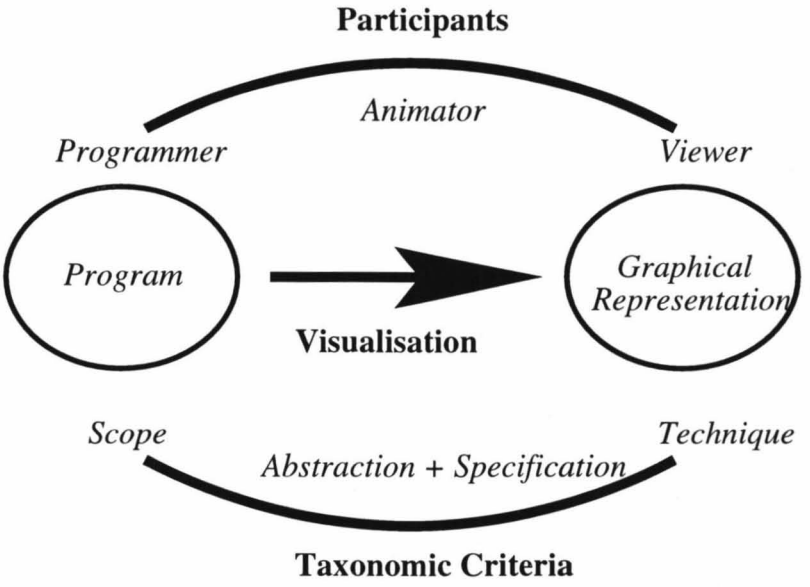

Figure 4.2: Roman and Cox's model for program visualisation and their taxonomy.

4.3 Interacting with Program Visualisation Tools

Roman & Cox identify three types of participants in typical interactions with program visualisation tools [RC93]. These are shown in figure 4.2. They are: programmers, animators and viewers. Programmers are responsible for the creation of the code that is to be visualised. Animators are responsible for implementing the visualisations of the programmer's code. These animators are not necessarily the software developers of the tool itself, as some tools allow for visualisations to be plugged in whenever needed rather than having a predetermined collection of animations that cannot be added to. Viewers are the end users of the tool and are for whom the visualisations are created. For the purpose of this thesis, we refer to Roman & Cox's *viewers* as *reusers*.

4.4 Characteristics of Program Visualisation Tools

Widjaja and Oudshoorn [OWE96] identify three aspects to program visualisation tools: purposes, ideals and mechanisms. The purposes of program visualisation tools are to understand behaviour, aid in debugging and in performance analysis. This thesis is concerned with the first of these three purposes.

Widjaja and Oudshoorn see two sets of ideals for program visualisation tools, one each from the perspectives of the system and the reuser. The reuser's ideals are that there be minimal disturbances to the visualised program; little or no reuser intervention; that it handle real world problems; and present the right things. Minimal code disturbances refers to how

the source code must be changed (if at all) to allow it to be used in the tool. Some visualisation tools require the source code be modified to alert the tool of important events, while other tools covertly gather this information. Reuser intervention refers to how much information a reuser must supply to the tool to create the visualisations. Some tools automatically create visualisations without any prompting, whereas other tools require some reuser input. The more abstract the visualisations become, the harder it tends to be for a tool to automatically work out what is needed in the visualisations. The tool should be able to handle real world examples. This refers to both the size of the examples and the range of source code allowed. If a tool only allowed small visualisations of source code written in a specially devised language, then it is of significantly less use to a reuser than a tool that allowed them to see visualisations of non-trivial code in their implementation language of choice. It is also important that visualisation tools show the right information.

The system's ideals are that the tool be scalable, extendible and portable. The issue of scalability is tied in with the user ideal of the system being able to handle real world examples. Extendible tools would permit new types of visualisations to be included and modified, so as to allow a reuser to better map what they want to see on to what they get to see. Portable tools allow for a movement to a new architecture with minimal required changes. Portability is a desirable feature as the reuser may not always work on one architecture and may want the benefit of visualisation regardless of what they are working on.

Lastly, they identify several mechanisms for program visualisation tools. These mechanisms are steps through the visualisation process, and start out with instrumentation and data collection, proceed through to post-processing, and ends with the actual act of visualisation. The mechanisms closely match the model of our visualiser discussed later in this chapter.

4.5 The PMV Model

The model we use for looking at program visualisation tools follows on from the work cited in section 4.4. The model has three distinct components: the program component, the mapping component and the visualisation component [Nob96]. These three components have one to one relationships with Widjaja & Oudshoorn's mechanisms and are also analogous to work done by other practitioners, such as Stasko and Jerding [JS94a].

4.5.1 Program Component

The *program component* is responsible for collecting information from the executing code, and is analogous to Widjaja & Oudshoorn's first mechanism. The program component can gather information in a variety of ways. One common form of information collection is for the source code to be annotated with calls to the visualiser. These annotations signal interesting events and pass through information that is needed for the visualisations to be created. These annotations can be manually inserted, or they can be inserted by a modified compiler. An alternative to annotation is to have the runtime environment interrogate the program while executing. This approach has the benefit of not requiring special modifications to the source code. One possible way of implementing the runtime interrogation is to use debuggers. Debuggers could execute the code and gather debugging information available in the executable.

Regardless of the method for gathering information, the program component is responsible for detection of runtime events. Once detected, an event is then handed on to the *mapping component*

4.5.2 Mapping Component

Not all of the information gathered by the programming component will be of interest to the visualisations. Also, the state of the information presented by the programming component may not be compatible with that needed by the visualisations. The mapping component exists to process the output of the programming component, removing any uninteresting events, and produce input to the visualiser in the correct format. This input is given to the third and last component, the *visualisation component.*

4.5.3 Visualisation Component

The visualisation component is handed information relating to interesting events in the runtime execution of code. This information should have been processed into the correct format by the mapping component, and the visualisation component must now take this input and create the visualisations. These visualisations can take a variety of forms. They can be static or dynamic, and show varying degrees of abstraction from the source code.

4.6 Taxonomies of Program Visualisation Tools

Over the years, practitioners have attempted to create program visualisation taxonomies. These taxonomies categorise existing technologies in program visualisation, and attempt to predict future advancements. While we do not attempt to devise our own taxonomy, we will briefly mention some of the more prominent taxonomies that have been presented. The concepts covered in these taxonomies will play a role in our discussion of successful tool support in chapter 5.

Myers [Mye86] first devised a categorisation along two axes, dividing tools up depending upon whether they visualise code, data or algorithms, and whether they create static or dynamic visualisations. Price, Baecker & Small[PBS94] break their taxonomy into six broad categories, these being *scope*, *content*, *form*, *method*, *interaction* and *effectiveness*. These categories can then be further broken down as needed. Roman & Cox originally used four categories: *scope*, *abstraction*, *specification* and *technique* (see figure 4.2), before splitting technique up further into *interface* and *presentation*. Most of these taxonomies have certain ideas in common.

4.6.1 What Can Be Visualised

Myer's first axis, along with Roman & Cox's *scope* and Price, Baecker & Small's *content*, refer to what aspect of the program is being visualised. A commonly agreed division of this aspect is into *code*, *data* and *algorithm*. An example of code visualisations is a pretty printing source code browser that could use colour to highlight certain features of the code (reserved words, comments, method signatures, and so on). Data visualisations concentrate on visualising the state of a program, such as the contents of an array during sorting, or the assignment of values to variables. Algorithm visualisations look at the dynamic nature of the execution. These show a sequence of states where a transition from one state to another could be interpreted as a statement or some such similar unit of execution.

4.6.2 How Can It Be Visualised

Roman & Cox's *technique* category (since split into *interface* and *presentation*) covers such topics as the vocabulary that can be used to construct the visualisations, and the types of visualisations that can be created. This latter point is also the focus of Myer's second axis on dynamic or static visualisations. The range of visualisations that can be created is important as the visualisations should be sufficiently powerful enough to convey the proper information

and, ideally, flexible enough to handle different kinds of visualisation, akin to Oudshoorn and Widjaja's user ideal of extendibility as cited in section 4.4. Static visualisations can show snapshots of features, whereas dynamic visualisations allow for a smooth animations of the events. In this thesis we primarily look at animations, although do not limit ourselves to smoothly flowing animations.

4.6.3 Requirements For User Interaction

The reuser may need to specify some information to the tool to help it create the visualisations. This reuser control can be a benefit as it allows for customisability, but too much required interaction puts strain on the reusers time. The reuser may not know enough about the program to know what parts to map (as the purpose of visualisation can be understanding, as identified above) so ideally the tool should be able to automate much, if not all, of the process. This is the focus of Roman & Cox's *specification* category and also of Price, Baecker & Small's *method* category.

4.7 Summary

In this chapter we have reviewed some of the research that has been undertaken in the field of program visualisation. In earlier chapters, we have identified program visualisation as possibly being of use in tool support for understanding assemblies. We have given brief examples of how visualisation is already widespread in common activities, such as reading. We have also seen that other practitioners have already come up with useful taxonomies and models. We use some of their terminology and concepts when we determine the requirements of, and design and implement, a tool to support understanding. In particular, our tool's architectural design makes use of the Program-Mapping-Visualisation model. By discussing past taxonomies, we see some of the general characteristics of program visualisation tools. These include the types of visualisations that are created, how they are created, and the form of interaction that is required, or allowed, on the part of the user.

This chapter, along with chapters 2 & 3, has provided a background, and supplied motivation for, the subsequent chapters on the requirements, design, implementation of our tool. In chapter 8, we combine discussion on the results of chapters 5, 6 & 7 with the material presented here.

Chapter 5

Tool Requirements

We have identified code reuse as a possible solution to the problem of meeting the high demand for software. Furthermore, we have identified the cost of understanding as a reason why code reuse has not been as successful as it could be. Software engineers often use tools to help reduce the time and effort required to complete tasks. It is our contention that tool support could reduce the cost of understanding. Two current methods of understanding are to have first hand experience of using the code, and reading any available documentation. Tools could support both these traditional forms of understanding. A tool could allow a reuser to test drive an assembly and create visualisations of the assembly being run. Test driving is discussed in chapter 3 and visualisation in chapter 4. This chapter now looks at the requirements of a tool that would support test driving and visualisation. Chapters 6 and 7 describe a tool that we have developed as part of our research. Chapter 8 will then evaluate this tool against the criteria discussed in this chapter.

Throughout this chapter we use terminology from the object oriented paradigm.

5.1 Test Driving Tools

In chapter 3, we mentioned that one current technique used to understand an assembly is to gain experience in using it. One way of gaining this experience is to write a piece of code tailored to invoke the assembly's functionality, and inspect its state. We call these pieces of code, *test harnesses*. Test harnesses do not use an assembly to solve a particular problem, rather they use it to show the reuser what it does, and how, so that the reuser knows if, and how, it can be used in later software solutions. We have called the act of creating and using test harnesses, *test driving*. We have argued that the manual creation of test harnesses

is time consuming, a factor that can have a detrimental effect on the cost of understanding, and therefore on the cost of code reuse. Manually written test harnesses also run the risk of having undetected faults that cause the assembly to be misinterpreted. Misinterpretation in turn leads to an ill-formed mental model of the assembly and problems ensue. It would be desirable to have tools that support test driving assemblies. We now look at some of the issues regarding test driving that tool support would need to address.

5.1.1 Matching Test Harness Capabilities

Using manually written test harnesses allows for a limited range of interaction with assemblies, and a tool should, at the very least, aim to support the same range. To achieve this, there are three major areas of functionality that tools would need to support. Firstly, the tool should allow the invocation of the assembly or parts of the assembly. In manually written test harnesses, the reuser would invoke individual methods of an assembly, so the level of granularity for functionality would be at the method level. A tool should therefore allow a reuser to invoke individual methods on an assembly. Secondly, a test harness could support the storage of values accrued during the test driving process. Assemblies, or parts thereof, may return values after execution, and these values can then be saved for use in later executions. This is analogous to the way that variables can store the return values from methods in manually written test harnesses, and are then used as parameters to other method invocations. Thirdly, test harnesses can iterate over code blocks. When iterating over a range of values in manually written test harnesses, the reuser can use the current value as a parameter to a method invocation, or as part of the state. The reuser can then see how a method's behaviour differs as changes are made to the parameters or state. Test driving tools could offer some similar feature, supporting iteration over a range of values. These values could then be used in accordance with a pattern specified either by the user or by general rules understood by the tool.

5.1.2 Access Control

Most OO languages have some form of access control, commonly some variation on the *public*, *protected* & *private* access controls that can be found in C++ and Java. These access controls help to promote encapsulation (discussed in chapter 2) by prohibiting a programmer from accessing or invoking non-public implementation. This forces them to go through the public interface. The design of tools to support test driving of assemblies requires that the designer

address the issue of access. The designer must decide how the access is handled, and how much access is allowed.

Since test harnesses are not normally part of, or derived from, the assemblies they are testing, test harnesses would typically be limited to going through the public interface. Therefore, tools supporting test driving should support at least a full range of interaction with the public interface of the assembly. The test harnesses would be limited to invoking an assembly's functionality at the level of granularity offered by the public methods. Protected or private helper methods would have to be invoked as a side-effect of a public method invocation, and the existence of private methods or state may not be publicly acknowledged. With regards to an assembly's state, a test harness would only be allowed to inspect that which was made available, and if the values were retrieved by public methods, then the reuser may not not be able to determine how data is stored. Since the reuser would normally be restricted to going through the public interface when reusing the assembly in their own code, it is tempting to ignore this issue. However, knowledge of the hidden implementation may be required to judge the performance of the assembly, such as whether or not an assembly uses a dynamic or static data store. The reuser may be interested in reusing the assembly, but it may need some modification to fit seamlessly into the new context. Reasons for modification could include the reuser using slightly different data types and wanting to change parameters types or state, or to improve the efficiency of an algorithm. If the reuser has incomplete knowledge of the hidden implementation, then this modification becomes more error-prone. State could be left behind that is required by the hidden implementation, or other methods that need to be included may be missed out. Tool support that allowed the test driving of an assembly's hidden implementation could therefore be of benefit.

Tools could make use of various techniques such as source code scanning or using libraries to bypass standard access controls, and allow finer access to the hidden implementation, and direct control over the internals of a class.

5.1.3 Restrictions on Allowed Assemblies

A tool that supported non-trivial inspection of assemblies would likely have to put some restriction on the set of assemblies that it could inspect. This limitation would help to reduce the complexity of the tool to a manageable level. One common restriction that most currently available tools impose is that all assemblies be implemented in one particular language. Some tools support C++ assemblies, others support Pascal assemblies, and so on. A tool might go

further than specifying that all assemblies be written in a certain language, and also specify that the assemblies must be implemented using some subset of the chosen language. The more restrictive the language set that is supported, the fewer the assemblies that will be able to be loaded. This will then reduce the range of opportunities for using the tool and reduce its usefulness.

Later in this chapter, we will discuss methods of mapping assembly implementation to abstractions in the visualisations. These mappings may require special hooks in the assembly code to alert the tool to events that are of interest (as mentioned in chapter 4 under the topic of annotation). Requirements that these hooks are in place would further restrict the set of assemblies that could be loaded. A reuser may wish to reuse an assembly whose source code is not available, such as a library. This assembly could not be used in a tool requiring event hooks.

As well as the issue of language, a tool's design would have to determine what size of assembly can be loaded and test driven. From chapter 2, the term assembly can be used to refer to an entire program or a single block of code. A tool might require that the entire program be loaded in, or it might allow for the loading of some subsystem, such as a class. The range of assemblies it allows affects its applicability and usefulness.

5.2 Creating Visualisations via Test Driving

In chapters 3 and 4, we discuss using visualisations to create dynamic documentation of assemblies. Combined with the test driving approach discussed in section 5.1, these visualisations could reinforce a reusers understanding of an assembly. As noted in section 4.1, not all visualisations are inherently useful, for much the same reasons that not all text documentation is inherently useful. Some visualisations may display information in a confusing manner, or leave out important details, so the creation of good visualisations is a non-trivial task. However, there is a need to minimise the time and effort required to use a tool that itself is trying to minimise the time and effort required for understanding. This suggests that having an easy and intuitive interface, and creating useful, if not perfect, visualisations is a good compromise.

5.2.1 Maximising Understanding

To maximise the effect that the visualisations would have on understanding, the question needs to be answered as to what should be visualised. This question can be split up into

several issues. The first issue is what the order should be for the sequence of visualised events. A method can be invoked on an assembly, thereby causing events to occur. However, some tasks may require the separate invocation of multiple methods, to achieve the desired effect. Similarly, the state of an object may need to be modified before an invocation, and the order in which these events occur could have a significant bearing on the outcome. If the visualisations are limited to covering just the events caused by the invocation of a single method, or modifying a single field, then there is a lost opportunity to show how the different parts of the class work together. An example is a class that allows access to a network. One method could be used to create a connection to another machine, another method could be used to bind a service to a particular port on a machine, and different methods could be invoked for sending and receiving packets. Each of these methods would require the execution of the other methods to achieve some meaningful goal for the reuser. Therefore, a tool would ideally allow visualisations to span multiple separate interactions with the assembly. This significantly complicates the automatic creation of visualisations. Given that a class usually has more than one public method, there is therefore more than one possible order of method invocations. As the reuser can theoretically initiate use of an assembly through any public method — even if not all possible scenarios are sensible — what invocations or field modifications should be visualised, and in what order? The assembly programmer typically has a lot of latitude in the nomenclature and decomposition of the assembly's functionality. This makes it difficult, in the general case, for a tool to completely automate the analysis of what orders of method invocations are meaningful. The reuser has specified an order of invocations or field modifications while test driving the assembly. It would be easier to visualise this order, given that it is known to be of interest to the reuser, and so the test driving acts as a catalyst and determinant for how the assembly is visualised.

5.2.2 Mapping Information From Assembly To Visualisation

Even if the visualisation uses the test driving to determine what features of the assembly are documented, the reuser may still have to supply the tool with information. The information would identify what particular events in the assembly were relevant to the visualisations, and what, if any, state information should be visualised. This mapping process was identified in Roman & Cox's definition of program visualisation given in section 4.2. It is conceivable that assembly programmers could supply pre-defined mappings from their assemblies to common visualisations, so as to help reusers. These pre-defined mappings could then be re-configured

to cater for changing circumstances. The amount of information would depend on the type of visualisation. In chapter 4, we mentioned that program visualisation can be split into three general areas: code, data and algorithm. This split is a feature of Myers [Mye86] taxonomy of program visualisation. Code visualisation would be akin to pretty-printing of source code. As such, the tool would most likely be able to determine what source code is currently executing by interrogation of the runtime system. The tool could well require no input at all from a reuser to be able to visualise source code. The main reason for this is that although the functionality of assemblies may differ greatly, the programming constructs used tend to be common to all assemblies in the same language. A tool that can parse a single programming language should therefore be able to visualise the code of any assembly implemented in that language.

The visualisation process becomes harder to automate as we move into data and algorithm visualisations. Data and algorithm visualisations show abstract representations of an assembly. As the visualisations become more abstract, the harder it is for a tool to determine what the correct abstraction is. Tools that only allowed a few types of generic visualisations, such as trace diagrams, could conceivably gather all the information needed from the runtime system. However, any visualisation that showed something specific to a particular type of assembly would run into problems. A tree visualisation could not assume that all the assemblies loaded into the tool had tree structures, or that all tree structures are organised identically. Otherwise it would have to either dramatically restrict the range of assemblies permitted in the tool, thereby reducing its usefulness, or fail when it tried to visualise a non-tree structure. Tools supporting assembly specific visualisations would have to be able to identify the category (such as tree, stack or queue) that an assembly resembled. This is a difficult task given the wide variety of different implementations of the same functionality. An example of this is a class that has a field containing an array of integers. The array is used as a queue and the reuser wants a visualisation of the queue so as to see what values are enqueued and dequeued, and when. However there is nothing explicit in the implementation language that identifies it as a queue, rather it is simply an array and, without knowledge of the algorithm, is indistinguishable from a stack, or any other array-based data structure. For a tool to be able to identify this array as a queue, it would have to recognise the algorithm that is used to determine how elements are added and removed. This is made more difficult by the fact that queues can be implemented in a variety of ways, each way achieving the same goal but in a slightly different manner. There is also no requirement that the assembly pro-

grammer use the common nomenclature for queues, so the tool can not rely on interrogating the naming scheme to determine what should be visualised and how. With this in mind, the reuser would most likely be required to supply some mapping information to the tool that would identify what state and methods are relevant to the visualisations, as well as the type of visualisation that is desired. It is conceivable that some subset of the assembly programmers and the visualisation programmers could decide on a convention to help in knowing what needs mapping. However, the mapping process should not be reliant on this, as not only does it remove the chance to use the large quantities of pre-convention assemblies, but since assemblies are not always written with reuse in mind, the assembly programmer may well ignore such a convention.

Given that the reuser is using the visualisations to gain an understanding of an assembly, then the amount of mapping information required should be kept to a minimum. Otherwise, creating the visualisations would require a deep understanding of the assembly, so as to know what in the assembly to map to the visualisation abstractions. A tool attempting to reduce the cost of understanding should aim to reduce the effort required in this mapping process. From this point on, we will be referring to data and algorithm visualisations, and the requirements for tools supporting these visualisations.

5.3 Accessing Runtime Information

Regardless of whether or not the reuser needs to supply mapping information, at some point the tool will need to gather information about events in the system. It may also need to gather information regarding the state of the assemblies when these events occur. This information can then be used to provide more detail to the visualisations. The information required could include the names of the methods that are invoked, the parameters that are passed, the return values, the state of objects that are used, the value of fields when accessed or modified, amongst other things [JS94b]. The tool could gather this information when some event has occurred, such as the invocation or return of a method, or the access or modification of a field.

Chapter 4 briefly mentioned various ways of retrieving this information, ranging from annotation of the source code (either manually, or in a modified compiler) to the use of debuggers to retrieve debugging information. The former technique could run into problems if the user does not have access to the actual source code, but rather wishes to reuse a pre-compiled library. Ideally, the reuser would not have to modify the source code of the assembly

under inspection as such modification may not always be possible (as in the library example above), and also has the potential for introducing faults into the assembly. In the worst case, these faults may pass unnoticed at re-compilation and cause incorrect behaviour at runtime. The incorrect behaviour may then be mistaken for the assembly's proper behaviour and misunderstanding would ensue, negating the value of the visualisation.

The latter case of using debugger technology requires that the debugger has sufficient access at runtime to gather all the required information, and also requires that the debugger be able to be easily driven from inside the tool[BMMWT99]. The reuser would typically have to give some indication of what events detected by the debugger are of interest (such as particular breakpoints in the code) and the data that should be gathered.

5.4 Visualisation Routines

Having gathered information from an executing assembly, and having been told by the reuser what is and is not useful, the tool then draws the requested diagrams or animations. The visualisation routines responsible for drawing could be built into the tool or be plugged into the tool when necessary. Since there is no accepted definitive visualisation that will perfectly convey all the information present in any type of assembly, there is a need to support multiple types of visualisations. We call a group of routines that draw a particular visualisation, a *view*. In section 4.3, we mentioned that Roman & Cox [RC93] identified three different types of people, who play different roles in program visualisation. One such category was that of the *animator*. The animator is responsible for implementing the views. A reuser may or may not also play the role of an animator.

The limits that the tool places on what a view can do have an important effect on its usefulness. These limitations come partly from the limitations on the information that can be passed through to the views for visualisation, but limitations also arise from the *vocabulary* that animators can use. In this context, I take the term vocabulary to mean the range of graphical constructs that can be used in the creation of a visualisation. Examples of such constructs would be squares, circles, lines, text, audio accompaniment, 2D/3D graphics and movement. If the vocabulary is very limited, then the animator may have trouble conveying information in the easiest and most efficient manner. A richer vocabulary allows for more sophisticated visualisations.

The manner in which visualisations can be added to the tool is also important. If the set of supported visualisations is fixed and cannot be added to, then the reuser is reliant on the tool's

animators being able to think up all the different visualisations that could be needed for all the possible types of assemblies. There is a chance of a useful, assembly specific, visualisation being unavailable, and this unavailability could impede understanding. A sufficiently abstract visualisation that could model all the assemblies conceivably used in the tool would only be able to show some common trait, such as an execution trace. A market could potentially form by having skilled animators come up with high quality views, that the reusers can then use good visualisations for different assemblies. Assembly programmers may decide to supply a visualisation specific to their assembly, and may even supply a default mapping for it.

Supporting the inclusion of different views allows for new and previously unthought of perspectives on assemblies. These perspectives can then help to give a better overall picture of how things work.

5.4.1 Creation of Views

Since the aim of the tool is save on the time and effort required in software development, the creation of visualisations should not be expensive in these regards. The tool may or may not directly support the creation of views through an editor. The creation of views would involve identifying the key events (or operations) in a visualisation, and the information (or attributes) that the animator requires, followed by implementation of the drawing algorithms. As an example of the operations and attributes in a visualisation, a stack visualisation might have different animations for the *pop* and *push* operations. These would be separately invoked by the tool when it detected an event that was analogous to the popping or pushing of a value onto a stack, as likely identified by the reuser. The animator would typically want to show some information in the visualisation, such as the current contents of the stack, so the animator should be allowed to request that this information be gathered from the assemblies. The tool should therefore support animators to specify the operations and attributes in the views. This would help to ensure that the tool can give the reuser all they need to know to be able to map from the concrete assemblies to the animator's more abstract operations and attributes.

5.4.2 Reuse of Views

Since we are interested in promoting code reuse, it would be in keeping with this philosophy to simplify the process of creating new views with the reuse of old views. Having general views could be of use to show common features of assemblies, and then more specific instances

of these views could be created to show more implementation detail. This would lead to a hierarchy of views, with the general views at the top of this hierarchy. These general views would then be broken down into more and more specific views, where the specific views could expand on the behaviour exhibited by a more general view in a similar fashion to the class inheritance hierarchies found in most OO languages. Alternatively, a view could be allowed to use other views as sub-components, similar to composition in OO languages.

5.5 Criteria for Test Driving and Visualisation

We now list a set of criteria for tools support test driving and visualisation. These criteria are derived from the discussions in previous sections of this chapter, and will be used in chapter 8 to evaluate our own implemented tool.

5.5.1 Access to Assemblies

1. A test driving tool must at least offer the range of interactions with assemblies that would be allowed using test harnesses. The reuser should be able to thoroughly inspect the public interface of an assembly and invoke functionality through it. The more access the tool allows, the more the reuser may be able to understand the implementation.

2. The tool should be able to store the values created during the test driving process, so that these values can be in later interactions.

3. The tool should allow an interaction to occur multiple times, possibly using a range of values, and should support the automation of this process as much as possible.

5.5.2 Required Changes to an Assembly

1. Source code annotation requires access and understanding of the source code. A reuser may not be able to meet either, or both, of these requirements, so a tool supporting understanding via test driving and visualisation should aim to gather runtime information without requiring assembly annotation.

5.5.3 Visualisation Capabilities

1. The visualisation vocabulary should be extensive enough to allow for a wide expression of concepts.

2. Visualisations that are generic enough to work on all assemblies can give a good overview, but often run into problems when describing highly abstract descriptions of assemblies. Support for useful, assembly-specific visualisations is important for getting a well rounded set of perspectives on an assembly. The inclusion of assembly-specific visualisations therefore requires that reusers (or animators) can create visualisations, and include them on the fly.

3. The cost of writing code for the visualisation routines should be as low as possible, as the tool aims to *save* time and effort expended by software engineers.

4. Given that these tools should not use annotation of source code for determining what information to visualise, the reuser may well need to tell the tool what parts of the assemblies relate to what parts of the visualisations. This mapping process would be difficult to automate completely, however the amount of time, effort and understanding required on the reuser's part, should be kept to a minimum.

5.5.4 Learnability, Portability & Scalability

1. It should not be hard to understand how to use tools that support easier understanding of assemblies. The more time and effort that is required to learn how to use the tool, the less the tool benefits the reuser, and the less likely it is to be used. Reusers may not have the chance to attend courses, or read documentation, about the tool, so the tool's interface needs to guide the reuser in what the correct course of action should be. The tool should therefore have good learnability.

2. The issues and costs involved in code reuse are not limited to any one particular platform or architecture. An assembly written for one architecture may be reusable, with perhaps some porting required, on another architecture. The ability to easily test the assembly on its own architecture would give insights as to how it currently works. This would therefore require that the tool be able to run on, or be easily ported to, different architectures.

3. The tool should also be able to handle real world assemblies rather than just small, practice assemblies. In terms of size, assemblies can range from the small to the very large, and these should all be supported in the tool. Unscalable tools may well be useful for pedagogical purposes, but would not help lower the cost of reuse in the wider software community.

Chapter 6

Design & Implementation

As part of our research, we have developed a tool to help users understand what an assembly does and how it does it. Our tool allows a reuser to test drive assemblies. It also employs visualisation techniques to create animated documentation of the assemblies at runtime. We have undertaken the development of this tool for two reasons: firstly as a means to further our own understanding as to what technology support is needed to develop such tools, and secondly as a proof of concept that such tools can aid in promoting reuse.

Chapter 5 discussed the requirements of a tool supporting reuse through the facility of test driving and the use of visualisation. This chapter will look at the design and implementation of our tool, which we have christened *Dyno*. Chapter 8 will discuss how Dyno measures up against the criteria from chapter 5.

We begin this chapter by outlining the various sections and components that form the architectural design of Dyno. Following on from this architectural overview, we discuss our choice of implementation language, and then give more detail on the individual components. The individual components are discussed in the order of information flow, from test driving to program visualisation. At relevant junctures, the technologies used in the components are introduced and discussed. The chapter ends by looking at some of the problems encountered during Dyno's design and implementation.

6.1 Design Overview

Dyno's core functionality is split up into two distinct, interconnected sections, the *test driver* and the *visualiser*. This is shown in figure 6.1. The test driver allows the user to inspect and execute assemblies, and in doing this can initiate the the creation of animations in the visu-

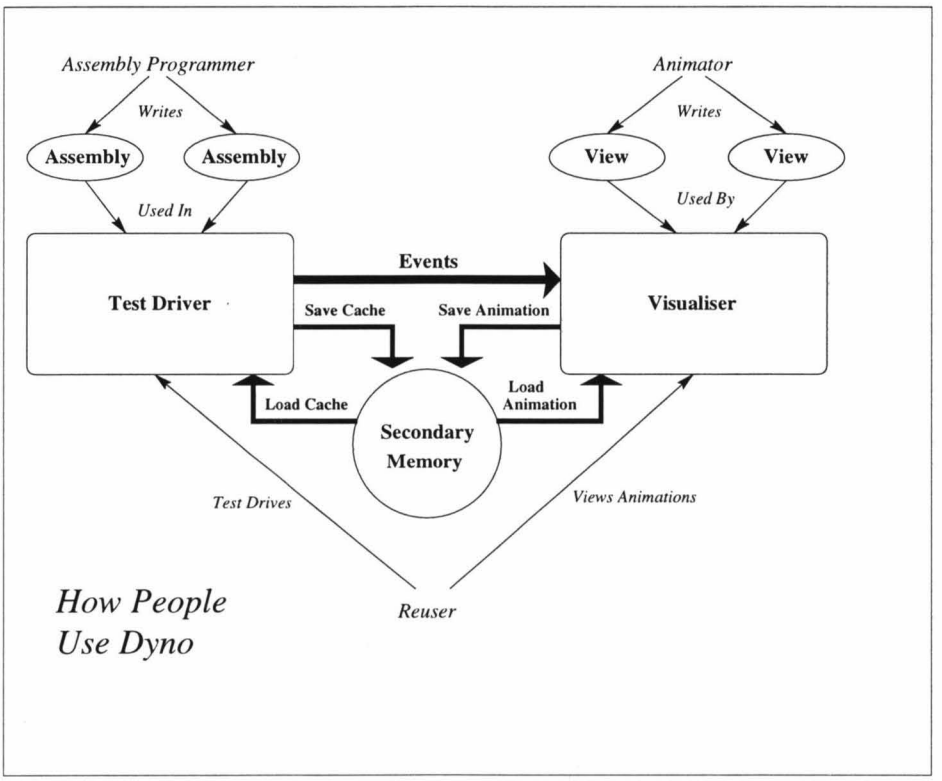

Figure 6.1: General Overview of Dyno.

aliser. The visualiser handles the creation and display of animations of the assemblies loaded into the test driver. It also provides facilities for storing and retrieving these animations as well as for viewing them. The visualiser creates the animations by gathering information from assemblies in a manner that is partly user specified. Both sections can operate quasi-independently, the test driver can be used to execute operations and inspect the attributes of an assembly without having to create animations, and the visualiser can view previously created animations without any input from the test driver.

Figure 6.2 shows a further breakdown of the test driver and visualiser sections. The test driver consists of the *object cache*, the *object cache browser* and the *entity information browser*. The *object cache* acts as a primary memory store for loaded assemblies and the instances of these assemblies. It can also store and retrieve a collection of assemblies, and their instances, to and from secondary memory. Dyno asks the user to enter a key, analogous to a name, for each instance of an assembly. This request is made when the instance first enters the object cache. This key is then stored with that instance, so that the reuser can easily retrieve it later on. To better support retrieval of data of a similar nature, such as all instances of the same assembly, there is a partial ordering imposed on these key-value pairs to keep like data logically together.

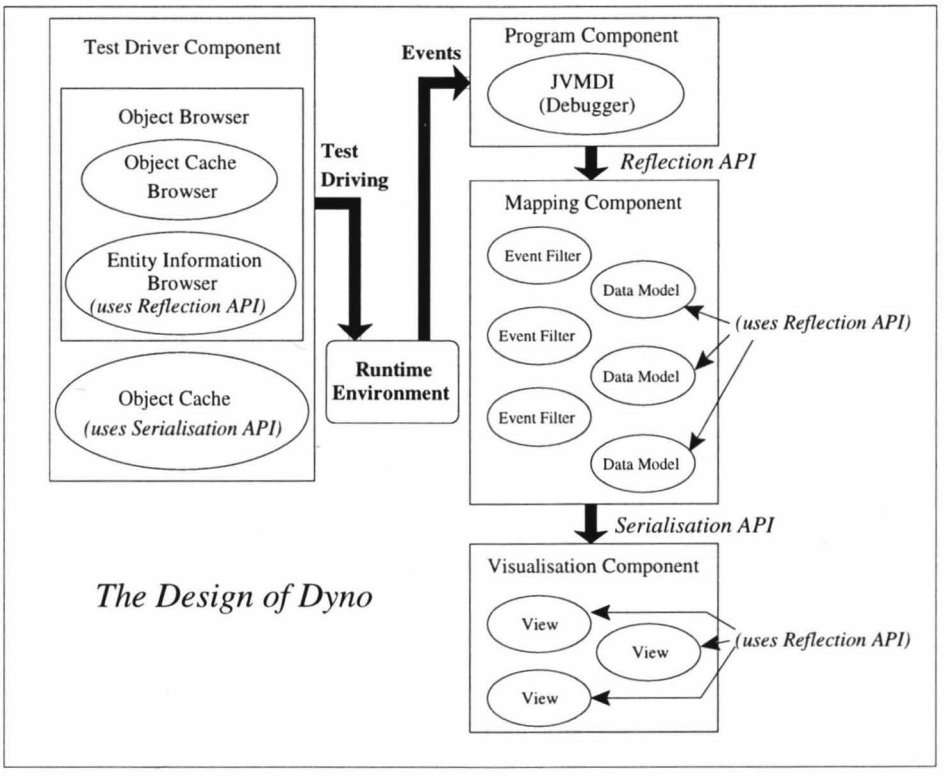

Figure 6.2: Dyno's architectural design.

The *object cache browser* acts as an interface on to the contents of the object cache. The *object cache browser* displays the assemblies and assembly instances in a logical order so as to give the user some appreciation of the different classifications present. The object cache browser displays only the keys and, implicitly, the types of the contents of the object cache. The *entity information browser* shows more specific information relating to the currently selected item in the object cache browser and is therefore tied closely to the object cache browser. The entity information browser shows the operations and attributes of an *entity*, a term we use for either an assembly or their instances. The entity information browser acts as the main focus of the test driving. The user can use the displayed operations information to invoke those operations, and can use the displayed attributes information to inspect their state. The object cache browser and entity information browser combined form the *object browser*.

The object browser and the object cache combine to form a single major component called the test driver component. In comparison, the visualiser section is split up into three separate components: the program component, mapping component and the visualisation component. These three components are derived from the PMV model discussed earlier in chapter 4. The program component is responsible for detecting events in the loaded assemblies at runtime,

and for gathering information regarding those events. The mapping component filters the information gathered by the program component so that only the information relevant to the animations is kept, and also adds any state information that may be useful in the animations. The visualisation component creates and manages the animations from the information passed to it by the mapping component.

The program component interrogates the runtime assemblies for information regarding method invocations, method returns, field accesses and field modifications. Having gathered this information, it passes the information on to the mapping component. The program component does not filter out any events of the four types listed above on the basis of usefulness to the animations, leaving this entirely to the mapping component. The program component requires no direct contact with the user.

The mapping component consists of event filters and data models. Event filters exist to filter information received from the program component, and only pass on information that the user has signalled as being relevant to animations. The design of our tool supports the creation of more than one animation during the course of a test drive, so there is one event filter per animation. Data models map information in the assemblies to any extra information that may be needed by the animations and that cannot be retrieved directly from the event data. For example, an event such as the invocation of the *push* method on a stack might not directly describe the entire contents of the stack. A data model would map the stack's array to some construct in the animation. As with the event filters, the mapping between assembly state and animation state is user specified. There is one data model per animation and thus one data model per event filter.

The visualisation component contains the animation code. Our design does not limit the type of animations to a pre-determined set, instead the design allows the user to load in code for creating the animations during the tool's execution. We refer to code for a particular animation as a *view*. When a user decides to create a new animation, and has loaded in the view for creating their preferred type of animation, then the tool requests the user to specify how state and events in the assemblies should map to state and events in the animation. This information is then stored in the aforementioned event filters and data models. As the user test drives, the visualisation component takes the information passed through by the mapping component, and passes it on to the correct view. The view then creates the animation and displays it in a display window. The display window provides functionality for the user to rewind, play, fast forward and stop the animation. As well as taking information on the fly

as the user test drives, the visualisation component may also take input from a file previously created by the mapping component.

6.2 The Use of Java in Dyno

We now discuss our choice of programming language, and its impact on the design and implementation. We have implemented Dyno using Java so as to benefit from the Java "write once, run anywhere" philosophy, as well as to allow us to explore several new features that have emerged in the Java language at the time of this research. Exploring the new features has allowed us to further our own understanding as to what kinds of language or library support are needed to develop tools of this kind. Throughout the rest of this chapter we will refer to certain language features or libraries when they are relevant to the implementation of a certain section of the design.

Our choice of Java extends to both the implementation and application of Dyno. Dyno supports the test driving and animation of assemblies written in Java, and our assemblies are Java classes. For the rest of this chapter, we will use the term class instead of assembly.

We make use of several Java Application Programmer Interfaces (APIs) throughout the implementation. The Reflection API allows us to introspect classes, object and arrays. The programmer can invoke methods and constructors, and access and modify fields, and leave the decision of what method or field until runtime. The entity information browser uses the Reflection API to get information on the methods, constructors and fields for display on screen, and also uses it to invoke methods upon user request. The program component uses the Reflection API to communicate with the mapping component. The mapping component uses the Reflection API to interrogate the state of entities deemed relevant to the animations. Lastly, the visualisation component uses it to invoke the correct method in the view to ensure that the animation progresses in the appropriate manner.

Dyno also makes use of the Serialisation API. The Serialisation API allows for the storage and retrieval of Java objects, as well as for the passing of objects over streams. This is useful for saving and loading from secondary memory, and also inter-component communication. Our discussion of these Java APIs and features will be expanded on as we discuss the implementation of Dyno.

As well as these two APIs, our use of Java extends to a new feature of the Java 1.2 release. The Java Virtual Machine Debugger Interface (JVMDI) gives us the required functionality to properly interrogate executing code and allows us to detect certain types of events, including

those that we have expressed interest about in section 6.1. One of the implications of using the JVMDI is that the JVMDI is built around the Java Native Interface (JNI). The JNI allows for *native programming*, whereby code is written in a language other than Java (typically C or C++) but may be linked with methods of a Java class. When that method is executed, the virtual machine instead executes the native code (C/C++) rather than any Java code. This allows for a programmer to write code that can take advantage of certain architectural and design features of the machine they are on, but limits the portability of the overall system to the portability of the C/C++ code.

By using the JVMDI code, we have reduced the portability of Dyno below that of pure Java code. However, C/C++ compilers are extremely common and the JVMDI code is highly portable in itself.

6.3 Test Driver Component

6.3.1 Object Cache

The object cache uses a hashtable (implemented using the *java.util.Hashtable* class) to store the entities. This hashtable is a table of key-value pairs, where the key is either supplied by the reuser, or in the case of entities that are classes, the fully qualified class name.

The object cache uses the Java *event/listener* model to alert any cache displays when a new value is added, so that the display can be updated. The object cache does not explicitly maintain any ordering on the key-value pairs, such as grouping all the entities of a similar class together. However, if an object is added to the cache, and that object's class has not been seen before, then that class, and any superclasses that have also not been seen before, will also be added into the cache. These classes will be included in the cache, and any cache display, with their fully qualified class names used as keys. The reason for this is to support cache displays that show a hierarchical ordering, such as the hierarchical ordering typified by the Java inheritance model.

6.3.2 Object Browser

The object browser allows the user to see the available methods and fields in a class, and also to invoke those methods independently. Normally a user is only allowed to execute code by executing the entire program, but Dyno allows the user to interact with code at a finer level of granularity than is typically allowed in other environments. The object browser acts

as the interface to the test driving features of Dyno.

Object Cache Browser

The object browser shows a list of the currently loaded Java classes and also shows any objects created and saved while test driving these classes. This is displayed using a hierarchical ordering based around the Java inheritance hierarchy. The user loads classes into Dyno by first giving a list of URLs of directories or Java Archive files (files that typically have a .jar extension) to search, and then giving the fully qualified name of the desired class. Dyno will then search the directories and Java Archive files for that class, storing it in the object cache. If the loaded class uses another class which is not currently loaded, then when that unloaded class is first used during an invocation or when a field of that class is first inspected, Dyno will load it using the same search procedure as for classes loaded explicitly by the user. There is no restriction that the class must be on the local machine, and it may be accessed from any machine the user has access to and can give a valid URL for.

Entity Information Browser

The entity information browser shows more detail information about a selected entity. The entity is selected via the object cache browser, and the entity information browser displays information regarding the methods and constructors available for that entity, along with its state. The entity information browser also allows the reuser to invoke the displayed methods and constructors, and to further investigate and modify the state. The Reflection API is used for both the static retrieval of information from an entity, and the invocation and modification of methods and fields respectively.

6.3.3 Reflection API

Dyno uses the Reflection API first available in release 1.1 of the Java language. The Reflection API is a collection of classes that supports introspection of other classes and objects, and accessing and modifying classes, objects and arrays at runtime. It allows us to request a list of objects representing the methods and fields of a class, as well as allowing us to invoke the methods on instances of an appropriate class (or on the class itself in the case of static methods), and also inspect and modify the state of objects.

Dyno can create instances of a class at the user's request. This is typically done by having the user select a constructor from the list of those available in the class of interest, and then

invoking that constructor. Ordinary methods may be called in the same manner, and may have return values that result in objects being created as a side-effect of invocation.

In release 1.2 of the Java language, we are permitted to access and use methods and fields irrespective of the access controls specified by the programmer of the class under inspection. Overriding the usual security measures allows the user to inspect and modify parts of an object or class that would ordinarily be off limits due to the lack of accessor and set methods, and to explicitly invoke the helper methods that are used by the publically available methods. We will discuss the usefulness of this in chapter 8.

Not only does Reflection permit us to access the state of ordinary objects, but it supports the accessing and modifying of array elements at runtime. This allows the user to keep and usefully maintain array instances in the object cache for use as arguments passed through to invoked methods or constructors. It may also be used to allow closer inspection as to the state of an array that comprises part of another object. While arrays are technically objects, they are handled by a separate set of Reflection methods. An array of objects cannot be substituted for a single object, even if the elements of the array are the same type as the single object. The reverse also holds, and arrays need special methods to access particular methods. Arrays also have a length property that individual objects do not possess.

6.3.4 Serialisation API

The Serialisation API allows for the storage and retrieval of Java objects and for the passing of these objects over streams. The act of serialising creates sufficient information that the object state can be fully recreated at some later point. Serialising an object requires serialising all of the objects that are reachable from the first object. The set of reachable objects is typically the fields of the object being serialised. If these fields are themselves objects, and not primitive variables, then their set of reachable objects is also serialised, repeating as necessary. If a single object would be serialised more than once, due to being referenced multiple times from other serialised objects, then a reference to the first serialised instance is inserted where the second serialised instance would go. The serialised information can be sent over a stream, that may connect to a file on secondary storage or some other process. The other side of this stream can then read this information, and use methods present in the Serialisation API to retrieve the objects. Any references that the serialiser inserted due to multiple serialisations of the same object, are then changed to be the actual object itself. This is done automatically for the programmer. The objects are then cast back into whatever type they should be by

the programmer. The Serialisation API has a different set of methods for serialising primitive types, but these methods work in a similar fashion.

The Serialisation API restricts the types of objects that can be serialised by requiring that all serialisable classes implement the Serialisable Java interface. There are no methods in the Serialisable interface, it merely acts as a flag to indicate to the runtime environment that the programmer of the class has determined that users of the class should be able to serialise instances of it. There are integrity implications to be considered when serialising objects as objects could be modified between being written out and read back in.

For use in later executions of the tool, the test driving component uses the Serialisation API for maintaining a secondary memory version of the object cache. Later it will be shown that the mapping component and visualisation component of the visualiser also use the Serialisation API to communicate with each other, both via files and streams.

6.4 Views

Dyno uses Java code to create the animations seen by the reuser. Dyno does not have a set number of animations that can be used in the system, instead it allows any class to be used to animate information as long as that class follows a few basic guidelines. We call a class that meet these guidelines a *View*. All views must inherit from the *dyno.visualiser.View* class. This class in turn extends the *java.awt.Canvas* ensuring that all views draw their animations on a canvas. This fact makes the visualisation component's display screens easier to implement later on. The writer of a view is called an animator, and an animator may or may not be a reuser.

A view has methods that relate to the different kinds of events that could occur in the thing being animated. Using a simplistic example, a view designed to show an animation of a stack would have methods for push and pop. The view would not implement a working stack, as the state of the stack and the events enacted on it would be be passed to it after Dyno had gathered that information at runtime. Rather when a push or pop occurs in the stack code under inspection, the push and pop methods in the view are invoked and cause the right animation to be drawn. Each execution of a view's methods results in the creation of a *frame* and the combined frames make up the complete animation. Given that a view method will be executed every time an interesting event occurs, then the events can be said to map 1—1 to the frames. Note that our use of frame is a little simplistic as the term frame implies a single still in an animation. However, there is nothing to stop a single event from

causing an sub-animation to occur, that could consist of multiple stills. We use the term frame for convenience, as our later examples create only a single still per event.

These views are part of the visualisation component in section 6.7 but are also used in the mapping component at the user interface level. The user interface asks the reuser what the events and state in the classes should be mapped to in the views, and for this the user interface needs to show a description of the view. For the description it shows the names of methods responsible for animation in a view and any attributes the animator has specified.

6.5 Program Component

The program component is responsible for gathering the information relating to events that occur when the user executes code loaded into Dyno. This information will be used in subsequent components to create the animations that the user will use to understand the executing code. In previous chapters we have discussed some of the ways that visualisation tools can gather this information, such as the programmer annotating the code to call visualisation libraries, or using a modified compiler to automatically put in these calls. These approaches all have their advantages and drawbacks. The approach used by Dyno is to use a new debugger technology called the Java Virtual Machine Debugger Interface (JVMDI). It is used in Java native code, code that is invoked from the Java Virtual Machine but is implemented in another language, most typically C or C++.

6.5.1 Java Virtual Machine Debugger Interface

The JVMDI permits someone using it to access the common functionality of debuggers from within their own programs. Someone using the JVMDI can set breakpoints in their code and be notified when those breakpoints are hit by having a specified method invoked. The JVMDI also allows inspection of the bytecode by bytecode execution of the code, and the detection of a host of different kinds of events beyond merely hitting breakpoints.

Detectable Events

The JVMDI can detect, amongst other things, the access or modification of a field, the invocation of a method or the popping of a frame off the call stack, the linking or unlinking of a class in the Java Virtual Machine, the beginning or end of a Java thread, and the throwing or catching of an exception. If a programmer using the JVMDI requests notification of these events when they occur, then a method specified by that programmer will be called

at the appropriate time and information relevant to that event passed as an argument. The programmer may then also request more information using the methods and classes available in the JVMDI. Dyno is interested primarily in the invocation and return of methods, the access and modification of fields and the throwing and catching of exceptions. We have chosen these events as they are some of the most significant in terms of interactions among classes and objects, and are should therefore be shown in the visualisations.

6.5.2 Detecting Events

Dyno will place breakpoints at the beginning of each method when a class is loaded, and will also ask the Java Virtual Machine for notification of when the fields of that class are accessed or modified. The user may then invoke methods on that class, or more likely an instance of that class, using the object browser. Alternatively they may inspect or change a field value. Dyno is immediately alerted when either of these events occurs, and the invocation of the method or operation on the field value is temporarily suspended until our native code, using the JVMDI, has gathered sufficient information for the visualisation process.

In the case of method invocation, Dyno will determine what called the method, on what the method was called, the name of the method in question and the arguments passed. Dyno will explicitly ask for notification as to when the current frame on the call stack is popped, so as to be able to later signal to visualisations that the method has returned. Frame pops are not automatically signalled by the JVMDI libraries, so must be requested on a individual basis. In the case of field access or modification, Dyno will determine what field was accessed, on what object and, if it was modified, what the old and new values are.

The following table shows the information that Dyno collects for the each type of event.

Event Type	**Event Information**
Method Invocation	Sender, Receiver, Method Name, Parameters
Method Return	Sender, Receiver, Method Name, Parameters, Return Value
Field Access	Field Name, Field Class, Container Object Accessor Class, Accessor Method
Field Modification	Field Name, Field Class, Accessor Class, Accessor Method, Container Object, New Value

Dyno will detect events that occur due to invocation or inspection of all classes except those that are part of the Dyno implementation itself or are part of the Java Development Kit libraries. This is a conscious design decision rather than a limitation of the JVMDI,

as we did not want events from commonly used and understood JDK classes to clutter and mask the information gathered from the classes the user is inspecting. This also avoids the problem of Dyno's animation classes creating a circularity by causing events to occur while processing previous events. The implications of this are discussed in chapter 8.

Dyno will pass the gathered information to the next component in the visualisation process: the mapping component. The program component makes minor use of the Reflection API to communicate with the mapping component.

6.6 Mapping Component

Not every piece of information gathered by the program component will be relevant to the animations. In some cases the information will be related to a event type the user is not currently interested in, in other cases the information may relate to *instances* of an event type that the user does not deem relevant.

As an example of the former, consider that a user may wish only to know about the execution trace through a code component, but Dyno will detect information relating to the access or modification of fields in that component as well as just the invocation and return of methods. An example of the latter would be a situation where a user wishes to see animations of calls to an object. The object is part of a collection of interconnected objects. Dyno will detect method invocations on the object in question and will also detect when those methods return, but it will also detect method invocations and returns relating to the other objects.

The mapping component takes all the information gathered by the program component and filters out all that is deemed unnecessary. If it deems an event detected by the program component to be relevant to an animation, it will then make a snapshot of the current state of any objects of interest and will pass all this on to the visualisation component. A snapshot is the current set of values for all the objects and primitive variables that the user has specified as being of use to the animations.

Dyno's mapping component makes use of two sub-components, the Event Filter and the Data Model, to fulfill these tasks. The event filter determines which events are important to a particular animation and the data model is responsible for determining the current values of data mapped to the animation.

6.6.1 Event Filter

An event filter is responsible for maintaining a list of all the events that should be passed onto the visualisation component if they are detected by the program component. Dyno supports the ability to construct more than one animation per test drive so there is a separate event filter for each animation. This helps to keep separate what events relate to what animations. The program component gives a copy of every detected event to each event filter. Each event filter then filters out those events deemed irrelevant for the animation they are responsible for.

The user supplies the information used to differentiate relevant events from an irrelevant events. When the user wants to create an animation, they are first asked to select a View class that represents the type of animation they want. They are then asked to map events that might occur on entities in the object cache onto the methods in the selected view class. The event filter uses a Java vector to store the list of events the user is interested in, so beyond memory considerations there is no limit to the number of events that the user can map to animations. Dyno has a graphical user interface that enables users to create mappings from loaded entities to a view, without having to know too many details about what is going on in the underlying system, and with a minimum of training.

A reuser can map as many events as they want to an animation. Each of these events can be one of eighteen basic types. These eighteen kinds come from the different possible combinations of three characteristics that each event can have. The characteristics are: *target object for event*, *field or method event*, and *timing of event*.

Target Objects

The reuser can specify that the event must occur on a particular object for it to be considered relevant to the animation. The reuser can either select a single object or have an empty entry for this characteristic, in which case the event is considered to be relevant to the animation as long as the other two characteristics match with something at runtime, irrespective of the object it occurs on. To specify a single object, the user supplies the name given to that object in the Object Cache mentioned earlier in this chapter. If the user wishes a particular event to be considered relevant if it occurs on two distinct objects, then two distinct mappings must be created.

Field or Method

A mapping can relate to either a field or a method of a class. It may also relate to all fields or all methods of one or all the classes. If no object was specified as the target object, then the only choices here are that the mapping be for all methods or for all fields of all classes (and not both, for which two separate mappings would be required). If a single object was specified, then as well as the all methods or all fields options (that now applies to all methods or fields *of that object* rather than all classes), the user may also specify a single method or field permitted on the specified object.

Timing of Event

As well as giving a target object and selecting a field or method (or all fields or all methods), the user may also specify when an event should be deemed relevant. If the previous characteristic has a method value, then the user can select whether they only want events relating to the invocation of that method passed through, or whether they only want events relating to that method returning, or whether they want both. Likewise if the user selected a field for the previous characteristic, they may specify whether the event should only be considered relevant if it is an access of that field, a modification to that field, or both. These values are valid regardless of whether this mapping concerns a single field or method, or all fields or methods.

6.6.2 Data Model

An animation not only needs to know when an event occurs in the code, but also needs to know some information about the state of objects in that code at the time the event occurred. For this reason, an animator can code a view to include attributes. For example, a stack view could have a *elements* attribute and a *size* attribute. The animator would specify Java types along with the names for these attributes, in the above example the *elements* attribute could be a java.lang.Object array and and the *size* attribute could be an primitive integer. When this view is loaded into our system and the user requests its use, the attributes appear in the interface for creating the mapping (as first mentioned in the discussion on Event Filters above) and the user must specify a mapping from the objects and information available to these attributes. The types do not need to match up precisely, as the system is capable of limited forms of conversion. Following on from the example above, if the user wished to map a vector to the object array attribute *elements* then the system will handle this.

When a relevant event is detected by the mapping component, the system takes a snapshot of the information that has been mapped to the animation's attributes. If the types of the information and the types of the attributes do not match, then translation will be attempted, such as vectors being turned into arrays or vice versa. If any fields need to be accessed, or methods invoked, for this snapshot to be taken, then the mapping component employs the Reflection API discussed earlier.

The snapshot of the attribute values is then passed to the visualisation component along with the rest of the information regarding the event. The animation can then access this information and use it in the creation of the individual frames that go to make up the complete animation.

6.6.3 Communicating with the Visualisation Component

The mapping component is responsible for sifting through the information gathered by the program component and determining what is relevant to an animation, and when it is relevant. Once something has been deemed relevant, it must be passed on to the animation. The visualisation is part of the next major component in our system: the visualisation component.

The animations are meant to be a form of documentation for the user to refer to. For this to be effective, it can not be a time consuming chore for the user to view the animation. Ideally there should be no need to duplicate the test driving the user went through to create the animations, so the user should be able to save the animation to some form of secondary storage that the visualisation component can then refer to later on. The mapping component performs the act of saving the information it has filtered by using the Serialisation API.

Our mapping component can send the serialised information to a user-specified file for later retrieval by the visualisation component, or can send it directly to the visualisation component to be shown in semi-real time. The benefits of this are that the animations are available for future reference both by the user who created them, and by other users further down the line, as well as being available immediately if the user wishes to see the animation as they are test driving. In the Serialisation API there is no need to treat a stream pointing to a file any differently from a stream pointing to another process. This support of streams makes the design and implementation of the mapping and visualisation components reasonably simple.

6.7 Visualisation Component

The visualisation component is the last major component in our system. At this point, the information has been gathered by the program component, filtered by the mapping component and has now been presented here for displaying. The visualisation component primarily consists of a display for controlling animations, based around the tape deck analogy, and the animations themselves. As mentioned earlier, the animations are handled by the views.

The display for the Visualisation component is the *view screen.* The view screen has buttons for playing, rewinding, fast forwarding and stopping animations. The *stop* button effectively pauses the animation rather than setting it back to the beginning. As well as this there are indicators for showing how far through an animation the user is, and a way of selecting from the available animations, referred to as *sequences* in the visualisation component. As the user may view more than one animation at any one time, there may be multiple view screens, with each view screen servicing one animation. Each view screen is associated with one instance of a view, and the view screen takes information from the mapping component (either indirectly via a file or directly over a stream) and uses the Reflection API to invoke the correct animation method on its view.

The view screens contain a canvas as the main element of the interface and the views that handle the animation write to this canvas. All view classes extend dyno.visualiser.View that itself extends java.awt.Canvas.

6.8 Problems

There were several problems encountered during the design and implementation of our system, all of which related back to the technology that we were using. We will now discuss these problems in the order in which they were encountered.

6.8.1 Accessing Return Values

The program component requires access to all kinds of information with regards to the user's test driving. Since test driving some code frequently means invoking methods, our system attempts to divine information relating to, amongst other things, the method being invoked; the object the method is being invoked on; the parameters being passed through; and, in the case of a method with a non-void return type, the returned value. The Java Virtual Machine Debugger Interface (JVMDI) permits us to easily do all but the last of these, failing

to allow easy access to the return value. Due to this missing functionality on the part of the debugger, the system requires the code under inspection to be specially annotated where any returns occur. This notation involves the invocation of a static method called *Send* from a specially created class called *Return* using the return value as a parameter. The system has been designed to be on the look out for invocations of Send on the Return class.

When our system sees an invocation of Send, it stores the parameter value that is then later used when the user's invoked method signals that its frame has been popped off the stack (i.e. it has returned). There are several methods called Send in the Return class, each taking a different type of argument. There is one Send method per primitive type, one Send method for objects and one Send method that takes no arguments to handle void return values. Void returns are handled in the same way. This allows a simplification so that automated scripts can modify code files to include this new annotation, even though strictly speaking there are no values to store in the case of void returns.

This use of annotation goes against our stated goal to allow unmodified code to be used in the system, however it is hoped that it is a temporary fix until the JVMDI is enhanced to cover the missing functionality.

Thread Safety in the Debugger

A side-effect of how return values are retrieved in the program component is that the implementation is no longer thread-safe with respect to the debugger. This problem arises from the fact that a global flag is set every time the system detects an invocation of a *Send* method of the *Return* class, and the parameter to the *Send* method is saved in a global object variable. The flag alerts the system that a return value has been stored in the object variable and is checked when the method that just invoked *Send* returns to its own caller, so that the object variable's value can be used.

If the user is invoking a multi-threaded class, then it is possible that two threads could call *Send* one immediately after another. The latter thread could then potentially overwrite the value stored in the global object variable by the former thread before the former has a chance to use them. This would result in the former thread mistakenly using the latter thread's return value, which in the best case would be a different type thus raising an exception. In the worst case, the latter thread's return value would be the same type as the former thread's return value and the error would go by unnoticed.

There has been no serious attempt to make the mapping component (or visualiser com-

ponent) thread-safe. There are a few other places where a reuser could get into trouble if using threaded applications, especially if they were to have two filters write to a shared file, and two threads concurrently caused events to write data to that file. Since the file is not locked using any of the usual thread safe mechanisms, such as condition variables or mutual exclusion devices, the data could be corrupted.

6.8.2 Non-Serialisable Classes

The use of Java's Serialisation API allows the storage and retrieval of Java objects. This permits information for the animations to be saved, and also allows the object cache mentioned at the beginning of this chapter to be saved between executions of the system.

However, the integrity implications stemming from being able to modify Java objects while they are being stored means that Java classes have to be specifically flagged as being serialisable by implementing the Serialisable interface. The majority of classes currently in the JDK do not implement this interface, nor would the majority of Java classes being written by programmers. This poses a problem when a reuser animator wants to save non-serialisable objects in the object cache to file, or when an animator wants to use non-serialisable objects in the animations. In the former case, Dyno ignores all non-serialisable objects while writing to file, so these values will be lost. In the latter case, it would seem more likely that the animator, in writing a view, would more likely use the primitive types and the String class — all of which are serialisable — to store the basic kinds of information to be displayed. The animator must therefore use only serialisable types for attributes and the arguments to operations.

When the user wants to use a class that does not implement the Serialisable interface, they need only set up a filter that maps the state of the non-serialisable objects to the serialisable types used by the animations, and the mapping component will do the relevant translation before sending the now serialisable data over the stream.

6.8.3 Threads, Serialisation and Real Time Animations

The visualisation component has the ability to create the animations as the user test drives. However, due to a problem with combining the data streams used to send the serialised information, and the thread technology that would be used for concurrently handling execution in the program, mapping and visualisation components, the animations do not occur in absolute real time. The problem arises because when a thread is closed, the thread will also close any

pipes or steams that are associated with it. Since a test drive can span multiple user-initiated method invocations, or field accesses and modifications, it is difficult to isolate what events should be handled by what threads, especially if there is more than one visualisation being created at once. The best solution is to have a single thread handle a single user-initiated action, i.e. all the events resulting from a user-initiated method invocation are handled by one thread, and those are the only events that thread handles. When this thread is destroyed however, the connection (pipe stream) between the mapping component and the visualisation component will also be destroyed, as the thread would have made use of it. But this connection needs to stay up, so that future events relevant to the same visualisation can get through to the visualisation component. The simplest solution was to not explicitly create threads at all on the program and mapping component side. The result of this is that the program and mapping component must finish processing all the events for a single user-initiated action before the animation can start playing.

6.8.4 Animation Storage

The information needed for the creation of animations can be saved to secondary storage for future use. Ideally, these animations would be saved after the information from the mapping component had been used to create the images that make up the animations. This would mean that only the images would need be stored, and in the future the images could be loaded directly from secondary storage and displayed immediately. Unfortunately, the serialisable images resulting from any decent sized animation take up too much space for this to be practical, so instead the images must be recreated by the Visualisation component each time they are required. This is also a problem when a user wishes to rewind through an animation, as the previous frames must also be recreated from the mapped information rather than from a store of images as the size of these images rapidly overwhelm the memory requirements of any modern personal workstation.

The result of this is that extra computation is required on the part of the Visualisation component, although no extra input is needed from the user, and the effect on speed is negligible in all but the most sophisticated animations.

Chapter 7

Dyno In Action

7.1 Introduction

We now demonstrate how a reuser uses Dyno to explore and understand an assembly. We describe a scenario where a reuser develops an understanding of what a Java class called *MyNetwork* does, and how it does it. *MyNetwork* implements functionality that enables a reuser to send and receive simple messages over the internet. The reuser develops their understanding by exploring the methods that can be invoked on *MyNetwork*, and by seeing what the state is and how the state changes. The reuser then invokes animations to document actions performed on *MyNetwork*. These animations show what methods are invoked, and when, to fulfill the reuser's requested actions. The reuser can gain a better understanding of what the *MyNetwork* component is doing, and how it is doing it, by seeing these animations. Multiple animations give multiple perspectives, and the reuser can keep the animations stored on a file for later use.

7.2 Initiating a Dyno Session

Dyno is a Java application. The reuser starts Dyno in a manner consistent with the standard procedure for starting up Java applications. Dyno allows for the exploration and documentation of assemblies, where an assembly in this case is typically a Java *class* or *interface*. Java arrays and primitive types may also be stored in the system, even though these are not what is commonly meant when talking about Java classes.

Having started a Dyno session, the reuser is presented with the main screen as shown in Figure 7.1. From here the reuser has access to a menu bar that may, amongst other things, be used to load assemblies, create instances of Java arrays, and handle the animation

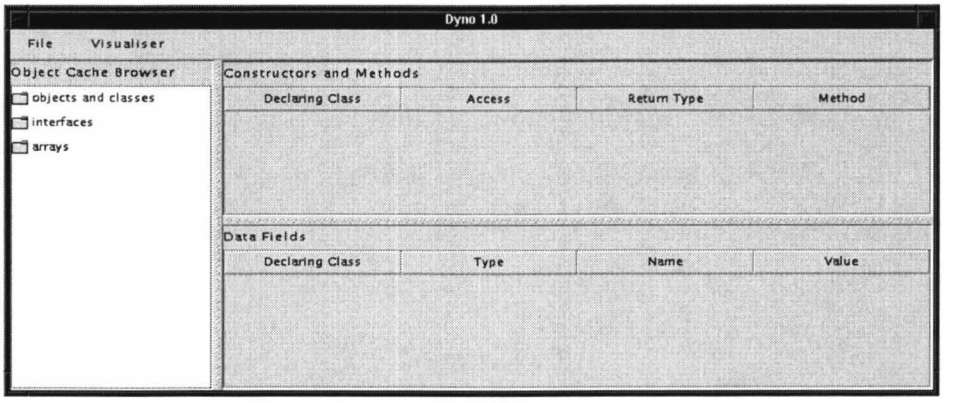

Figure 7.1: Dyno's main screen that the reuser is presented with on startup.

capabilities. The rest of the screen is split into two main areas, the object cache browser and the entity information browser.

7.2.1 The Object Cache & The Object Cache Browser

We use the term *Entity* to describe an element from the set of assemblies and the instantiated instances of these assemblies. These entities are stored in the *Object Cache*. The left hand side of the screen in Figure 7.1 contains the *Object Cache Browser*. The reuser uses the object cache browser to select the entity they wish to inspect. The object cache browser is organised as a tree-like hierarchical structure. The three subtrees at the top level represent the three categories for entities in the object cache.

The three categories are:

- Objects and Classes
- Interfaces
- Arrays

An assembly can be loaded into Dyno from a Java class file or from a previously saved object cache, read from a file. Instances of an assembly result from invoking constructors and from method return values. The subtrees are ordered hierarchically, using the Java Inheritance model for the assemblies, and ensuring that all instances of an assembly are shown as child nodes of the node representing that assembly.

Some entities may exist in more than one of the three subtrees. This would occur if an entity was an instantiation of an assembly, in which case it will appear in the *objects and classes* subtree, and also conforms to a particular Java interface, in which case it will also

appear in the *interfaces* subtree. Entities are grouped in this manner to help the user find an entity that conforms to a certain type. This is important for finding a particular entity to explore, given the reuser will typically knows its general type and name, and also for browsing the stored values that could be used as a particular argument in later method invocations.

Even though arrays technically have classes [LY97], they are stored separately from ordinary objects and classes in the object cache and object cache browser. The reason for this is that although there is a class for a String array, instances of this class are not substitutable for instances of the String class. The same applies for all other classes and their array equivalents. Separating them out makes for a clearer distinction between what can be used in particular situations (such as the value of a parameter to a method).

7.2.2 The Entity Information Browser

The right hand side of the main screen contains the *Entity Information Browser* and shows specific information for the entity currently selected in the object cache browser. If no entity is selected in the object cache browser, then the entity information browser is empty. The entity information browser is split in half horizontally. The top half contains a table that lists any methods or constructors that can be performed. The bottom half contains a table listing the fields. If the entity selected is a class, then only the static methods and static fields will be shown. If the entity selected is an instance, then only the non-static methods and non-static fields will be shown.

7.3 Loading An Assembly

We have discussed how the main screen is laid out. We demonstrate this layout by using our example whereby the reuser decides to load a component called *MyNetwork*.

Loading *MyNetwork* requires at most two steps. Assemblies can be loaded from anywhere that the reuser can give a valid URL for. The reuser gives Dyno the URL for the directory or Java Archive File where the assembly is stored. Dyno maintains a list of these URLs and searches through them when given a fully qualified class name to find. The first of the two steps is therefore to check if Dyno's list of URLs contains the right URL. This is done by going to the *File* menu and selecting the *Add To Classpath* option. This brings up a new window that contains a list of the URLs that Dyno knows to look at, and a text field and button with which to add new URLs.

In our example, the URL of the directory where *MyNetwork* can be found is not currently

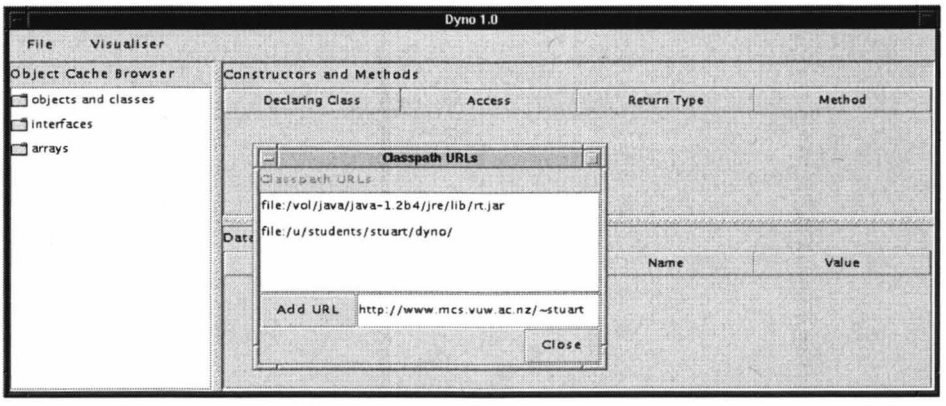

Figure 7.2: Interface for adding new URLs to the current Classpath, enabling Dyno to find classes later specified by the reuser.

listed. The URL in question is "http://www.mcs.vuw.ac.nz/~stuart/". The reuser types this URL into the text field as demonstrated in Figure 7.2, and clicks on the button to the left. An alternative to clicking the button is to press the *Return* key in the text field. At this point the list box will be updated to include the new URL.

Now satisfied that Dyno knows about the directory where *MyNetwork* is stored, the reuser proceeds to the second step. This second step is to select *Load a Class* from the *File* menu. The reuser is presented with a simple dialog box requesting the name of the desired assembly. The reuser supplies Dyno with the fully qualified class name, in this case just *MyNetwork*, via the usual form of dialog box interaction. Dyno now uses the list of URLs to find the class. Had the class not existed, then Dyno would have given feedback to the reuser noting that the name is not valid. However in this case the reuser has correctly typed in *MyNetwork* and, as the correct URL has been added in the first step, Dyno is able to find the *MyNetwork.class* file in the directory pointed at by "http://www.mcs.vuw.ac.nz/~stuart". The class is now loaded into the system. At this point the object cache browser is updated so that *MyNetwork* is visible, as can be seen in Figure 7.3.

7.3.1 Specifics of the Entity Information Browser

The addition of the *MyNetwork* assembly has caused an automatic update of the object cache browser. The class *MyNetwork* inherits directly from *java.lang.Object* and a node in the *objects and classes* subtree appears for *java.lang.Object*. This only occurs the first time that something inheriting from *java.lang.Object* is loaded. Another node appears as a child node of *java.lang.Object*, representing *MyNetwork*.

Any other assemblies that are later loaded and that inherit directly from *java.lang.Object*

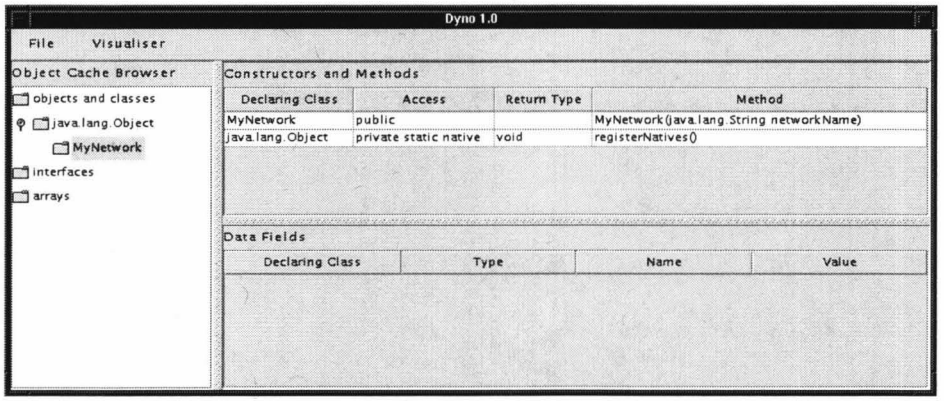

Figure 7.3: Main Screen updated for the now loaded MyNetwork component.

will appear as children of the *java.lang.Object* node along with *MyNetwork*.

The reuser clicks on *MyNetwork* in the object cache browser and Figure 7.3 shows the updated screen. *MyNetwork* is a class and so only the static methods and fields are shown in the entity information browser. Note that all inherited static methods and fields are shown.

The information shown in Figure 7.3 includes: access properties of a method, constructor or field; the return value in the case of a method; the class where it is defined (in case it is inherited from a super class); and the signature (name and, for methods and constructors, the parameter list).

7.4 Creating Instances of Assemblies

The reuser decides to create an instance of *MyNetwork*. This is be achieved by double-clicking on a row containing the name of a constructor in the method/constructor table. There are two possible responses to this action. If the constructor requires no parameters then Dyno will ask for confirmation that the constructor is to be invoked. Assuming an affirmative confirmation, the code for that constructor is then invoked. The reuser is alerted upon completion of the constructor that a new object has been created and a name is requested for the object — analogous to a variable name — for future identification in the object cache. A default name is based on the name of the class and a unique, monotonically increasing, number. The instance is then placed in the object cache and the object cache browser updated in the same manner as previously described for assemblies.

The second possible response occurs if, as in this scenario, the reuser has selected a constructor that requires arguments. In this scenario the selected constructor takes a single *java.lang.String* argument. Dyno displays a window (shown in Figure 7.4) listing the argu-

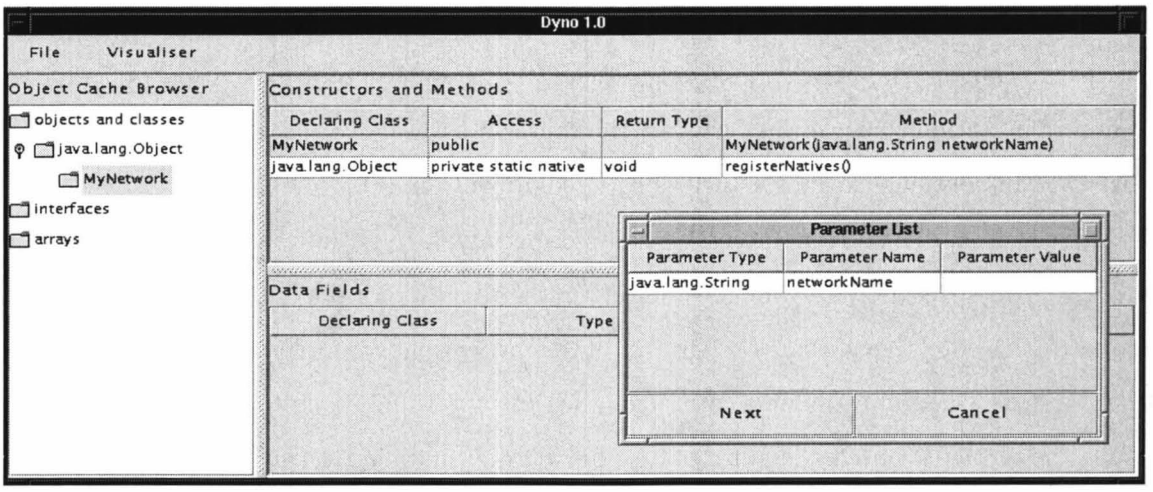

Figure 7.4: An interface that allows the reuser to specify values for parameters to be passed to a method or constructor.

ment types and names, and has text fields for the reuser to type in literal values or the names of objects (of a conforming type) in the object cache.

There are currently no objects of *java.lang.String*, or any other conforming type, in the object cache. The reuser enters a literal value. The name of the parameter is "host" so the reuser decides to enter "debretts.mcs.vuw.ac.nz", a host the reuser knows exists. The reuser clicks on *Ok* and now gets a request for confirmation that the constructor should be invoked with the given values. The reuser affirms and the result of all this is that the reuser now has an instance of the *MyNetwork* class. The reuser types in *myNetworkObj* when requested for an instance name. The object cache browser is now updated to show a new node for *myNetworkObj* as a child of the MyNetwork node.

7.5 Browsing Assemblies and Their Instances

Selecting the *myNetworkObj* node updates the entity information browser to show the instance methods and fields only. The reuser can browse through the list of methods and fields at will, and may derive some useful understanding of the class just from the naming scheme used in the class.

The reuser decides to go one step further, and test drive the *MyNetwork* class by invoking some methods and seeing what happens. Invoking instance methods is accomplished in the same manner as invocating constructors. If the method implementation requires any interaction with the reuser, then that interaction occurs normally as if Dyno was not there. The only situation where Dyno makes its presence felt during the method invocation (beyond the

creation of animations) is if the method implementation raises an exception. If an exception is raised, Dyno brings up a message box identifying the type of exception and any other useful information that can be derived. The reuser must remove this message box before the method can continue executing (if it is capable of continuing).

If the method runs to a successful completion, Dyno tells the reuser if there is a return value, and if so what the value is. The reuser then has the opportunity to save that return value to the object cache for later use in other invocations or for further study. As before, the reuser is asked for a name to store as a key for any value saved to the object cache.

If the method caused the state of an object to be modified, then the values in the fields table will be updated. The method could have affected more than just the object it was invoked on, and when those objects are displayed they will show the new values.

7.5.1 Browsing Through Data Fields

The reuser can continue to invoke methods. They could then gain some understanding by looking at the side-effects, in the form of return values or modifications to state. Extending the idea of exploration, the reuser can also look in more detail at the fields of an object or class. Figure 7.5 shows *myNetworkObj*'s fields. Each field is displayed in a single row, and this limits how much can be shown and how much the reuser can learn. An object value for a particular field may have a state consisting of several fields of its own. However unless the field is a primitive type, or one of a select few classes that implement a more extensive way of printing its own state as a string (examples of which are *java.lang.String* and *java.util.Vector*), only the type, field name and object's unique hash code are shown. A primitive field will show the primitive value, and classes such as *java.lang.String* and *java.util.Vector*, that implement a *toString()* method will show whatever is returned by *toString()*. If the entity selected in the object cache browser had been an array object, then each row in the fields table would have shown a different element of that array. The name of the field on each row would have been the particular index of that row's element. Each row would follow the same rules as above to determine what should be printed out as the field value.

In the *MyNetwork* class, there are two private fields that demonstrate this behaviour. Both _outputBuffer and _inputBuffer are instances of *MyQueue*. *MyQueue* is a class that was loaded automatically by Dyno the first time it was needed by *MyNetwork*, using the same search and retrieve technique as for user-specified classes. The information shown in Figure 7.5 does not tell the reuser much about their state however. The only thing that can be

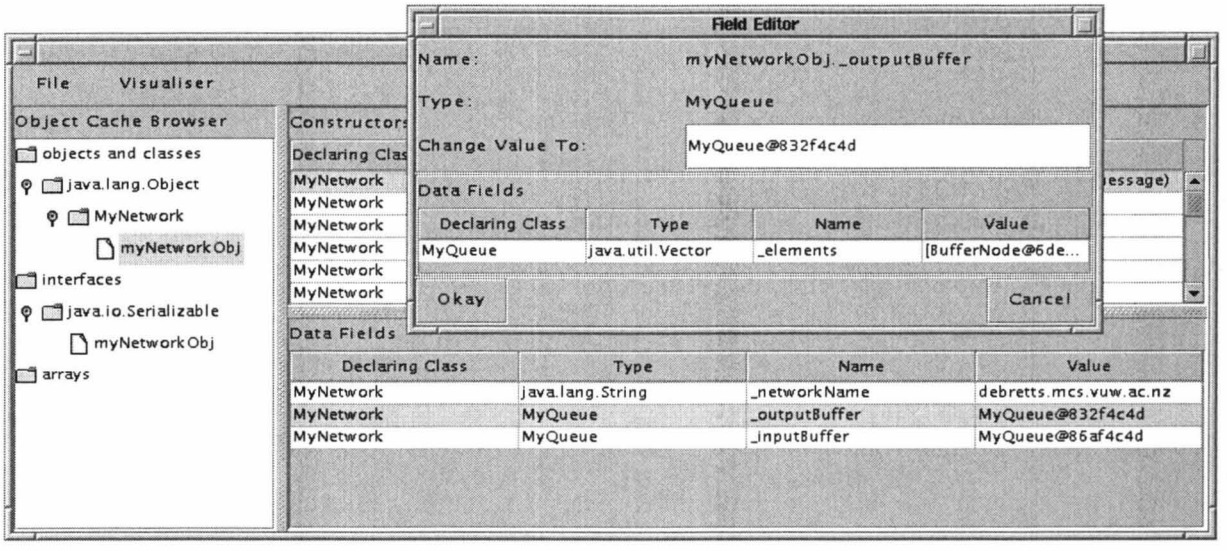

Figure 7.5: This interface allows the reuser to browse and edit a field's own internal state, to add a finer level of control over the state of the component under inspection.

determined from Figure 7.5 is that they are different instances because they have different hash codes following the class name.

The reuser wants to know more about _outputBuffer, more specifically what is currently stored in the queue-like structure. The reuser double-clicks on the _outputBuffer's row in the data fields table and is presented with the new window shown in Figure 7.5. This new window lists _outputBuffer's fields. Had the reuser double clicked on a primitive type, then the table would have shown no fields but merely repeated the primitive value. The reuser can now see the internal state of _outputBuffer and get a better understanding of what is currently in it. The reuser now explores even further. Double-clicking on fields in this window causes new windows of a similar style to appear, showing the states of the chosen fields.

7.6 Creating Animations

The reuser has invoked methods and investigated the state. At this point, the reuser knows how the structure of *MyNetwork* is organised. Assuming the reuser has invoked a few more methods, then there is the chance to see how data structures change as different methods are invoked. The reuser has also gained first hand experience of what the method arguments and side-effects are. By varying the arguments to methods, and changing the invocation order of a collection of methods, the reuser can see if there are any pre-conditions that a method expects another method to have already set, potentially causing different results. However, this is only one part of Dyno's functionality. As well as allowing the reuser to invoke methods

of a component, the reuser may request that Dyno create animations to illustrate their test drives.

Code for drawing the animations is written in Java. Java classes that specify how animations are to be drawn, based on information gathered by Dyno, are called *Views*. A view contains methods that may be invoked by Dyno to create animations that represent the events (method invocations, method returns, field accesses or field modifications) in the entities the reuser is working with. Certain events may result in a particular method in the view being selected. A number of different types of action may result in the same view method being called, and more than one view method can be invoked as a result of a single action. The views can get state information from the executing entity, and that is then used to flesh out what the animations show. In our example, the reuser selects a view that creates an animation similar to a UML style Sequence Diagram. An execution trace is only one of the types of animations that Dyno can handle. A view could also represent a graph animation, a tree animation, a stack animation, or some more user-specific animation written with a particular assembly in mind. Dyno does not offer an editor or compiler with which to write these views, but there are classes within the Dyno implementation that the animator can use to retrieve recursive or historical state information.

7.6.1 Supplying Dyno With Mapping Information

To create these animations, the reuser tells Dyno to map certain actions in the entity (such as a particular method being invoked) to a method or methods in the selected view. The reuser also identifies the parts of entity state that the view should use when presenting information in the animations. The reuser decides they want to create a UML-style sequence diagram of *myNetworkObj*. The reuser knows of the existence of a view class that animates a UML-style sequence diagram. This view is implemented in the *TraceView* class. Using a view to handle an animation requires that the reuser first create a *filter* for the animation. A filter is a representation inside Dyno of the information needed to be able to map from the concrete implementation details in the loaded assemblies to the methods and state used by the views. Dyno can detect events in the runtime system that relate to method invocation, returns, field access, field modifications and exceptions being thrown. When Dyno detects events in the runtime system, it passes them through the filter to determine if the event is relevant to the animation. The filter will also determine what data should be passed through to the animation methods. Therefore, the creation of a TraceView animation requires the creation

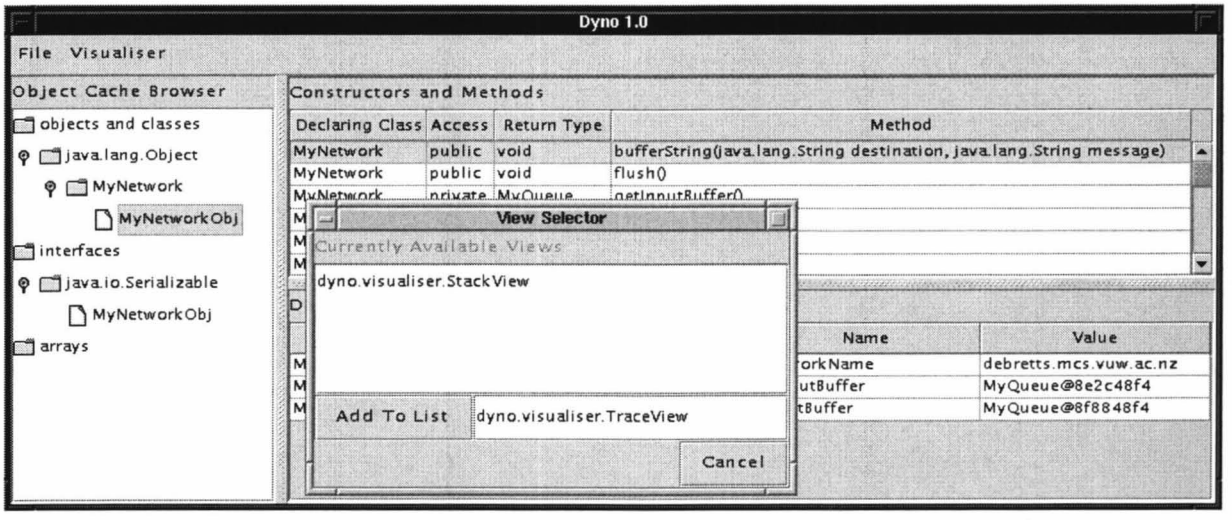

Figure 7.6: This interface shows a list of the views that Dyno currently knows about. These views are needed not only to create animations, but to give the reuser more information about what is needed in the way of mappings.

of a TraceView filter. For the reuser to use *TraceView*, they go to the *Visualiser* menu and select *New Animation.*

The reuser will be presented with a new window similar in style to the one for adding a URLs to the classpath, seen earlier in the session. Figure 7.6 shows the result. The comparison between loading views and loading classes does not end there. Dyno uses the same list of URLs that it used to search for the assembly class files to search for the specified view's class file. The list at the top shows the currently loaded views, and *TraceView* is not among them. The reuser types *TraceView* into the text box, either hitting the return key, or the button to the left, upon completion. *TraceView* is now loaded into Dyno and the reuser can now create a new filter that will collect information on state and actions, for that kind of view, to translate into animations. The reuser does this by double-clicking on the name in the list.

Once the reuser has double-clicked on a view name, another window appears. This window is split into two halves as shown in Figure 7.7. It lists the methods in the view class and the attributes that it uses to draw the animations. The attributes are used as places on which to map state from the reuser's assembly. The TraceView does not contain many methods or any attributes. The methods are listed showing name and have a boolean tick-box associated with them, that shows a tick if there are any actions mapped to that method. The attributes are also listed by showing names supplied by the animator. Each individual attribute also has a tick box to show whether there is anything currently mapped to it.

From the reusers point of view, each method in a view implements one part of an ani-

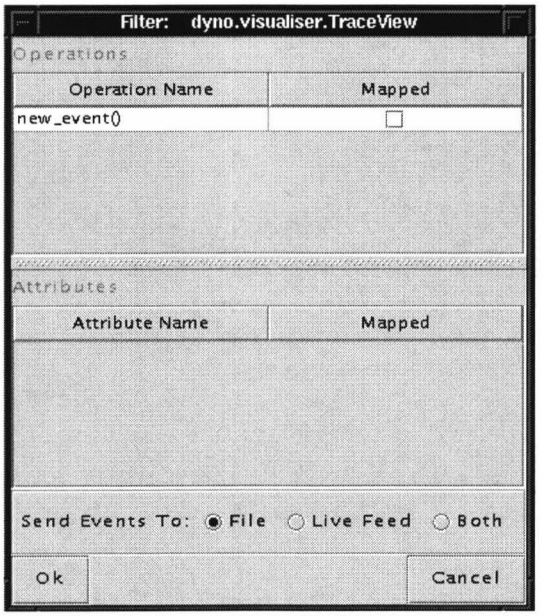

Figure 7.7: Having selected the TraceView, the reuser is presented with a list of the attributes and methods in that view. These must then be mapped to some counterpart in the entity under inspection.

mation. Some view methods may not be required. Not all view methods need to be utilised during a test drive for a complete animation to result. A Stack view with methods for *push* and *pop* might only have its *push* method used during a particular use of a stack assembly. Likewise, a B-Tree view may never have its *remove* method used during the test driving of a B-Tree assembly. A B-Tree view could have a *remove* method for animating removals, but the B-Tree assembly the reuser is test driving might not have functionality for removal. For this reason, not all view methods need to have something mapped to them. This is not true for the attributes however. Since the attributes are accessible by all the view methods, there is no easy way to tell what attributes might be unnecessary for a particular use of a view. An unmapped attribute that is then referenced by a method could result in a possible null reference exception, depending on how careful the animator was in pre-checking the validity of the attribute values. Dyno chooses the safe option of requiring that all attributes be mapped to something before it will use the filter and create an animation.

Mapping to View Methods

As figure 7.7 shows, TraceView only has one method, *new_event*, that needs to be mapped. Remembering back to earlier in this section, an action can be the invocation of a method, the return of a method, the accessing of a field, or the modification of a field.

Double-clicking on *new_event()* will result in the reuser being presented with a window

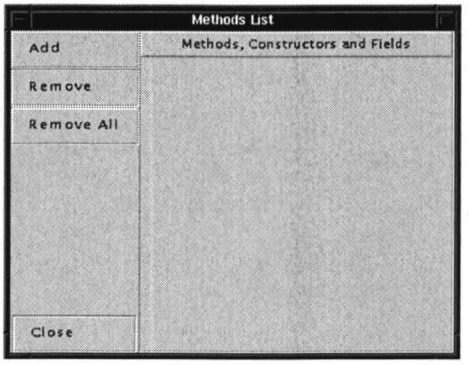

Figure 7.8: This lists the current mappings for the selected method in the selected view.

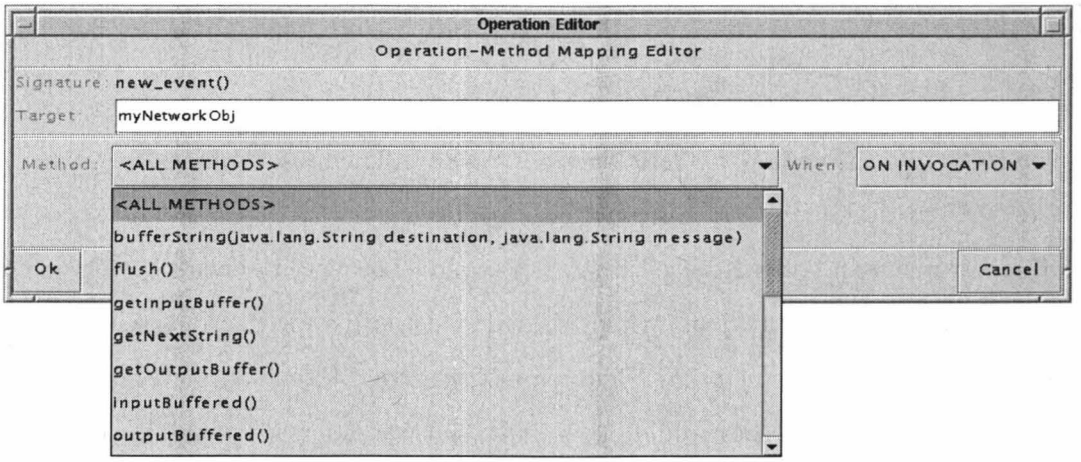

Figure 7.9: This interface allows the reuser to specify a mapping to a view method from methods of some entity in the object cache.

through which they can add actions to a list stored against this method, as seen in Figure 7.8. The list is currently empty and there are buttons on the left hand side that allow the reuser to manipulate the list. The reuser decides to add a new action to this list and so clicks the *Add* button.

The reuser is then presented with a window for entering information regarding the action, as shown in figures 7.9 & 7.10. Actions are typically connected with a single entity in the object cache. The name of the entity is entered in the top-most text field. Had the reuser not entered in a name and left the field blank (i.e. a NULL value) then the system would have assumed that events matching this action should be passed to the visualiser regardless of the object it occurred on.

There are also two check buttons that enable the reuser to switch between specifying a method-based action (invocation or return of a particular method, or all methods) and a field-based action (access or modification of a particular field, or all fields) on the object specified in the text field, or all objects if the text field is empty. Depending on which check button is enabled, the two popup menus also in the window will either list the methods and types of method actions (see Figure 7.9) or will list the fields and types of field actions (see Figure 7.10).

The reuser has one value currently stored in the object cache, called *myNetworkObj*, and so types this name in at the prompt, hitting the return key upon completion. Having done this, Dyno updates the two popup menus currently on display. In this case, they are the method-based menus and the first popup menu now lists all of the methods that may be invoked on the *myNetworkObj*, i.e. all the instance methods. The second popup menu gives three options, "On Invocation", "On Return" or "Both". The reuser decides they want a new event to be signalled in the TraceView when the *flush()* method returns, so selects *flush()* from the first menu and "On Return" from the second. The reuser now clicks on *Ok* and the window disappears. The previous window that listed the mapped actions will be updated to include an entry for this mapping. The reuser has second thoughts, however, and decides that rather than just *flush*, they would like to have the TraceView know about all the method invocations and returns on *myNetworkObj*. The user could add a second mapping, but since the first isn't needed, they may as well modify that (or remove the first and then add a new one). The user double-clicks on the entry for the previously created mapping. This brings the operation-method mapping editor (refer to Figure 7.9 or Figure 7.10) back up, with all the values set as the reuser had specified. The reuser now pulls down the methods list and

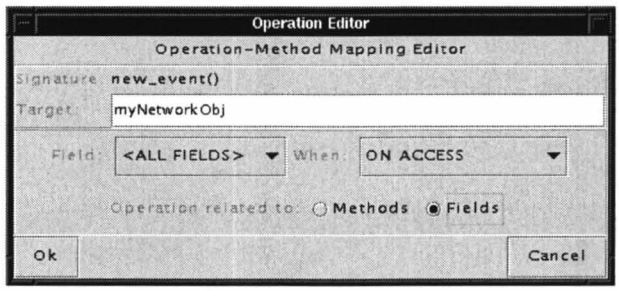

Figure 7.10: This interface allows the reuser to specify a mapping to a view method from events on fields on some component, or component instance, in the object cache.

rather than selecting *flush()*, decides to select the top-most item *<ALL METHODS>*, which is a special entry and not a method name. The reuser also changes the "when" condition from "On Return" to "Both". Clicking on *Ok*, the entry in the list of mappings now changes accordingly.

Clicking on *Ok* in the mapping list window will return the reuser back to the original Filter window that listed the actions in the view, and any of its attributes (refer to Figure 7.7). As we have mapped something to *new_event()* there is now a tick against it.

As there are no attributes to map, the reuser proceeds to select where they would like the information that the system collects to be stored. The options are: to a file whose name will be asked for later, a live feed that opens up a ViewScreen directly and pipes the information to the animation code as the component is running, or both. The reuser selects the *file* option and then clicks on *Ok*. Dyno will then open a file browser window and ask the user for the name of the file they wish the information to be saved to. The user types in "mySaveFile" and clicks *Ok* on the file browser. At this point the reuser has now successfully given Dyno sufficient information to be able to animate the component based on a sequence of the user's interactions with the component.

Mapping To View Attributes

If the view had attributes, then they would be listed by the name the view writer had given them, the type of data needed by that attribute (whether a *java.lang.String*, *java.util.Vector* or some other object or primitive value) and a boolean box which shows a tick when something has been mapped to that attribute by the user.

7.6.2 Creating a Sequence of Actions

The reuser next decides to put the animation to the test and invokes a few methods that they hope to see animated. Before doing so, the reuser must signal to Dyno that it should start detecting events. Dyno does not attempt to detect events all of the time, as the reuser may want to use Dyno for a period of time without sending information to the animations. The reuser turns on event detection by going to the *Visualiser* menu and setting *Event Trace* to be on. A tick will appear next to it when it is on. Having done this, Dyno is now waiting for events to detect. Using the same procedure as before, the reuser invokes a number of methods on *myNetworkObj*, passing arguments through and saving return values as necessary. All the while, Dyno is storing all the relevant information on this sequence of interactions to the file "mySaveFile" for future use by a ViewScreen. Once the reuser has finished with this particular test drive, they can turn the event detector off by de-selecting the same menu item. Upon being turned off, the event detector will signal any filter that has had information passed through it since the event trace was turned on last. Each filter that has seen an event significant to its animation will then ask the reuser for a name for that sequence of events. Since more than one animation can be stored per file, this name is used later on to uniquely identify animations. The reuser types in "aSequence", clicks *Ok* and now has all the information that is required for Dyno to create an animation.

7.7 Viewing the Animation

The next step for the reuser is to view the animation. They do this by opening up a ViewScreen. To open a ViewScreen, the reuser selects *New ViewScreen* from the *Visualiser* menu. This now brings up a viewscreen, as seen in Figure 7.11. The viewscreen works on a tape deck analogy, where by the reuser has control over the chronological direction and speed of the animation via rewind, play, stop and fast forward buttons. The viewscreen has a main window where the animation will be drawn, that may be scrolled if the animation is bigger than the window size. A progress bar shows the reuser how far through the animation they currently are. The last thing on the viewscreen is a popup menu that lists all of the sequences in the currently loaded file. No file is loaded at this point in the example.

The reuser decides to load the previously created "mySaveFile". The reuser does this by selecting *Open File* from the *File* menu in the ViewScreen. The *Open File* menu item will open up a similar file browser as was seen earlier in the session when the mapping filter

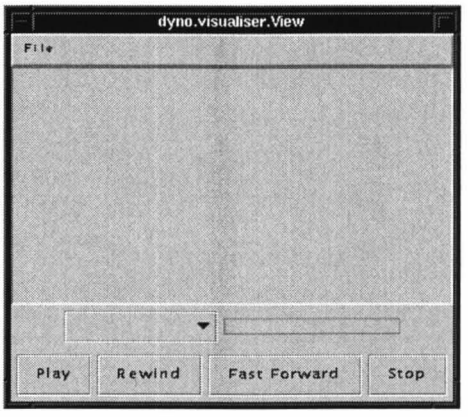

Figure 7.11: A blank ViewScreen, with no loaded file.

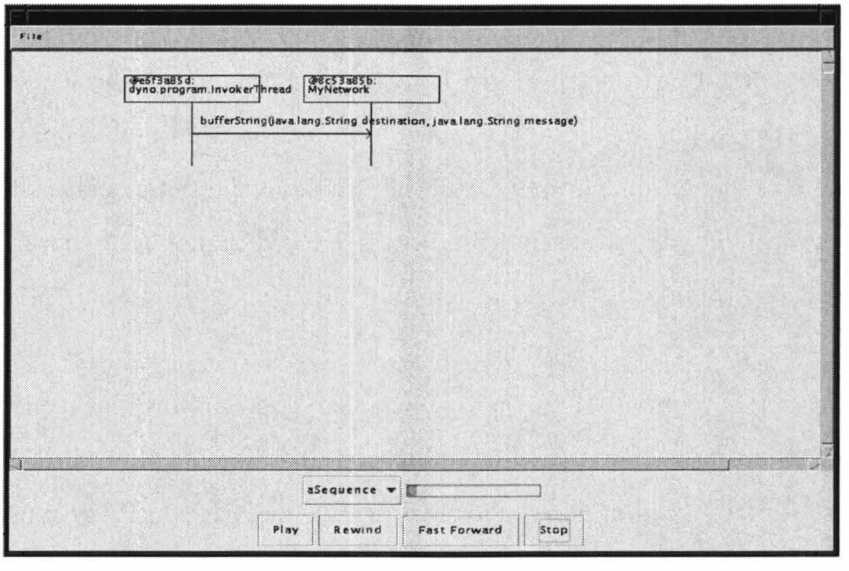

Figure 7.12: Early on in the Trace of the myNetworkObj. The trace is described in terms of a UML style sequence diagram.

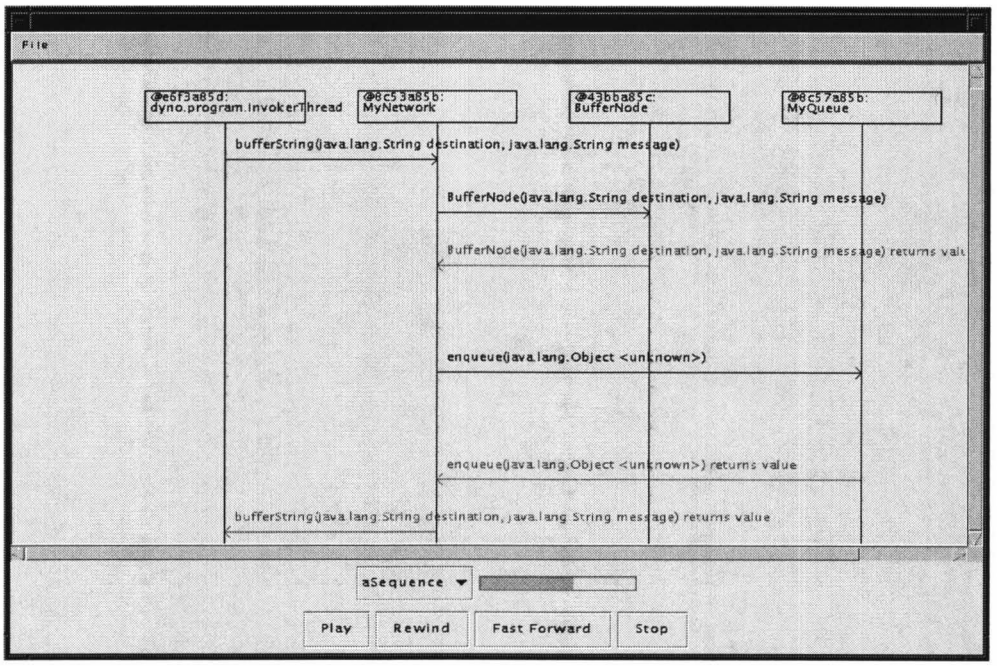

Figure 7.13: Later on in the Trace of the myNetworkObj. By this point several methods have been invoked and returned. The darker lines indicate a method invocation and the lighter lines indicate a return.

asked for a file to save to. Selecting "mySaveFile" from the directory the reuser put it in, the reuser then selects the *Scan File* menu item from the *File* menu. This retrieves all of the sequences that are stored in the file. The reason for having a separate option for scanning is so that the file can be appended to by Dyno while the reuser is viewing animations from it, and the new sequences can be retrieved without it being reopened. The sequence names will then be put into the popup menu on the ViewScreen display. As there is only one sequence in the file, that one sequence is automatically selected and the reuser may now play, rewind, fast forward or stop the animation at will. Figure 7.12 & Figure 7.13 shows the animation in action at different points in the execution. The animation shows a UML style sequence diagram representation of the execution trace.

The reuser can have more than one ViewScreen open at any one time. This allows the reuser to concurrently view the same animation at different points in the execution, or view completely different animations. This can be especially useful if the reuser has created more than one visualisation from a single test drive, and wishes to compare the different visualisations on an event by event basis. An example of creating two visualisations from a single test drive would be to have a QueueView of *myNetworkObj*'s Queue private fields, as well as showing the events in the UML-style TraceView. The reuser could use the *View Controls* sub-menu in Dyno's Visualiser menu for sending a single play, rewind, fast-forward

or stop command to all open views.

Dyno also allows the reuser to save or load the contents of the object cache by using the *Save Cache* and *Load Cache* menu items in Dyno's *File* menu. This opens up the standard Java file browser window, and the reuser selects a file to save to or load from in the usual manner. The object cache will only store those objects that are serialisable, so some parts of the Cache may be lost between separate sessions. At the moment the reuser is not warned that they may potentially lose objects from the cache in between saving and loading to and from file.

Once the reuser has finished with either a particular ViewScreen, or with Dyno, they can select the *Exit* menu item in the respective *File* menus.

7.8 Assembly Specific Views

The example we have covered in this chapter demonstrates a generic view that could be used with any Java class. Dyno also supports the use of assembly-specific views. One such example would be a tree view, that would work with assemblies that represent tree-like data structures, but would be unable to usefully work with other data structures. These assembly-specific views can create a more detailed visualisation than would ordinarily be afforded by generic views, as the latter cannot make any assumptions about the structure of the assemblies they will be used with.

With these assembly-specific views, there would be more effort required in mapping the state of the tree assembly to the attributes on show in the tree view. Since these tree structures are usually recursive in nature, Dyno allows a reuser to simply identify where the root node of the tree is, what elements of that node act as the children, and what elements act as the values. Assuming that all the nodes in the tree are of the same general type, it will then recurse down the tree structure every time that an event is detected, and copy all of the data over to the visualisations, where the animator will have coded the view to use this information to draw a tree diagram, such as in figure 7.14.

There are no explicit limits on the types of assembly-specific views that can be created. During the development of Dyno, we also created stack and graph views. The ability to handle graph views demonstrates the ability to handle data structures that may contain loops. When traversing the graph, so as to serialise its data content for transfer to the view, Dyno will recognise any nodes it has already traversed, and not loop through a second time.

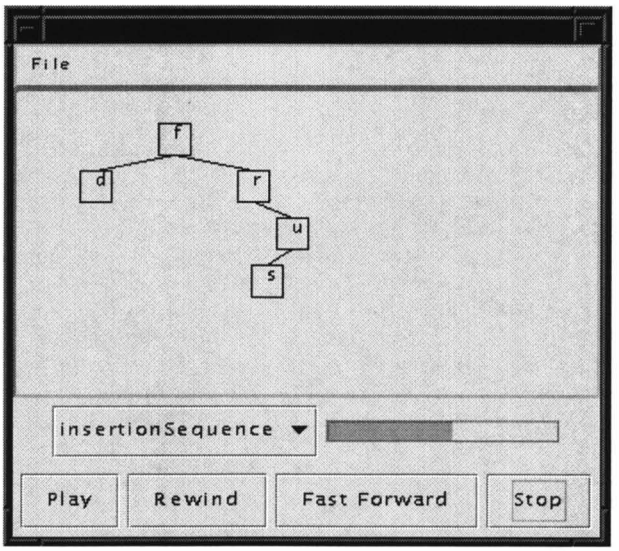

Figure 7.14: An example of a simple Tree View in action, showing the standard tree representation.

7.9 Summary

In this chapter, we have demonstrated how a reuser might typically interact with Dyno. In our example, the user wishes to understand what the *MyNetwork* class does, and how it does it. The user first has to load the class into the tool, possibly having to modify the classpath so that Dyno knows where the class is. The user then proceeded to create instances of the class, invoke methods on these instances, and inspect their state. This allowed them to see how the methods were invoked, what their side-effects were, and whether the order of invocation effected on the results (and if so, how). The test driving of the *MyNetwork* class laid the foundation for an understanding of the assembly's functionality and structure. The user also wanted to create animated documentation of the class, in the form of a UML-style sequence diagram. The test driving of *MyNetwork* served as the motivation, and a source of information, for the sequence diagram. The events resulting from the test driving caused changes in the sequence diagram, and the user supplied some limited information that described the mapping from assembly to visualisation.

In general, test driving and animations help the user develop an understanding of a class, possibly by complementing knowledge accessible by other means. The user can see what is possible with a class, and how the class stores its state, as well as how the state and the methods interact. With a better understanding of what a class does and how it does it, the reuser should have a better understanding of the possibilities for its reuse.

Chapter 8

Evaluation and Discussion

In chapters 2, 3 and 4, we discussed some of the issues involved in code reuse, such as the impact of cost on the application of code reuse principles. We also discussed a possible approach to reducing this cost, by helping a reuser to understand an assembly through the use of tool support. In chapters 5, 6 and 7, we discuss the requirements for a tool that would support understanding, the design and implementation of one such tool, and a demonstration of its use. In this chapter, we evaluate the tool seen in chapter 7, with respect to the criteria laid out in chapter 5. We discuss how, if at all, this tool helps promote understanding, and therefore code reuse. We also discuss some of the technologies that are of use in the creation of tools meeting chapter 5's criteria. We then move on to discussing the general issues in test driving and visualisation, and how they can help promote understanding.

8.1 Fulfilling Criteria

One reason for designing and implementing Dyno was as a proof of concept for our research. We hoped to show, through Dyno, that tools could support the understanding of assemblies, and in doing so reduce the cost of understanding, and promote code reuse. We now evaluate Dyno and discuss its success relative to our criteria. First, we re-cap the criteria from section 5.5:

1. Access to assemblies:
   (a) Assembly access should at least be at the level offered by manually written test harnesses.
   (b) Values accrued during test driving can be stored for later use or inspection.

(c) Interactions can occur multiple times, with the option of using a range of values to change the state, or parameters, on each iteration.

2. Required changes to an assembly:

   (a) There should be no required changes to the source code of an assembly.

3. Visualisation capabilities:

   (a) The graphical vocabulary should be extensive enough to be able to create useful visualisations.

   (b) The tool can be extended to include assembly-specific visualisations.

   (c) Time and effort required in writing visualisations should be minimised.

   (d) Any reuser interaction with respect to mapping between assembly and visualisation should be minimised and as simple as possible.

4. Learnability, Portability & Scalability:

   (a) The tool should not require a large amount of time and effort be spent learning how to achieve meaningful goals.

   (b) The tool should be portable.

   (c) The tool should be able to handle realisticly sized assemblies.

8.1.1 Access to Assemblies

Dyno permits a reuser to both access the public interface of a class, and also directly access its protected and private parts (see criterion 1(a) above). Access to the class includes the ability to invoke the methods individually, and also to inspect and modify the fields of that class, or instances thereof. A reuser supplies any required parameter values when invoking a method, and may store the return value. This return value can then be further inspected, or used in later invocations (fulfilling criterion 1(b)). By allowing direct access to the protected and private parts of a class, Dyno offers a greater level of access than would be afforded to the reuser when invoking it in the context of the new project. This extra access can help a reuser to identify specific parts of the design and see how the methods and fields combine to achieve certain functionality. It allows the reuser to form a greater understanding of how the state of an object is stored. This may be useful if the nature of the state affects its suitability for reuse, such as if the new project requires an assembly that uses dynamic storage, or if the

state of an assembly includes data structures, such as an array, that are too large and would, if used a significant number of times, lead to significant memory requirements. This breaks encapsulation, and adherence to proper encapsulation is often identified as good design, as it limits the dependencies on modifiable implementation. However, once the assembly is used outside of Dyno, the reuser must return to obeying the standard access permissions. Therefore, the reuser cannot form potentially dangerous dependencies on the assembly's implementation, that could otherwise lead to maintenance problems later on.

One feature that is not explicitly supported however is the multiple invocation of a method, iterating over a range of values (criterion 1(c) above). An assembly programmer could supply a second class, along with their assembly, that demonstrated this behaviour, and from which visualisations could be created. However this is left up to the assembly programmer rather than directly supported through the Dyno tool. This was left out primarily due to time constraints, and to simplify the design and implementation of Dyno.

8.1.2 Required Changes to an Assembly

The range of Java assemblies that can make full use of Dyno's features are only really limited by a single requirement. This requirement is a result of the inability to easily extract method return values using JVMDI. It only affects the ability of some assemblies to have their method returns detected, and does not invalidate the detection of the other types of events Dyno currently supports. The requirement is that an assembly should be annotated with calls to one of the static *Send* methods of the *Return* class, just before a *return* in the source code. The value to be returned is passed through as an argument to the *Send* call. Even if the assembly is not annotated in this manner, the assembly may still be test driven in Dyno. The return value of the user-selected method can still be retrieved and stored. However, return values can not be detected for the other methods invoked as a consequence of the user-selected method. Therefore the visualisation capabilities are reduced by removing some method returns from the set of detectable events. As already mentioned, other types of events can be detected without problem, and visualisations can still be created from these.

A second, less significant, requirement relates to the method Dyno uses to recover the names given to method parameters in the source code. These names are not the type names, but rather the variable names. To retrieve these variable names, Dyno uses debugging information in the Java *.class* files that is only present if the class was compiled with the debugging flags on. If this debugging information is not present, then Dyno can still determine the types

of the parameters. The names of class fields are present irrespective of whether or not compiled debugging information is available. If no parameter name can be found, Dyno displays the name as "unknown". This minor requirement has no affect on whether or not a method can be test-driven, or on the visualisations that can be derived from such a test drive. It only affects how much information the reuser is given with regards to what the parameters are for.

8.1.3 Visualisation Capabilities

Animators are responsible for creating the views used by Dyno. A view's methods create distinct animations for particular types of events. The creation of views raises two issues (criterion 3(a) above). Firstly, the vocabulary that the animator can use; and secondly, the types and amount of information that can be visualised. Views in Dyno are implemented using the standard Java graphical packages, for example: *AWT* or *Swing*. All the views inherit from the class *dyno.visualiser.View*, that in turn inherits from *java.awt.Canvas*. The animator is restricted to writing to the canvas, but has a reasonable vocabulary to choose from, including all usual geometric shapes, text and image processing capabilities. With regards to the second issue, Dyno sends information from the mapping component to the visualisation by serialising Java objects. Since not all Java classes are serialisable, this places a limit on the objects can be passed through to the views, which in turn places a limitation on the type of information that can be given to the views, and therefore the expressiveness of the visualisations. Another limitation on the visualisation capabilities is the amount of information that is gathered at runtime. This relates both to the nature of the detectable events, and also what information is gathered when those events occur.

Another measure of Dyno's visualisation capabilities is the degree to which they can be extended to handle new types of visualisations (criterion 3(b) above). Dyno's visualisation capabilities are extendible insofar that new views can be imported into the tool. This permits reusers to see visualisations that we have not thought of during Dyno's development. It also supports animators in being able to modify and improve on visualisations, so as to react to any weaknesses and strengths identified during a view's use. The views implemented by animators must conform to some standards, such as extending *dyno.visualiser.View*. The animator must also identify what attributes the animation needs to have mapped from the loaded assemblies, and does this by invoking a predefined library of methods. Beyond this, there are no major restrictions on the types of views that can be incorporated.

The time and effort involved in writing a view is reasonably small (criterion 3(c) above). The trace view shown in chapter 7 was approximately 120 lines of simple Java, making use of the basic line, box and text graphical constructs available in the Java AWT. Most of those 120 lines were in fact taken up drawing the arrows to point in the right directions. The tree view only required approximately 60 lines of Java code, and again only needed to use the basic graphical constructs that programmers who have used the Java canvas should be experienced with.

Lastly, with regards to criterion 3(d), the reuser does need to specify some information to map the assemblies to the visualisations. This information describes what events in the assemblies correspond to what events in the visualisations, and also what state in the objects corresponds to the visualisations' attributes. We have attempted to make this interface as user friendly as possible, without overly restricting the kinds of mappings that can occur. We have also added support for automatic translation between similar types (similar in a logical sense, such as converting arrays to vectors for example) so as to add further flexibility in what can be mapped to what. We will discuss this mapping process further in section 8.2.

8.1.4 Learnability

We identified learnability as an important criteria for our tool. With this in mind, we have performed further studies on the learnability of Dyno, and detailed discussions of these studies can be found in section 8.2.

8.1.5 Portability

Reuse issues are not unique to any one environment, and tool support that can work in different environments will be able to help more programmers, and in more situations. Dyno uses Java for most of the implementation. Java benefits from storing its executables as byte codes, that are then run on a virtual machine. This virtual machine can be, and has been, ported to most commonly used machine architectures. Therefore any environment for which there is an implementation of the Java Virtual Machine, should be able to execute the majority of Dyno's code without any modification. However, as well as using Java, Dyno uses the C language in the form of the Java Virtual Machine Debugger Interface (JVMDI). One of the reasons for using C code with Java is that the implementation can make use of machine specific features for efficiency reasons. These machine specific features are normally off-limits to Java. The use of the JVMDI means that if the tool is ported to another architecture, then

the C code must be rewritten and recompiled. These code modifications are the reason that Java removes the ability to use machine specific features, and is one of the reasons why it is so portable. The portability of Dyno is therefore the portability of its least portable component: the JVMDI code. Fortunately, C compilers are very common, and the JVMDI code only makes use of standard C libraries, along with extra library support that is distributed in the standard Java Development Kit (JDK). This helps to reduce the difficulty in porting Dyno.

8.1.6 Scalability

Scalability is the measure of how the tool behaves as the size of the assemblies increase. The example given in chapter 7 was not a large class. However, there is no practical limit on the number of methods or fields that Dyno can display, although the display update slows down the more methods and fields there are in the tables. Several of the JDK classes have a large number of methods and fields (especially the *AWT* classes), and these can be used in Dyno, demonstrating its applicability to handle large classes. Huge classes that have very high memory requirements (for example: huge arrays in class fields, or large local variables in methods) would run into the same kinds of problems in Dyno that would be encountered during normal use in projects.

If requested to by the reuser, Dyno will write the filtered runtime information for a visualisation to secondary memory. There is therefore a restriction that there be sufficient disk storage to store this information. There is a similar restriction that there be sufficient primary memory to store the filtered information as it is used to create the individual frames for the visualisation. The images used in the visualisation are not stored or cached, so that an individual frame is redrawn from the stored information each time it is needed. This has the disadvantage that more processor time is required for the redrawing, but has the advantage that only the filtered information needs to be stored, rather than a potentially large sequence of images. An image typically requires more space than the filtered information from which it is drawn.

As the size of the classes increases, it becomes more difficult for a reuser with little background knowledge to know what methods to call first, and in what order methods should be invoked to achieve a particular result. With this in mind, it is possible for assembly programmers to actually supply test harnesses along with their assemblies. These could then be used by a reuser to see what methods should be called when, by referring to visualisations created from the execution of the test harness. The test harness could be loaded into Dyno,

and invoked, just like a normal assembly.

8.1.7 Summary

Dyno succeeds according to most of the criteria developed for these kinds of tools. There are one or two areas for improvement, such as removal of the need for annotation with respect to accessing the return values of methods. Overall however, Dyno has demonstrated that tools can be implemented to fulfill these criteria.

8.2 Evaluation of Learnability

We have identified easy learning as one important requirement for a tool aimed at reducing the cost of understanding. To evaluate Dyno's success at meeting this requirement, we use the usability inspection method known as the *cognitive walkthrough* [WRLP94]. We first describe what a cognitive walkthrough is, and we then describe an actual cognitive walkthrough that we performed during Dyno's development.

8.2.1 Overview of Cognitive Walkthroughs

In section 5.5, we mention that it should be easy to learn how to properly use a tool. This reduces the overhead in time and effort, and means that the overall time and effort saved by using the tool is increased. The degree of time and effort required to learn how to achieve meaningful goals with a tool is referred to as the tool's *learnability.* One usability inspection method that specifically measures the learnability of a tool is the cognitive walkthrough. A cognitive walkthrough is the inspection of a tool's user interface by observing what happens during a typical, realistic interaction. In a cognitive walkthrough, the tool's inspectors describe a typical user, choose a task the user might attempt and note a correct course for the task. The evaluators then discuss what a user would do if he or she were trying to complete the chosen task. The evaluators decide what the user would do at each stage, on the basis of what the user currently knows and sees. This is then compared with the correct course of action. In a good interface, the user would perform the correct sequence of sub-tasks to achieve the desired result, and would perform this sequence because it was clearly indicated by the interface to be the correct choice.

Having briefly described the process of a cognitive walkthrough, we now discuss one such walkthrough that we performed using Dyno. We performed two cognitive walkthroughs during the duration of the tool's development. The first walkthrough led to several improvements

to the interface, including a clearer naming scheme and improved feedback to the user. The second is presented below.

8.2.2 Cognitive Walkthrough of Dyno

Our cognitive walkthrough is based around the example interaction described in chapter 7. Here we focus on the learnability issues that arise as the interaction proceeds. We first review the various roles and goals of this walkthrough.

The User

The user is assumed to be a programmer. The programmer is searching for an assembly to reuse, and would have some experience in the Java language, or at least some knowledge of the OO programming paradigm. A user with this knowledge can then be expected to understand the concepts of class, type, method and signature, that are used throughout the interface. The programmer would also be expected to understand how classes are organised into the Java inheritance hierarchy, although not necessarily where each particular class appears in this hierarchy. The user is assumed to have good knowledge of how typical mouse and keyboard interactions occur.

The Task

The task, as demonstrated in chapter 7, is to gain an understanding of the *MyNetwork* class. This understanding is gained through a combination of test driving the class, and creating visualisations of the class in action.

Correct Course of Action and User Analysis

We now intersperse a description of one correct course of action for fulfilling this task, with a story about how the user reacts at each stage. In the latter case, the emphasis is on what the user would do when presented with the interface in a certain state, and whether or not the user would take the correct action. Since this example has already been covered in chapter 7, our walkthrough will not repeat all the details but merely concentrate on the learnability issues. The analysis of the users actions consists of asking four questions at each stage in the interaction [WRLP94]. These questions are:

- Will the user try to achieve the right effect?

- Will the user notice that the correct action is available?
- Will the user associate the correct action with the effect trying to be achieved?
- If the correct action is performed, will the user see that progress is being made towards solution of the task?

Step	Correct Course Of Action
1	start the application.
2	select the *Edit Classpath* menu item from the *File* menu.
3	type in "http://www.mcs.vuw.ac.nz/~stuart/" as the URL location of the root of the package hierarchy.
4	select the *Load Class* menu item from the *File* menu.
5	type in *MyNetwork* as the fully qualified class name.

The user starts out by executing Dyno (step 1). Since the user is assumed to be a Java programmer, they are assumed to know how to start applications. This means they should know what effect is to be achieved, how it can be achieved, recognise the use of a Java interpreter as achieving this, and recognise the application when it appears. Steps 2–5 require the user to load in a class. This potentially requires them to modify the classpath so that the class can be found by Dyno's search procedures. The right effect is the modification of the classpath, and then the loading of the class. This modification and loading requires the use of menu items in the *File* menu. There are only two menus, *File* and *Visualiser*. The user is a programmer, and presumably has used tools that have menu bars. There no visible buttons or icons in the display for modifying classpaths, or loading classes, so the user would then inspect the next possibility: the menus. The *File* menu is the first menu they would encounter going left to right (the direction English-speaking people read). It is also a common name for a menu that offers loading capabilities. There are clearly identified menu options for editing the classpath and loading classes. Since the menu item for adding to the classpath appears above the menu item for loading a class, if the user is scanning from top to bottom they will encounter the former menu item first. Since the user is a Java programmer, they presumably have some knowledge of how classpaths work, and why. With this knowledge they should be able to associate the need to check the classpath before loading a class, and therefore would select the correct course of action. When adding to the classpath, the user is presented with a simple interface with clearly identified fields, that should cause no immediate concerns. Having added any required URLs to the classpath, the user should then

continue down the *File* menu. They should then discover the *load class* menu item, and from this name be able to immediately associate this menu item with their immediate short term goal. The *load class* feature prompts the user with an uncomplicated, clearly sign-posted query for the fully-qualified class name. Fully qualified class names are something the user should be familiar with from past experience of Java code, so they should have no problems determining the correct course of action. When this course of action has been completed, the name *MyNetwork* is visible in the object cache browser, and a dialog box appears informing the user that the class has been loaded. These should be sufficient indication to the user that they have successfully completed this part of the overall task.

Step	Correct Course Of Action
6	select the newly created *MyNetwork* node in the object browser.
7	double-click on the one argument constructor in the methods table.
8	type the name of a host machine, such as "mcs.vuw.ac.nz", into the table entry for the constructor's argument. Then click on the *Next* button.
9	click *Ok* to confirm execution of this method.
10	type in "myNetworkObj" as the name of the new object, and click *Ok*.

The next steps (steps 6–10) deal with the creation of a new instance of the *MyNetwork* class. The user must first identify the correct course of action to display the interface of the *MyNetwork* class, and then be able to invoke a constructor. Since the user knows Java and realises that objects are created by method or constructor invocations, they would be looking for some way of invoking a method or constructor on the class. Since there is no listing of the methods and constructors currently on screen, the user would probably be trying to get such a listing. The name *MyNetwork* has appeared in the object cache browser, and is, so far, the only occurrence of the name in the interface. With this in mind, the user would typically click on this name. This (correct) course of action results in the entity information browser displaying the static methods and static fields of the *MyNetwork* class. This matches what the user would expect to need to be able to invoke a method or constructor. The listing is immediately obvious in the display, as the entity information browser is a significant portion of the screen space required. Having then been shown this listing, the user should be able to identify the name of the constructors given that the information is displayed in a manner consistent with its layout in ordinary source code. The next step is slightly harder

conceptually. Having brought up a list of the constructors and methods, the user must then be able to correctly determine how to invoke a constructor.

The user is expected to know that constructors have the same name as the class, and is expected to be able to find this in the listing. Having done this, it is likely that the user will then attempt to invoke the listed constructor. The correct course of action is to double-click the table row holding that constructor. The user might first inspect the menus, or search the interface for an explicit *invoke* button, but they will not find anything. At this point, they may then associate invocation of a constructor with *opening* that constructor. It is common in interfaces to be able to open items of interest, whether in a list or table, to display more information. The user, attempting to find more information regarding a constructor so as to be able to invoke it, will then double-click on the constructor name. This, as previously identified, is the correct course of action for invoking the constructor. At this point, the interface responds by displaying a new window, asking for parameter values to be specified. Since the user is familiar with having to pass through arguments, the fact that they are being asked for values should indicate that they have chosen the correct course of action. From this point on, the correct course of action is to type in either a literal value or the name of an already existing object. This is not obvious from the interface, given that only a text box and the parameter name and type are shown. However, a programmer should have experience of being able to use variable names and literal values in invocations. Since the names in the object cache browser relate to class or variable names, then the user may be able to draw this comparison, and know what is expected of them. Assuming the user realises that they have the option of typing in a literal, and correctly performs step 8, then the constructor will execute and a new object is created.

The return of the new object is signaled by a new prompt that tells the user what has happened, and requests a new name. The user should know that the side-effect of invoking constructors is the creation of an object, and that these objects must be given names (analogous to variable names in source code) and should be able to identify this as the proper result of a constructor invocation. Having typed the name *myNetworkObj*, the name appears in the object cache browser. This is similar to how the class name appeared when it was loaded, so the user should have no problem realising that the object has now been stored.

Step	Correct Course Of Action
11	select *New Animation* from the *Visualiser* menu.
12	type in *dyno.visualiser.TraceView* into the text box and press the *Add to List button.*
13	double-click on *dyno.visualiser.TraceView* in the list box.
14	double-click on *new_event* in the operations table.
15	leave the *target* text box next to the label "target" empty.
16	select *ALL METHODS* in the *methods* combo box.
17	select *BOTH* in the *when* combo box.

The user now needs to create an animation of them test driving *MyNetwork*. The correct course of action is to first supply Dyno with the information needed to map the assembly to the visualisation. This is achieved by selecting *New Animation* from the *Visualiser* menu. The user is not explicitly told that this step is necessary, although we assume our user knows enough about our tool that they realise some form of mapping is required at some stage. Since animations should be associated the term *visualisation*, it is not unreasonable to assume that the user will browse the *Visualiser* menu when thinking about creating an animation. If they do this, then they will encounter the *New Animation* option, that should then be associated with their current sub-goal.

Selection of the *New Animation* menu option will then cause a new window to appear. This new window will list all of the currently loaded views, and one of the loaded views is *TraceView*. For the sake of this example, we assume that the user wishes to use a *TraceView*. The user should already be familiar with the idea that clicking on a name should make something happen, as demonstrated by their invocation of the constructor in the earlier steps. Since this new window is also list-like in structure, and there are no other immediately obvious interface features with which to interact, the user might therefore decide to click on the word *TraceView*. If they do this, then a second new window appears.

The second new window shows information regarding the operations and attributes present in the selected view. For *TraceView*, the attributes table will be empty. The operations table shows one element, called *new_event*. There is also a tick box next to the name, and no tick currently present. The correct course of action is to map assembly information to this operation. The lack of a tick in the tick box should indicate to the user that this operation has not been successfully mapped yet. To map information, the user needs to click on that table row.

Again, the user should be familiar achieving results by clicking on table rows, from the way they previously invoked the constructor. If they do click on the row, then a third new window appears, and this window lists all of the current information mapped to that operation. As no information has been mapped, the list is empty. Next to the list are several buttons, one of which is *Add*. Since the user has identified that they need to add mapping information, they should then be able to make the connection between their desired goal, and the need to select this button.

Selecting *Add* will cause a fourth new window to appear, and at this point the user can finally add mapping information. To add the information, the user is presented with a selection of text boxes and combo boxes, all identified by having names next to them. The user has, for the sake of this example, decided that they want to have all events that occur on any object appear in the visualisation. Since the interface has asked for the name of the object on which events are to be deemed relevant, the user decides to leave this text field blank. This is the correct course of action, although there is no immediate indication of this. Secondly, they can decide what methods are to be deemed relevant to the visualisation. Since they have not specified a particular object as being the target object, then the only option is to select *ALL METHODS*. Progressing down the form (as you would if you were filling out any other kind of form, either computer based, or in the real world), the user is then asked if the event should be deemed relevant when the method is invoked, or when the method returns, or both. For the sake of our example, the user selects both. This is the correct course of action.

Step	**Correct Course Of Action**
18	click *Ok* in the operation mapping editor.
19	click *Close* in the *Methods List* window.
20	select the *file* radio button in the main Filter window.
21	click *Ok*.
22	use the file browser to find the right directory, and then type in "animationFile" as the name of the file to save animation data to.

Having told Dyno the information needed to map events and data to the visualisations, the user should now be thinking about test driving. They must first confirm all the entered information. Given that *Ok* and *Close* are commonly understood buttons, the user should have no problem with identifying these buttons as necessary to closing the currently opened windows. After having closed the windows, in the customary manner of closing windows in

reverse order to that they were opened, the user will end up back at the main Filter window.

The user can specify where they want the visualisation data sent to. Either they can have it played through a live animation feed, be saved to a file, or both. The option to save it to file is selected by default, so that the user does not strictly have to worry about this step, unless they wish to have the animation running concurrently to the test driving.

Our user decides to save the visualisations directly to file. Leaving the clearly indicated *file* radio button activated in the Filter window, the user would then proceed, as is typical for these kinds of interfaces, to select *Ok*. Once *Ok* has been selected, Dyno will identify that the user wishes to save to a file, and will prompt the user to specify what file to save the information to. This is achieved by a file browser window, the style of which should be familiar to anyone who is reasonably experienced with using a computer. Once the user identifies the file that the visualisation data should be saved to, the Filter window disappears. There is no explicit notification that Dyno has stored the mapping information, so the user is assumed to understand that the disappearance of the window without further dialog indicates success. This is not unlike other interfaces, so again should not be too much of a problem for experienced computer users.

Step	**Correct Course Of Action**
23	turn *Trace Events* on in the *Visualiser* menu.
24	select "myNetworkObj" in the object browser.
25	invoke the *bufferString* method by double-clicking.
26	enter the name of a destination and message to send, and click *Ok*.
27	confirm that the method should be executed.

The user has given *Dyno* information regarding events and data that are of relevance to the desired visualisations. The user now needs to test drive the class, and from this will be formed the visualisations. A first time user might attempt to directly invoke the methods of the *myNetworkObj* immediately, in much the same fashion as the constructor was invoked on the *MyNetwork* class. Unfortunately, while this will achieve the goal of test driving the class, it will not create the visualisations. The reason for this is that the user needs to tell the tool to start using the generated events to create visualisations. Without this, the tool would have to assume that all events generated by the user were to be used in the visualisations. Since the user may want to merely test drive without changing, or adding to, the current state of the visualisations, the user has to explicitly turn the event detector on and off. Turning the

event detector off also lets the tool know when the events relevant to a particular visualisation have finished.

Unfortunately, there is no immediate indication to a new user that such a course of action is required. Once they test drive *myNetworkObj* for a while, using a process described below, then they will realise that no visualisations have been created. At this point, they should stop to explore the interface to try and determine why the visualisations have not been created. There is nothing in the *object cache browser*, or *entity information browser*, that seems related to visualisations, however there is a *visualisation* menu that the user is familiar with from the mapping process prior to this. In this menu is a clearly identified menu item *Trace Events*. The user makes the connection between tracing events, and those events being available for use by visualisations, and therefore selects this menu item. A tick then appears next to the menu item. In most interfaces, a tick is used to indicate *success*, or *activation*, as it does in this case. An experience computer programmer, such as our user, should then be able to determine that the event trace is now active. Having done this, the user will then try to create events to be visualised.

Having chosen our tool, the user is assumed to know that the visualisations are created from the test driving, and that the act of test driving is the act of invoking methods on objects, and inspecting their state.

The user has now completed the tasks for loading *MyNetwork*, creating an object of this class, and informing *Dyno* as to what events to map to the visualisations. The user has seen the general layout of the interface, and should now be familiar with the technique of executing methods. They also have experience of using the object cache browser, having used it to select the *MyNetwork* node. The user's overall goal is to test drive and visualise the execution of *MyNetwork*. In Java programming, the majority of class methods are non-static, meaning that they are invoked on objects rather than on the class itself. Constructors are, like in most OO languages, exceptions to this rule. As the user wishes to test drive *MyNetwork*, they should make the connection between how they invoke methods in their own Java programs, and the fact that they need to invoke methods on the *myNetworkObj*. They have selected *MyNetwork* by clicking on the node in the *object cache browser*. When they selected *MyNetwork*, they were then presented with information for that class, and were able to invoke methods. It is therefore reasonable to assume that our user decides to execute a method by first selecting *myNetworkObj* from the *object cache browser*.

Selecting this node will cause the *entity information browser* to be updated with the information for *myNetworkObj*. The non-static methods and non-static fields are displayed. The user knows how to invoke methods, and this is, as far as they know, the next required step in creating a visualisation. As the user has no prior experience of the *MyNetwork* class, and is not sure what particular methods do, they choose by random selection to invoke *bufferString*. Since the user already knows how to invoke a method, and pass arguments, we will not repeat this here. The argument being passed through is a string, just as before, and is passed through in exactly the same manner. Once execution of the method has completed, the user will be alerted by a dialog box. Since the return value of *bufferString* was void, there will be no return value, and the user is alerted of this fact. The dialog box indicates to the user that the method has completed, and the text of the dialog box should convince them that the method has executed correctly.

However, the user does not know whether or not they have succeeded in creating the visualisation. Remember that the user specified that the visualisation should be saved directly to file, rather than show the visualisation as the test drive took place. There is no indication that the visualisation data has been saved to the file (without directly looking at the file, or loading it up into a view screen). Therefore, while the user has undertaken the correct course of action, they may not realise this.

Step	Correct Course Of Action
28	repeat the last four steps, for two more calls to *bufferString*.
29	repeat steps 26–28 for the *flush* method.
30	repeat steps for the *receive* and *getNextString* methods.
31	turn *Trace Events* off in the *Visualiser* menu.
32	Call the sequence of events "mySequence".

A test drive may consist of more than one method execution or state inspection. Having invoked the *bufferString* method, the user may decide to perform some more invocations to further understand how the assembly works, and reacts. Since the process is similar to that already described, and the user should now be comfortable with this process, we will not repeat our discussion of how users invoke methods.

At some point, the user will wish to finish the test drive and view the resultant visualisation. Note that the tool expects to be told when it should stop using detected events to create visualisations. Once the user has finished test driving, they should then turn off the *Trace Events* option in the *Visualiser* menu. However, there is no explicit direction given

by the interface, so the user can be reasonably expected not to know that this needs to be done. However, having turned the event trace on to begin the visualisation, the user may well make the connection that turning the event trace off will finish the visualisation. This particular sequence of events in the visualisation ends once *Trace Events* has been selected. The user is likely to select the same option they selected to turn the event trace on, as the presence or absence of a tick, to indicate activation or deactivation, is a common feature of many interfaces.

Once the user has done this, they will be presented with a dialog box requesting a name for the sequence of events just detected. The user is assumed to know that multiple sequences of events can be stored in the same file, and their experience as a programmer should tell them that if this is the case, then a key would need to be stored to later refer to specific sequences. The user can type in any name they want, in this case "mySequence", and this then gets stored to file. This is the first indication that the user has been performing the right course of actions to create a visualisation, as they now have feedback that information was being saved to the file they earlier specified.

Step	**Correct Course Of Action**
33	select *New ViewScreen* in the *Visualiser* menu.
34	select *Open File* from the *File* menu of the new View Screen, and select the file "animationFile".
35	select *Scan File* from the *File* menu.
36	select the sequence "mySequence" via the ViewScreen interface.
37	play and fast forward the animation.

Having performed a test drive, the user should now view their new animation. We have assumed that the user knows that the visualisations are created by test driving, and having performed the test drive, they should now be on the look out for ways of viewing the animation. While the test driving was going on, the visualisation information was being saved directly to file, so there was no ViewScreen currently open. Since the animations are seen through these ViewScreens, the first thing the user must do is create a new ViewScreen.

The correct course of action is to go to the *Visualiser* menu, that the user has used before and is familiar with, and select the *New ViewScreen* option. This is the top most option in the menu, and we assume our user reads the options from top to bottom. With this in mind, the user would associate the need for a ViewScreen with the act of viewing their new animation, and select this menu item. Immediately, a new ViewScreen appears in their environment,

and this should successfully indicate to the user that they are on the correct course of action.

The next step is to load the visualisation information, so that it can be viewed. An experienced computer user would realise that since they were earlier given the choice of naming the data file, they may well have to give this name to the ViewScreen, so that it knows where to look. There are no buttons on the ViewScreen that can be associated with loading information. However there is a *File* menu, and it is common for loading functions to be available via *File* menus. When the user looks in the *File* menu, they should immediately encounter the *Open File* option. This is again a standard name throughout many applications. Selecting *Open File* will cause exactly the same kind of file browser to appear as was presented the user earlier on. At this point, it should be intuitive to experienced computer users that they can use this browser to select the file to open.

Once the file has been opened, the browser disappears, indicating the completion of that particular sub-task. However, the file has not yet been scanned. It would seem intuitive at this point for the user to try and view their loaded animation. If they try and use the various *play*, *rewind* and *fast forward* buttons however, working off a tape deck analogy they should be familiar with, nothing will happen. This lack of activity should indicate that the correct course of action has not yet been achieved. The correct course of action requires that the file first be scanned. This is because the file may be changed while being used, perhaps because of updates being sent to that file by the test driver. The separation of *open* and *scan* allows the file to be re-read *without* having to re-specify it via the file browser. However, this may not be immediately obvious to a user.

Once the user has seen that the animations are still not available, their only recourse is to review the contents of the *File* menu. This review will discover that there is a second menu option, *Scan File*. The user should be able to associate *scan* with *read*, and see this option as the correct option to take. Once *Scan File* has been selected, the animations will be read in, and the sequence name entered earlier will appear in the combo box. This is an indication to the user that the animation is now available. They should then be able to associate use of the *play* and other buttons as the correct course of action, as this is a common analogy. The ViewScreen now shows the animation, and this should indicate to the user that the correct course of action has been taken.

8.2.3 Summary

The cognitive walkthrough that we've performed makes several assumptions about a reusers actions. We believe these assumptions to be reasonable, given the context of who they are. However, there are certainly other possible paths of interaction that the reuser could follow. We have demonstrated one path so as to learn more about a user might use the tool, and the successes and failures of the interface to be easy to learn.

The cognitive walkthrough seems to suggest that there is room for improvement in the mapping process, and to a lesser degree the actual act of viewing the animations. However, on the whole, our user was able to follow the correct course of actions and was able to achieve their goal of visualising a test drive of the *MyNetwork* class.

8.3 Comparison to Other Tools

There has been substantial research in the field of program visualisation, resulting in the creation of numerous visualisation tools. These tools promote understanding of the thing being visualised, although not necessarily with the explicit goal of promoting code reuse. These tools exhibit a wide range of characteristics, and we now compare some of these tools to Dyno. Chapter 4 discussed the characteristics of program visualisation tools, and we will continue to use its terminology.

The two most notable tools that Dyno shares a common purpose with are *Dyre* and *Tarraingím*. Both of these tools were created during the early to mid 1990s: John Miller-Williams' Dyre [Mil97] is a trace view animator for C++ applications.; James Noble's Tarraingím [Nob96] is a general purpose visualisation system for Self programs.

8.3.1 Dyre

Dyre is a tool that enables the visualisation of C++ applications, with the emphasis on the reusability features of the source code. A user loads C++ applications into Dyre, and can then view and manipulate trace views of the execution. A user manipulates the trace view by interacting with a tape-deck style interface to play, pause, fast-forward and stop the animations. The types of animations Dyre supports is restricted to trace views, and does not allow other kinds of animations, such as visualisations of tree structures or stacks. Dyre does not require the C++ source code to be annotated with calls to the visualisation routines. Rather it uses the commonly available C++ debugger *gdb* to gather information at

runtime. This information is then used to create the animations. Dyno shares several things in common with Dyre. Both tools use the PMV model in the design architecture, and both tools use debugger technology, rather than source code annotation, as the primary means of gathering information. There are also differences. Dyre was developed to support *reusability* (see section 2.2). By being shown the dependencies and features in reusable code, the user can gain a better understanding for what makes code more reusable, that in turn promotes the creation of more reusable code later on. Dyno supports the act of reuse, although since the visualisations created in Dyre are at least partially reproducible in Dyno, Dyno could also support reusability. Another difference is that Dyre works at the application level, requiring that entire applications be loaded in. Dyno, on the other hand, works on the class level and does not require the class be a totally self-sufficient program in its own right.

8.3.2 Tarraingím

Tarraingím enables the visualisation of programs written in the Self object oriented language. Tarraingím supports a wide variety of visualisations, across several levels of abstraction, such as trace views and visualisations of data structures. Tarraingím offers at least the same range of visualisations that are capable with Dyno, and Dyno uses the same PMV architectural model as Tarraingím. Also, Tarraingím does not require the user to manually annotate the source code. Instead it uses monitors to detect events in the program and then pass these on to a *strategy subsystem*, that are similar in principle to the event filters and data models used in Dyno. The strategy subsystem translates the events into meaningful information for the display subsystem.

Dyno, Tarraingím and Dyre are all limited to visualising source code of a single language, this being Java, Self and C++ respectively. The graphical vocabulary available to animators of Dyno views is primarily dependent on the capabilities of the Java AWT, a library external to the tool itself. In comparison, Tarraingím offers its own graphical vocabulary to its animators.

8.3.3 Algorithm Animation Tools

Numerous other algorithm animation tools have been developed during the past few decades. We now mention a few of these tools as a broadly representative sample of the field, although our sample is by no means an exhaustive one.

Balsa, and its successor *Zeus* [Bro92], were developed at Brown University. Written in C, they support the animations of algorithms, typically implemented in Pascal. Balsa supports multiple views of an algorithm or data structure, as does Dyno. Another tool developed at Brown University was Tango [Sta90]. Tango first came onto the scene in the late 1980s, created by John Stasko et al. Tango created animations of annotated algorithms, using a three step procedure consisting of defining abstract operations and events, designing animations to simulate abstractions and operations, and then mapping the algorithm to the animation scenes. Following on from the work done with Tango, two more systems were developed during the early 1990s, firstly XTango, and then Polka. These systems were aimed at being easy to use. Users would not need to know a lot about graphics to be able to create animations of algorithms. The later tools were general purpose algorithm animation tools that allowed users to employ colour, sound and smooth animations in two, or two and a half, dimensions to document their algorithms in an easy fashion. Polka also supported the visualisation of parallel programs. Another visualisation tool developed for parallel programs is the Pavane algorithm animation tool [RCWP], developed by Roman and Cox at Washington University in St Louis. The visualisations in Pavane are specified in a declarative manner.

Balsa, Tango, and their derivatives, are tools that require the algorithms be annotated by the user to output a trace file, that can then be used in conjunction with animation constructs, resulting in the algorithm animations. Dyno differs in this respect, as it retrieves information by using debugging tools, independent of the original source code. These tools also share something in common with Dyre, in that they are a family of tools that work only on entire applications, rather than at the individual class level. By requiring that an entire application be loaded in, the range of reusable assemblies that could be studied and understood is restricted, as many assemblies are only parts of an independent executable. One more difference is that Polka and Pavane support the visualisation of parallel programs, unlike Dyno.

Lastly, Anim [BK91] is an example of an algorithm animation tool capable of working with applications written in different languages. Anim is a post-mortem animation tool. It requires that an application be annotated with simple printout statements, so that upon execution, the application will create a plain text script of what happened. This plain text script is then run through a visualiser to create animations, or stills for inclusion in documentation. As the script used by the visualiser is produced by annotated print statements in the source code, and since most languages support printing, Anim is language independent. This differs

from Dyno, as Dyno can only visualise Java code. However, the requirement to annotate the source code is problematic for a tool supporting reuse of assemblies, as discussed elsewhere.

8.3.4 Visual Debuggers

Another type of tool to make use of visualisation is the visual debugger. Visual debuggers support the testing stage of the software lifecycle, and we now give one example of this genre.

Lens is a visual debugging tool created at Georgia Tech and both uses the XTango system for creating animations, and uses Unix's dbx debugger. At its inception, Lens was designed to run on Sun Sparc stations running X11, and would enable the user to specify certain animations that should occur when the program reached a breakpoint in the debugger. As well as this, Lens allowed users to visualise certain data structures. Lens and Dyno do not share very much in common with respect to the method of data collection, or user interaction. Dyno is primarily aimed at test driving assemblies, whereas Lens supports testing. This difference is fundamental, as the former looks at understanding how functional code works, and the latter looks to check if code is functional. However, we mention it as it is a visualisation tool that uses debugger technology.

8.3.5 Blue, BlueJ and Visual Programming Environments

Blue is an integrated environment for teaching object oriented languages to undergraduates [RK97]. This environment is an example of a visual programming environment, using graphical constructs for creating programs written in the Blue language. There is a strong emphasis on visualisation and interaction techniques in the development of this environment. In particular, Blue allows methods to be called interactively, without the use of normal input/output operations. Following on from the original Blue environment, the tool was reimplemented to use and support the Java programming language, and renamed BlueJ [Blu]. BlueJ continues the same basic principles of Blue, using visualisation to support the creation of programs, rather than the understanding of programs that Dyno supports. BlueJ also uses a debugger, as does Dyno, although not for the same reasons. BlueJ uses a debugger to support testing, where as Dyno uses a debugger to determine what should be visualised in the test drives.

8.4 Tool Technologies

We designed and implemented Dyno not only as a proof of concept, but to also have a chance to identify the key technologies needed, and to be able to experiment with these technologies.

As a point of reference for future tools implemented in other languages, Dyno was written in 10,000 lines of Java code, spread across 130 classes. There was an additional 670 lines of C++ code written for use with the JVMDI.

Having discussed in section 8.1 how Dyno measures up against our previously determined criteria, we now look at some of the technologies that were necessary. We use our experiences of these technologies in the JDK, to provide concrete examples throughout this section.

8.4.1 Using Debuggers To Gather Information

A tool to visualise the runtime behaviour of an assembly must be able to access, at runtime, information regarding that assembly. In section 5.3, we discuss various ways that this information can be retrieved. One commonly used approach is annotation. Annotation requires that a reuser be able to modify the source code of an assembly to include calls to the visualisation routines. Since a reuser may not have easy access to the source code (for example: reusing a pre-compiled library), or does not have a complete understanding of the assembly with which to know where the annotations should go, this approach is problematic for tools supporting code reuse.

Instead, Dyno makes use of the JVMDI to gather information. JVMDI allows us to detect events that occur at runtime and also gather information relating to these events. Using debuggers is advantageous because they are designed to be able to extract information from a system with a minimum of intrusion in the source code [BMMWT99]. Debuggers are frequently used during implementation and, to facilitate the finding and fixing of faults, often have a wide range of features such as the ability to temporarily halt execution at certain points, execute single source statements, determine execution traces, interrogate state and so on. Many of these features can be used to detect certain events (such as the arrival at a particular point in the code, like a breakpoint), and then retrieve information regarding that event and also the state of the data at that point. Reusing debuggers in this method allows us to avoid having to modify the source code, compiler or runtime environment to retrieve the required information.

8.4.2 Support for Circumventing Encapsulation

Dyno allows a reuser to invoke methods and inspect fields that would ordinarily be hidden from the programmer. Numerous OO languages (for example: Java, C++) support encapsulation as a method of promoting a better coding style. This encapsulation hides details

that may be of use in understanding how code works and what the code does. Dyno uses the Reflection API as library support for circumventing the usual access permissions and has therefore broken the encapsulation of the assemblies. Such support for circumventing access controls has led to the reuser having more control over what they execute and visualise in the assemblies. The latter point is perhaps more relevant than the former, as the visualisations may well depend on showing, or at least having access to, information about the private state of an object (such as the tree view example from chapter 7). Since the reuser may need to map these private data fields of an object to attributes in a view, the tool needs support for accessing those private data fields, and for displaying them to the reuser so they know what can be mapped. Without this information, the reuser may be limited to only visualising publicly available information, and this may present a barrier that further increases the difficulty for the reuser in the mapping process.

8.5 Understanding Assemblies

The motivating idea of this research is that the cost of reuse can be reduced by reducing the cost of understanding. We have set out to demonstrate that reducing the cost of understanding can be achieved by supplying reusers with tools that support their understanding of assemblies. The tool that we have developed supports understanding by allowing a reuser to gain first hand experience in using an assembly, as well as by creating visualisations of the assembly in action. We now look at test driving and visualisation in more depth.

8.5.1 Test Driving

We claim that test driving promotes understanding of assemblies. A tool supporting test driving should present the reuser with the chance to explore the interface of an assembly, and could optionally include features to allow exploration of its implementation. Documentation could then be created by visualising the results of the user's test driving. We now discuss what the reuser can learn by using tool support that includes test driving functionality.

Test Driving Assembly Contexts

Rosson and Carroll have researched how experienced Smalltalk programmers reuse code [RC96]. Their empirical research suggests that experienced programmers learn how to reuse a piece of code by looking at its past uses. Such programmers incorporate these uses into the

current problem, customising and modifying as necessary, so as to gain a better understanding for how the code is used, and what its side-effects may be.

The programmers' use of context code could be further supported by a test driven visualisation tool. The test driving would allow a reuser to explore how a context uses an assembly, and the visualisations could help to describe what happens to the assembly as various events occur. Rosson and Carroll have developed a tool called the Reuse View Matcher (RVM). The RVM shows examples of an assembly being used in different scenarios. The examples use animations to show assemblies in action, so that the programmer can form an understanding of what that assembly does. The programmer watches these examples and then can use the example code as a guide to how they should reuse an assembly. Given that there are typically multiple scenarios in which an assembly is reused, the reuser can have a choice in what they see. This can be extended to include our idea of test driving. Permitting a reuser to test drive the example code in more detail, using the test drive as the structure of the animation, gives the reuser finer control over what they end up seeing.

8.5.2 Understanding via Visualisation

A reuser can also better understand an assembly by using documentation. Most forms of documentation are text, or some other form of static media, and can suffer from a lack of expressive power in describing dynamic runtime behaviour. Visualisation research has looked at ways of better expressing information, including dynamic information.

Animating Algorithms and Data Structures

Animations have often been used to reinforce the learning and understanding of data structures. Films and videos of data structures in actions, such as *Sorting Out Sorting* [Bae81], have been created, while some visualisation tools concentrate solely on data structures. We also have developed tools, implemented in the scripting language Tcl/Tk, that demonstrate B-Trees, AVL-Trees, Skip-Lists and Hash-Tables working [BMW95].

These were then used during a second year undergraduate paper in software engineering, to demonstrate how these data structures performed. The user could drive the animations, determining what was added to the structure. The implementation of the data structures were built directly into the animations and were therefore difficult to reuse in projects that actually required implementations of these data structures. However, students did continually refer to these animations when implementing the structures in C++ for their own assignments.

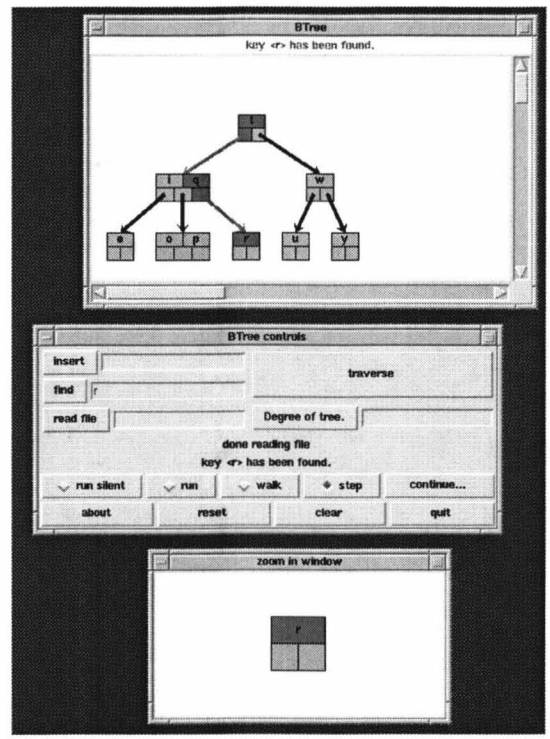

Figure 8.1: A B-Tree data structure, in which the letter 'r' has been found. The user can enter keys to store in the tree, and also lookup keys. The tool then shows animations describing how the keys are entered or found.

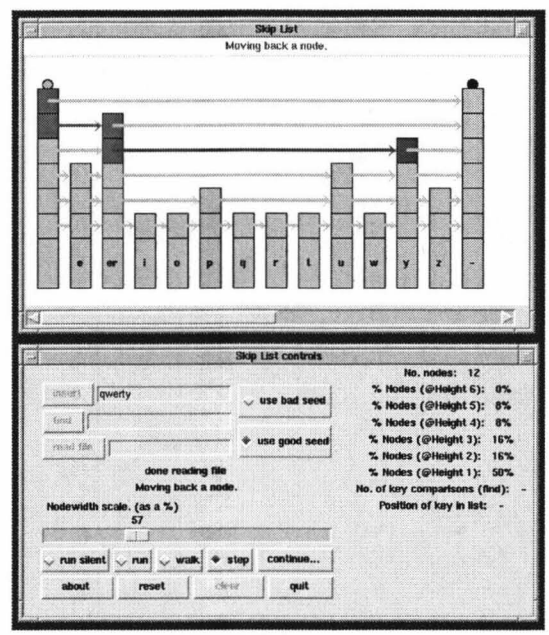

Figure 8.2: A skip-list data structure showing the various heights of each node and their connections. As with the B-Tree data structure, the tool has specific animation routines for showing to a user how keys are added into, and found in, a skip-list.

VICTORIA UNIVERSITY OF WELLINGTON

Figures 8.1 and 8.2 show B-Tree and Skip-List data structures respectively.

Each type of animation is a separate program, and the user exercises control over what is added, or found, in the data structures. The programs then animate accordingly. The user has no input into how the animations handle certain events.

Documenting Assemblies via Documentation

In chapter 4, we proposed that the animations created from our visualisation process could be used as a form of documentation. The information contained within this documentation would be initiated and controlled by the results of the reusers test driving. This would make the documentation action-oriented and centered on tasks that the reuser has deemed relevant to their understanding of the assembly. Research by van der Meij and Carroll into minimalist documentation [vdMC98] has identified choosing an action-oriented approach, along with selecting real tasks, as key principles. Much of their research in this particular area centers on documentation for entire tools, aimed at the end-user. However the principles can be carried across into programmer-oriented documentation at the assembly level. Van der Meij and Carroll see the creation of documentation that uses real tasks as being well supported by the user's original motivation for being interested in the tool. Likewise, a reuser's interest in an assembly can be used as a guide for what the documentation should describe. The tasks that a reuser performs while test driving an assembly indicate what the reuser is interested in. Documentation that describes these tasks would therefore be more likely to be relevant to a reuser, and this relevance should lead to a greater willingness to use the documentation and gain an understanding of the assembly.

A potential problem is that the reuser may not have a sufficient initial understanding of the assembly to know what to do, or what to visualise. Unlike the documentation often created by our tools, the documentation created for the minimal manual in van der Meij and Carroll's research is not written by the user, but for the user by members of the tool's development team. This means that the documentation is written by someone proficient in the tool's execution and behaviour, rather than by someone who is trying to understand how the tool works. If the reuser does not know how to perform tasks properly, or is not aware of certain functionality, then the resulting documentation may be equally confused and non-representative of the true abilities of the assembly.

The rebuttal to these criticisms is that the animations seek to reinforce what the reuser already knows (for example: "if I call *push* then the element I specify appears at the end of

the array"), or offer a new perspective on what the reuser thinks the functionality is. While the clarity of the animations, with regards to explaining the functionality, are prone to errors or misjudgments the reuser makes during test driving, the animations may still be able to help the reuser know *when* they have done something right. Also, Dyno allows the reuser to save an animation to secondary storage for later reference.

With this in mind, the programmers of the assembly, who should know the functionality of their own code, could create the animations themselves. These animations are still derived from test driving, and can be based on tasks that the programmers believe would be relevant to later reusers. This should ensure that the documentation remains action-oriented, and focussed on tasks the reuser would deem relevant and useful. The latter point, regarding relevancy and usefulness, is based on the assumption that the programmers understand their own assembly sufficiently well to be able to predict how reusers may generally go about reusing it. If it is not the case that the programmer knows how the reuser is going to reuse the assembly, then the case for programmer created visualisations being focussed on the reuser's needs is weakened. However, if this is the case, then the programmers should know the sequence of events that would properly demonstrate its functionality and proper operation to a reuser. This sequence of events should mirror the sequence of events the reuser would undertake to use the assembly.

Another potential problem is that the documentation is also reliant on the reuser knowing the view well enough to be able to successfully map from the assembly to the animations. Any errors in the mapping could lead to confusing visualisations and incorrect understanding. The tool should offer some help here by supplying the reuser with an interface that supports the act of mapping. As well as this, the view's components (its methods and attributes) should be given clear and appropriate names to help the reuser. But as well as this, the animators in charge of creating the views might be able to devise a pre-determined mapping that the reuser can plug their assembly into, or customise. We now discuss the role of the animator, and the potential of this approach to supporting mapping.

The Role of Animators in the Visualisation Process

Animators are responsible for the creation of views. Views create animations from information, supplied by the runtime environment, that the reuser identifies as being relevant. The quality of the views' implementations are important, as visualisations that give incorrect interpretations of information are no less harmful than misleading documentation. Visual-

isations can also overburden the reuser with information, leading to information overload. With this in mind, the animators' role in creating good views is crucial to the overall success of using visualisations in tool support for understanding.

Animators may create views with no particular assembly in mind, and then provide them to reusers for whom they are relevant. They might conceivably be able to devise pre-defined mappings for their views so as to better help reusers in knowing what in an assembly relates to the features of a visualisation. These mappings could then be customised for individual assemblies, or simply have the assemblies hooked into them. Given that an assembly programmer has a high degree of choice in the naming scheme and structure of an assembly, these mappings will either have to be very general to fit many assemblies, or be specific and only suit a few assemblies. The potential of this approach also depends on how much mapping is required by the tool in the first place, as the less mapping is required the less call there would be for pre-determined mappings, which may themselves need non-trivial customisation.

Creating sophisticated visualisations is a non-trivial task, especially for assemblies that require large amounts of information to be visualised in a concise and unambiguous manner. The art of creating these visualisations requires some understanding of how people best process visual information, and expert animators are crucial to this process. However, even simplified visualisations, such as the UML-style sequence diagram used in chapter 7, can be of use, giving the reuser some idea of what is going on, with relatively little complexity. While these simpler views may not describe everything that is going on, they can still show the reuser a new perspective on an assembly. This, possibly combined with other sources of documentation, can then further the reuser's understanding of the assembly. Combinations of simple views can also work to cover most of the fundamental information, trading the lost details for visualisations that are easier to comprehend.

8.5.3 Supporting Code Reuse

Understanding an assembly is an important activity in the code reuse process. By understanding an assembly, the reuser is in a good position to judge the assembly's applicability to a new context, and how that assembly would fit into that new context. If there are any modifications required for the assembly to fit, then a thorough understanding will reduce the chances of introducing faults. Dyno supports the reuser by allowing them to test drive an assembly, and create visualisations from the test driving. By test driving, a reuser can see

first hand what the methods of an assembly do, both in terms of the return value and also in any modifications to the state that may occur. The visualisations created from the test driving act as a form of dynamic documentation. The visualisations can create different views of the runtime behaviour, and give the reuser an appreciation for what the assembly is doing, and how it is doing it. By knowing what the assembly is doing, the reuser can determine if it fulfills the task required in the current situation. By being able to see how it does what it does, the reuser can then judge whether the assembly is appropriate for reuse with respect to how it interacts, and what is required for its execution. Although Dyno has not been thoroughly user-tested, it would seem likely that first hand experience of an assembly would be beneficial, as would the use of visualisations to take advantage of the human ability to rapidly process these. Both approaches work in other fields, such as the use of pictures in text books, and audio files in multimedia such as electronic encyclopedias and other reference materials. We have sought to apply these successful techniques in the support of code reuse.

Chapter 9

Conclusions

Having discussed the role understanding plays in code reuse, and the potentials for tool support based on test driving and program visualisation, we now conclude this thesis.

9.1 Overview

Code reuse is a possible solution to meeting high demand for software. However code reuse can be costly to perform, negating much of the benefit. We believe one reason why the software engineering community has not fully embraced code reuse is its high cost. One method of promoting code reuse is to bring the cost down, thereby increasing its appeal. We identified the cost of understanding as a core component of the cost of code reuse. This thesis has looked at a way of reducing the cost of understanding, by using an approach that has been used to positive effect in other areas of software engineering. This approach involves creating tool support to reduce the time and effort required to understand an assembly. Specifically, we have looked at the roles that test driving and visualisation can play both in the process of understanding, and in tool support for promoting understanding.

We began by introducing the general principles of code reuse. Having established a basic terminology, we then discussed the need for understanding in the code reuse process. Various approaches that support understanding were discussed, such as gaining experience with an assembly via test driving, and the creation of dynamic documentation to mirror the dynamic nature of execution. Following this was a discussion on some of the principles and tools to be found in program visualisation, a field that has a specific goal of promoting understanding, although not with reuse explicitly in mind. This laid the groundwork for the identification of requirements that tool support for understanding would need to exhibit. Next, the design,

implementation and example behaviour of Dyno were introduced. Dyno is a tool developed during this research both as as an exploration of the requirements and issues relating to such tools, and also a proof of concept. Dyno supports the reuser in test driving Java classes and visualising their execution. The test driving functionality allows a reuser to invoke individual methods, and to also inspect the state of objects. The reuser does need to supply some information to Dyno for the creation of the visualisations, in terms of mapping the state and events of a class onto the abstract attributes and operations in the visualisations. However, there are minimal required changes to the class source code. We then evaluated Dyno with respect to our previously discussed criteria, and performed several cognitive walkthroughs to better understand its learnability. We then discussed the limitations of the approach we have taken.

9.2 What Has Been Achieved?

The research in this thesis has identified test driving and visualisation as two techniques for understanding assemblies with the purpose of reuse, and identified the required characteristics of tools to support these techniques. Permitting a reuser to test drive an assembly gives that reuser important experience of the assembly's functionality. Program visualisations can further the understanding of assemblies by making use of the human capability to process large amounts of visual information. By tying the visualisations to the events that occur during the test driving, the resultant documentation is centred on real tasks that the reuser is interested in. Dyno was designed and implemented to test these ideas.

Dyno allows the test driving and visualisation of Java classes. The visualisations for a class are based on a test drive of the class. Dyno does not require complete applications. This is important as requiring that only entire applications can be loaded could overload the reuser with information, or hide the salient reusable code. By allowing the reuser a finer level of granularity in their assemblies, in this case at the Java class level, Dyno allows a reuser to have direct control and manipulation of individual methods and state, giving them a better appreciation for the structure and responsibility of different parts of the code. The resulting visualised documentation can then be stored and replayed, allowing a reuser to interpret what an assembly does, and how. The design and implemention of Dyno has allowed us to gain a better appreciation for the techniques required, and the characteristics that such tools should exhibit.

Dyno has been briefly compared to other program visualisation tools. This was done to

highlight the differences and similarities in the characteristics of tools designed for general understanding (as most PV tools have been), and tools specifically designed with understanding for code reuse (as in the case of Dyno). For example, many program visualisation tools require annotation of an algorithm's source code. However, annotating assemblies can lead to problems. Firstly, the reuser may not be able to modify the source code, and secondly the annotation may be faulty, reducing the worth of the resultant visualisations. Dyno's solution to this, following on from previous attempts by the Dyre [Mil97] and suggestions relating to Tarraingím [Nob96] tools, is to use debugger technology. This thesis has offered another example of the use of debugger technology to interrogate the runtime environment, and act as an information source for visualisations.

9.3 Future Work

There are two areas of possible future work resulting from this thesis. The first area involves combining test driving and program visualisation tools with other forms of code reuse support. The tools mentioned in section 2.6 include such things as code repositories, search engines, and smart editors. Test driving and visualisation tools could be combined with these tools to create a single integrated reuse environment. This environment would then seamlessly support all parts of the code reuse process, from the storage of assemblies, to their retrieval, understanding, and subsequent inclusion in a new context. By adding in other CASE tools that support other facets of the software engineering process, the new environment could make reuse easier and more convenient. The time and effort taken to move from the main programming environment, to one that allowed a programmer to reuse, would be removed and therefore a programmer may be more inclined to consider reuse as a possible solution. This in turn might help to cement reuse's role in the software lifecycle, and by strengthening its appearance of being standard practice, programmers reservations might be at least partially alleviated.

The second area of future work centres on possible improvements to the Dyno tool. Firstly, the success of such a tool is at least partially determined by how easy it is to use. The hardest part of using Dyno is supplying mapping information to link the assemblies to the visualisations. This mapping process requires the reuser to expend time and effort on a tool that is supposed to be saving them time and effort. With this in mind, future work could be done on increasing the amount of automation in the mapping process, or on improved interfaces to further help the reuser. It seems unlikely that the process could be totally automated, due

to the latitude an assembly programmer has in structure and nomenclature. However, some help could be afforded by allowing an animator to supply some default mapping with their views. The reuser could then customise these default mappings as necessary. Similarly, an assembly programmer could supply, and Dyno could support the importing of, pre-defined mappings.

Secondly, Dyno has some limitations, notably in its support for multi-threaded assemblies, or those that make use of parallel processing. There are visualisation tools that support these kinds of assemblies (although again, not with reuse in mind), so Dyno could be enhanced to include this form of support.

Thirdly, support could be given to better measure the performance of an assembly. As performance (both in terms of memory requirements and time requirements) are often important to an assembly's suitability for reuse, it would be advantageous if the reuser was able to properly measure these quantities. With these measurements, a reuser would then be in a better position to ensure that an assembly was not improperly reused for an inappropriate task. Improper reuse of an inappropriate assembly not only causes problems in the project where it is reused, but the experiencing of these problems may also cause a programmer not to reuse in the future.

Fourthly, more research should be undertaken into the usability of Dyno, along with empirical studies on the actual effect on the amount of code that is reused, and the time and effort required. A similar approach could be used to that demonstrated by Rosson and Carroll [RC96], who performed empirical tests on their Reuse View Matcher by using several professional (rather than student) Smalltalk programmers. The Smalltalk programmers were asked to complete some reasonably realistic problems, and were then observed as to how they went about achieving a solution. The time and effort involved in using Dyno, along with its overall effectiveness, could be measured by gathering a pool of experienced Java programmers, and evaluating how they used Dyno, and reuse principles in general, in creating solutions to problems.

Finally, perhaps most importantly, the claim that Dyno helps a reuser to understand an assembly, should be evaluated. This could be achieved by user studies, that empirically test to see how much understanding is benefited by using Dyno.

9.4 Contributions Of This Thesis

This thesis has identified understanding as a key requirement, and a potentially significant cost, of the code reuse process. It has proposed that tools can, and should, be created for supporting the process of understanding, so as to make understanding easier for a reuser, and less expensive in terms of time and effort. The thesis has identified test driving of assemblies, and visualisation of same, as two approaches to understanding, and has looked at how these two approaches can be combined in tool support. It was then determined that the combination of approaches should result in the visualisations being created using the events and information created during the test driving process. Research has been conducted into the technologies required for tool support of this nature, and especially focussed on the use of debugger technology to replace the common annotation techniques used by other visualisation tools.

The design, implementation and evaluation of our tool acts as a proof of concept that test driving and visualisation may form the basis of tools for supporting understanding and code reuse.

Bibliography

[Bae81] R. M. Baecker. Sorting out sorting. Video, 1981. Narrated colour videotape, ACM SIGGRAPH '81.

[BK91] Jon L. Bentley and Brian W. Kernighan. A system for algorithm animation. *Computing Systems*, 4(1):5–30, Winter 1991.

[Blu] Bluej: The interactive Java environment homepage. project webpage, URL: "http://www.sd.monash.edu.au/bluej/".

[BMMWT99] Robert Biddle, Stuart Marshall, John Miller-Williams, and Ewan Tempero. Reuse of debuggers for visualization of reuse. In *Proceedings of the Symposium on Software Reusability*, May 1999.

[BMW95] Robert Biddle, Stuart Marshall, and Ben Wong. Tcl/tk algorithm animation tools. Source Code, 1995. These are applications written in the Tcl/Tk for animating a variety of algorithms.

[BP89] Ted Biggerstaff and Alan Perlis. *Software Reusability: Concepts and Models*, volume 1, chapter 1. ACM Press, 1989.

[Bro92] Marc H. Brown. Zeus: A system for algorithm animation and multi-view editing. Technical Report 75, Digital Equipment Corporation, Systems Research Centre, 28 February 1992.

[BT96] Robert Biddle and Ewan Tempero. Understanding the impact of language features on reusability. In *Proceedings of the Fourth International Conference on Software Reuse*, April 1996.

[Glo94] Al Globus. Principles of information display for visualization practitioners. Technical report, NAS, November 1994.

[Jav] Javadoc - Java API Documentation Generator. The Javadoc Homepage http://java.sun.com/products/jdk/javadoc/index.html.

[JS94a] Dean Jerding and John Stasko. Using visualization to foster object-oriented program understanding. Technical Report GIT-GVU-94-33, Graphics, Visualization and Usability Center, College of Computing, Georgia Institute of Technology, July 1994.

[JS94b] Dean F. Jerding and John T. Stasko. Using visualization to foster object-oriented program understanding. Technical Report GIT-GVU-94-33, Graphics, Visualization, and Usability Center Georgia Institute of Technology, Atlanta, GA, July 1994.

[LY97] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison Wesley, 1997.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice-Hall, 1997.

[Mil97] John MillerWilliams. A program visualisation tool for emphasising the dynamic nature of reusability. Master's thesis, School of Mathematical and Computing Sciences, Victoria University Of Wellington, 1997.

[MMM95] Hafedh Mili, Fatma Mili, and Ali Mili. Reusing software: Issues and research directions. *IEEE Transactions on Software Engineering*, 21(6), June 1995.

[Mye86] B. A. Myers. Visual programming, programming by example, and program visualisation: A taxonomy. In *Proceedings of Human Factors in Computing Systems (CHI '86)*, pages 59–66. ACM Press, 1986.

[Nob96] James Noble. *Abstract Program Visualisation.* PhD thesis, Department of Computer Science, Victoria University of Wellington, New Zealand, 1996.

[OW96] Michael Oudshoorn and Hendra Widjaja. Devising a program visualisation tool for concurrent and object-oriented programs: A survey. Technical report, Univeristy of Adelaide, 1996.

[OWE96] Michael Oudshoorn, Hendra Widjaja, and Sharon Ellershaw. *Program Visualisation: A Snapshot*, volume 7 of *Software Engineering and Knowledge Engineering*, chapter 1, pages 3–26. World Scientific Press, 1996.

[PBS94] B Price, R Baecker, and I Small. A principled taxonomy of software visualization. *Journal of Visual Languages and Computing*, pages 211–266, 1994.

[Pre92] Roger Pressman. *Software Engineering: A Practitioner's Approach.* McGraw-Hill International Editions, 3rd edition, 1992.

[RC93] Gruia-Catalin Roman and Kenneth Cox. A taxonomy of program visualisation systems. *IEEE Computer*, pages 11–24, December 1993.

[RC96] Mary Beth Rosson and John M. Carroll. The reuse of uses in smalltalk programming. *ACM Transactions on Computer-Human Interaction*, 3(3):219–253, September 1996.

[RCWP] G.-C. Roman, K. Cox, C. Wilcox, and J. Plun. Pavane: A system for declarative visualization of concurrent computations. Technical Report 91-26, Washington University.

[RK97] John Rosenberg and Michael Kölling. I/O considered harmful (at least for the first few weeks). In *The Proceedings of the Second Australasian Conference on Computer Science Education*, pages 216–223, July 1997.

[Sta90] John T. Stasko. Tango: A framework and system for algorithm animation. *Computer*, 23(9):27–39, September 1990.

[Sun] Swing connection, sun microsystems. http://java.sun.com/products/jfc/tsc/.

[Tra95] Will Tracz. *Confessions of a Used Program Salesman: Institutionalizing Software Reuse.* Addison-Wesley Publishing Company, 1995.

[vdMC98] Hans van der Meij and John M. Carroll. *Minimalism Beyond the Nurnberg Funnel*, chapter 2. MIT Press, 1998.

[WRLP94] Cathleen Wharton, John Rieman, Clayton Lewis, and Peter Polson. *Usability Inspection Methods*, chapter 5, pages 105–140. John Wiley & Sons Inc., 1994.