Abstract Program Visualisation

Object Orientation in the Tarraingím Program Exploratorium

by

Robert James Noble

A thesis

submitted to the Victoria University of Wellington
in fulfilment of the
requirements for the degree of
Doctor of Philosophy

Victoria University of Wellington
1996

Abstract

Program visualisation uses computer graphics and animation techniques to produce pictures illustrating the dynamic behaviour of a running computer program. Most program visualisation systems display either language-level details of programs or high-level overviews of the program's algorithm. This thesis investigates the use of abstraction in program visualisation. The goal of the project was to find techniques which could produce displays of programs at all levels of abstraction, and which would not require a large amount of information about the implementation of the program to be visualised. Based upon analyses of program visualisation and abstraction in programming, a model of abstract program visualisation is developed. This model uses object orientation to explicitly represent abstractions in the programs to be visualised. An object oriented framework for the design of an interactive program visualisation system (a program exploratorium) is developed based upon this model. This framework is used to construct Tarraingím, a prototype program exploratorium, as a proof of concept for the model. Tarraingím produces multiple views of object oriented programs at multiple levels of abstraction, by invisibly monitoring the programs' execution and using the information gathered to control graphical animations. Tarraingím is written in the SELF programming language, and visualises programs written in that language. A series of examples is presented to illustrate Tarraingím in action.

Acknowledgements

I would like to thank my two crusading supervisors, Lindsay Groves and Robert Biddle, for keeping me moving on a long and slow road, and it is greatly to their credit that I have reached the end. Their refusal to take *OO* for an answer has markedly improved this thesis, and they were always graceful when I ignored their advice.

Thanks also to John Hosking and Phil Cox, the external examiners for this thesis.

Fellow travellers at VUW have kept me company along the way. Ray Nickson deserves special mention, for sharing an office and using the X terminal in the morning so I could use it in the afternoon. David Andreae, Andy Bond, and more recently Aaron Roydhouse also showed by their commitment that the whole business was worthwhile. VUW staff, especially Peter Andreae, Brian Boutel, and Ewan Temporo provided encouragement and support as I took the first hesitant steps into academic life. I am indebted to the students I had the privilege of teaching: I am sure I learned much more from you than you from me.

For the last eight years, I've used what must be the best Unix installation in the Southern Hemisphere. Thanks for this, and for the last-minute rescues of machines and files above the call of duty are due especially to Mark Davies, Duncan McEwan, Jules Anderson, and Andy Linton, who also provided a name for the system.

This project would not have been possible without the support of the SELF group (lead by David Ungar and Randall Smith) not only for SELF, but also for answering all my questions, and fixing all my bugs. Thanks to Craig Chambers, who put unwind protection into the compiler (some other program *must* use this by now), and Urs Hölzle, Ole Agesen, and Bay-Wei Chang for their comments and support.

Dave Morley and Michael Richardson kindly used various versions of NAVEL in their own projects, and provided much useful feedback. Thanks to Geoffrey Pascoe, for interesting information about the genesis of encapsulators. John Grundy, Mark Utting, and Antero Taivalsaari offered helpful comments and interesting conversation.

The community of faith at All Saint's Hataitai, my friends, family, and the sometime members of the *Heybabies* helped remind me that there were more important things than the thesis, and that there would, one day, be life after varsity.

Finally, my deepest gratitude to Katherine for her love, support, tolerance, and proofreading. I only hope the experience of the last eight years hasn't put you off the idea of study forever: I'm sure you'll manage rather better than I.

> *Perhaps universities have always worried about what would happen to their image*
> *if it turned out that a Ph.D. degree had been awarded to an illiterate.*
> *Hence, the thesis requirement.*
>
> How to write and publish a scientific paper, *Robert A. Day* [59]

iv

Contents

List of Figures

List of Tables

1

Introduction

An algorithm animation environment is an "exploratorium" for investigating the dynamic behavior of programs. ... It presents multiple graphical displays of an algorithm in action, exposing properties of the program that might otherwise be difficult to understand or might even remain unnoticed.

Marc Brown, Introduction to *Algorithm Animation* [33]

1.1 Abstraction and Program Visualisation

Program Visualisation is the application of computer graphics techniques to computer programs, in the same way that scientific or engineering visualisation applies these techniques to scientific data or engineering artifacts. Visualisations can assist programmers in constructing, debugging and maintaining programs. They can also be used in teaching general principles of computer science, including the design of data structures and algorithms.

The existing work in program visualisation can be grouped into two broad categories, according to the kinds of views presented. *Algorithm Animation* systems display high-level pictures of the operation of an algorithm, while *Graphical Debuggers* display language-level views of a program.

Algorithm animation systems illustrate an algorithm's intent, and may bear no relation to that algorithm's implementation. Such views must be specially designed because the intent of an algorithm is somewhat intangible and cannot in general be determined automatically. Typically, the algorithm to be visualised is implemented using facilities provided by the visualisation system, so that as it runs the animated displays are updated.

Graphical debuggers (and graphical programming environments) display information about a program's implementation. A graphical debugger illustrates run-time control flow and data structure, while a programming environment presents static views which may be edited to create programs. For example, a debugger may display a data structure constructed of records and pointers using a "box-and-arrows" diagram, and a programming environment could display the intermodule dependencies within a program as a graph. These views are typically updated by request, rather than continuously.

The difference between these categories is one of the level of *abstraction*. Graphical debugging works at the language level, while algorithm animation displays high-level features of the program. Note that abstraction in this sense does not concern simply the amount of detail a visualisation contains. For

example, a graphical programming environment may produce a histogram of a program's execution profile, or a graph summarising its heap memory usage. These displays present information about the execution of a program in a compressed form, and are useful aids to understanding a program. However, they do not provide insight about the *intent* of the programmer and the abstract ideas used in the construction of the target program.

1.2   Goals of this Research

Programs are generally constructed from many interacting abstractions, with high-level abstractions implemented by lower-level abstractions, which are eventually implemented in the primitive facilities of the programming language. Algorithm animation systems typically present views at only one level of abstraction — the highest level in the program, and graphical debuggers display only programming language level views.

Marc Brown's seminal thesis *Algorithm Animation* [33] described the design rationale of the BALSA system. The first program visualisation system to become widely used and well known, BALSA produced multiple illustrations of the execution of an algorithm, and gave its user control of the animation using an interactive environment. BALSA's user could choose an algorithm to watch, and select several views to display different illustrations of the algorithm. This is Brown's *algorithm exploratorium* — a computer system for exploring the behaviour of algorithms.

Our ultimate aim is to build a *program* exploratorium — an interactive system for investigating programs in all their multifaceted complexity. BALSA, although flexible, performed algorithm animation, and the views it presented were limited to those describing high-level aspects of algorithms. A program exploratorium should take a broad-spectrum approach, capable of dynamically visualising a program at multiple levels of abstraction. For example, a program exploratorium could show how a program implements a given algorithm by presenting views at many levels of abstraction simultaneously.

The practical aim of this thesis is to investigate the design of such a program exploratorium.

1.3   Program Visualisation with Explicit Abstractions

In order to produce an abstract visualisation we must determine the abstractions used within the program to be visualised. We then need to establish a connection between the program and the pictures to be produced, so that pictures can be updated as necessary during program execution. Since the pictures should depict ideas embodied in the program design, this means that we need to associate pictures with the program components corresponding to these abstract ideas.

This approach will work provided that the abstract ideas on which a program is based are represented in the program in an easily identifiable way — that is, they are explicit in the program's structure. Our approach is to use object orientation to organise the program to be visualised. An object oriented program is structured as a collection of self-contained objects, which package data together with the behaviour to process those data. Assuming the program is well designed, each object should represent an individual abstraction in the program, and every important design idea should be represented as an object.

The abstractions can then be visualised by monitoring the objects representing them, and drawing appropriate pictures as the program executes. This monitoring must be efficient, so that the visualisation can proceed at a reasonable rate, and unintrusive, so that it does not affect the operation of the monitored program.

To provide some empirical evidence that this approach is practical, we have designed and implemented a prototype program visualisation system called *Tarraingím* (from the Gælic for *drawing*). Tarraingím produces multiple views of object oriented programs at multiple levels of abstractions, by transparently monitoring the programs' execution and using the information gathered to control graphical animations.

Tarraingím is built as an object oriented framework, that is, as a collection of reusable objects which can be composed to produce an actual working system. It uses *encapsulator* objects to monitor the

program, *view* objects to display visualisations, and *watcher* objects to link views to encapsulators. This gives Tarraingím a very flexible architecture, which can be specialised and extended to produce many kinds of visualisations.

1.4 The Organisation of this Thesis

This thesis is structured as follows:

Chapter 2 starts by describing the PMV (*program, mapping,* and *visualisation*) model of program visualisation, and then reviews the evolution of program visualisation techniques over the last fifteen years. It also surveys related work in program monitoring.

Chapter 3 discusses in detail the various rôles of abstractions in program visualisation. It argues that the existing approaches to handling abstraction are not adequate to support a program exploratorium, and describes a novel approach based upon design abstractions which are explicitly represented in the program to be visualised. This chapter then investigates several alternative paradigms of abstraction in programming, and identifies object orientation as suitable for further investigation. The chapter concludes by introducing the novel APMV *abstract* program visualisation model.

Chapter 4 describes how the APMV model can be employed within a program visualisation system. It addresses two main issues: do objects capture sufficient information about a program's abstractions to allow them to be visualised, and can this information be used efficiently by a visualisation system? This chapter introduces the idea of a monitoring *strategy*, which describes the information about an abstraction that is required to produce a display of that abstraction.

Chapter 5 introduces Tarraingím, a novel program visualisation system designed to test the concepts developed in Chapters 3 and 4. This chapter describes Tarraingím's architectural design as an object oriented framework, and then discusses implementation issues such as the choice of programming language and graphics system. It also introduces the SELF programming language, in which both Tarraingím, and the programs that Tarraingím visualises, are written.

Chapter 6 describes the design of Tarraingím's *display* subsystem. This subsystem consists of *view* objects which are responsible for generating Tarraingím's graphical display and handling user interaction.

Chapter 7 presents the design of Tarraingím's *strategy* subsystem. This subsystem consists of *watcher* objects which connect the display subsystem's views to the objects in the target program. Watchers embody the strategies described in Chapter 4.

Chapter 8 presents the design of Tarraingím's *monitoring* subsystem. This subsystem collects information about the program and supplies it to the strategy subsystem.

Chapter 9 discusses the *encapsulator* objects which actually monitor the target SELF program. It evaluates several alternative encapsulator designs in terms of their effects on the program to be visualised and the kind of information they can supply to the monitoring subsystem.

Chapter 10 presents several examples of Tarraingím in use. It begins by describing Tarraingím from the user's perspective, showing how interface views are used to select objects and views. The chapter then uses several reflexive displays to illustrate the way Tarraingím's components cooperate to produce visualisations. It finishes with a larger example, showing how Tarraingím's views can be used to portray the design abstractions within a parser for a PL/0-like language.

Chapter 11 concludes this thesis with a summary of the contributions that we claim for this work, and an indication of directions to be pursued in future.

2

Related Work

This chapter reviews previous and current work which has influenced our research. We begin by discussing a model of program visualisation that has been proposed by several researchers in the field, and then survey the systems which have been built. We also consider some research in program monitoring.

The first surveys of program visualisation (abbreviated PV) appeared in the mid-1980s, generally as components of general reviews of the applications of graphics to programming [175, 47]. The most comprehensive survey is Myers' [151, 152], and Brown's and Stasko's dissertations [33, 199] also include detailed analysis of previous systems. These surveys typically group systems into broad categories according to the type of displays they produce — whether they portray a program's code, data, or algorithm, and whether they display static or dynamic images.

More recent surveys have attempted to evaluate the field, rather than simply presenting an overview of work [35, 207, 173]. Typically several axes or categories are defined, describing attributes of PV systems. Real or hypothetical systems are then evaluated against those definitions. Several empirical studies of the effectiveness of program visualisation systems have also been carried out, particularly with regard to use in computer science education [71, 37, 206].

Much of the work cited in this chapter was first presented at the *IEEE Workshop on Visual Languages*, and subsequently appeared in the *Journal of Visual Languages and Computation*. Glinert's collection *Visual Programming Environments* [81, 82] reprints many important papers.

The first section of this chapter describes a generic model of program visualisation. Section 2.2 then describes visual tools (such as graphical debuggers) which produce visualisations in terms of the programming language. We then describe *algorithm animation* systems, which are based upon either annotation (§2.3), or mapping rules (§2.4). The final section (§2.5) surveys the program monitoring techniques which have been used by program visualisation systems.

2.1   A Model of Program Visualisation

Several researchers have constructed conceptual models to describe the architecture of program visualisation systems. We use the architectural model developed by Stasko [199, 200] and Roman and Cox [186]. This model considers program visualisation to involve a mapping between the program to be visualised (the *target program*) and a visualisation to be produced (see Figure 2.1). We call this the PMV model after its three components (*Program*, *Mapping*, and *Visualisation*). We use the term *actions* to denote the information about the target program gathered by the program component, and the term *changes* for the input to the visualisation component. For example, an action could record that a procedure named

insert had been called, while its corresponding change would inform a view that a new element had been added to a list.
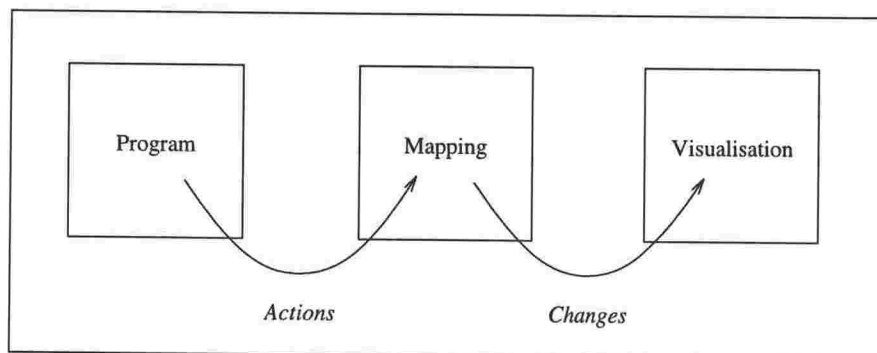
Figure 2.1: The PMV Model

Roman and Cox's and Stasko's presentations of the PMV model substantially agree. Stasko originally introduced the model to describe his TANGO algorithm animation system [200], and so he named the components *Algorithm*, *Mapping*, and *Animation*. Roman and Cox use the model to motivate the design of their PAVANE system (§2.4.1), and they also use it to analyse two other systems including TANGO.

A similar model underlies the taxonomy recently presented by Price et. al. [173]. The main difference between Price's model and PMV is that Price includes other components representing the context of the program visualisation system.

2.1.1   Program Component

The program component presents the target program to the visualisation system. Most PV systems work within a specific *target programming language*, and can in theory visualise any program in that language. Some (such as ANIM [18]) are applicable to programs written in several target languages, and a few (such as the ILLUSTRATED COMPILER [7]) only work with one specific target program.

The program component sends information to the mapping component describing the *actions* of the target program. Different PV systems characterise these actions in different ways. Some describe the target program in terms of its data structures, and others in terms of its control flow. Some systems produce *correctness* information about the actual values within the program, while others supply *performance* data, such as memory usage or execution speed.

Different PV systems not only require different information about the target program's actions but they also gather it in different ways. A common approach is to modify the program so that it provides data about its actions to the PV system — this is known as *annotating* the program. Alternatively, the target language's implementation or even the hardware executing the target program may be modified.

The program's actions must be communicated to the monitoring component of the system. Actions can be logged as the program executes, and then interpreted by the PV system in a *post mortem* analysis [40]. More commonly, the PV system and target program execute as coroutines, and when the PV system is executing the target program is suspended. If parallel hardware is available, actions may be transmitted to the PV system in a continuous stream.

2.1.2   Mapping Component

The mapping component receives information about the target program's actions from the program component, and sends *changes* to the visualisation component. While actions describe the target program in its own terms, changes package this information so that it can be understood by the visualisation component.

The mapping component varies widely across the systems studied, from GESTURAL [66] where a change is directly (and graphically) associated with an action in the program, to systems such as PAVANE [187] which use the power of a full programming language to perform the mapping. Whatever its design, the mapping component may perform the following tasks:

Translation The actions received from the program component may be translated so they can be understood by the visualisation component. This provides a measure of independence between the names and data types used in the target program, and those expected by the visualisation component.

For example, the program component may detect that the target program has called a subroutine named *swap*. The mapping component would then generate a change notification indicating that the program is performing a *swap* operation. A different program may name the corresponding subroutine *exchange* rather than *swap*. The mapping subsystem could translate this name and send a change named *swap* to the visualisation component. In this way, the visualisation component can be insulated from the precise details of the target program.

Selection The mapping component may focus the system upon particular actions of interest. The execution of a moderately sized program can involve billions of instructions and millions of bits of data. This is too much information to handle directly, especially as most of it is uninteresting at any given time. The mapping component filters this flood of information, presenting the visualisation component with only those changes relating to the parts of the program currently being displayed.

For example, the mapping component may receive an action describing each memory reference in the target program. If the visualisation system is displaying the values of a few variables, it should only receive changes describing assignments to those variables.

The mapping component often has to filter information, because the program component may collect unnecessary information. Ideally, the program component's monitoring of the target program would be sufficiently precise that only actions of interest to the visualisation component would be monitored. Unfortunately, most monitoring systems are rather blunt instruments, so some support for selecting actions is required in the mapping component.

Aggregation The mapping component may produce aggregate data. As with selection, this helps to ensure only relevant information is passed to the visualisation component.

For example, rather than produce an exhaustive trace of the target program's execution, the number of invocations of each subroutine may be accumulated to produce a procedure call profile. Although some information is lost (the precise sequence and arguments of each subroutine call), in many situations such a profile provides more useful information than a detailed trace, and is easier to analyse [87].

Abstraction Recovery The mapping component transforms the target program's actions into change notifications used to produce displays. Abstract displays such as those produced by algorithm animation systems (and our proposed program exploratorium) require changes expressed in terms of the abstractions they display, rather than the operations of the target program. For example, a view displaying a sorting algorithm requires changes comprising abstract *compare* and *swap* operations rather than procedure calls or memory references [35].

Generating these abstract changes is the most important task of the mapping component. The above techniques (translation, selection and aggregation) are used to redescribe the actions received from the program component, translating them from the domain of the program to that of the abstract visualisation. Much of the remainder of this chapter describes abstraction recovery techniques which have been used in previous work, including writing display procedures, annotating the target program, writing mapping rules, and using inferencing techniques.

2.1.3 Visualisation Component

The visualisation component handles the system's output, and input if any. It receives *changes* from the mapping component and uses them to draw and maintain the images comprising the visualisation.

Many PV systems can display several independent images simultaneously, each comprising a single *view* of the target program. Some visualisation systems allow the user to define target program specific views. This allows the user to tailor the appearance of the displays, and is required to produce target program specific views.

Each view may be drawn only once, or dynamically updated to produce an animated display. It is important to realise that static information may be displayed in a dynamic view, and vice versa. For example, systems such as ZEUS [36] and ANIM can produce *history* views containing a series of static snapshots of a dynamic view.

2.1.4 Rôles

The users of a program visualisation system fall into several different groups depending upon the way they use the system [33, 173].

- *Programmers* write target programs. In general, they do so without any knowledge that their programs are to be visualised.

- *Visualisers* construct visualisations of target programs, specifying the mappings and images to be used. Depending on the type of view, this may be a trivial task, requiring no *a priori* knowledge of the target program, for example as when specifying a simple display of the target program's structure. Alternatively, specifying an algorithm animation view can require a detailed knowledge of the target program and its implementation, so that the view can be tailored carefully to highlight the crucial properties of the target algorithm.

- Finally, *end-users* use visualisations to explore and investigate the target program.

The way these rôles are filled depends on the way a particular system is used. For example, programmers debugging their own work may fill every rôle at various times. For educational applications, where end-users may be inexperienced in using computers (let alone programming) there is a much greater separation of rôles.

2.2 Visual Programming Tools

In the practical application of program visualisation systems there is a "grand divide" between those systems used in production programming, and those used mostly for a few classroom exercises. This division is as follows: target program specific systems (algorithm animation systems) are used mostly in the classroom, while target program independent systems producing more generic displays (perhaps based upon a programming language) are used in production programming.

The main advantage of target program independent systems is that they make few demands on the visualiser, as they can be applied to any program in the target language without previously preparing the program, specifying complex mappings, or designing specialised images. A debugger (whether visual or textual) is used to find bugs in the target program in the easiest possible manner, often as a last resort. A visualiser is unlikely to expend much effort in learning to use a debugger, let alone in designing specialised visualisations.

This section describes several types of visual programming tools, which require little or no preparation of the target program. These include a variety of tools to develop and debug programs in traditional textual languages, completely visual programming languages, and tools to handle the maintenance of very large programs.

In terms of the PMV model, visual programming tools have a program component which is as close as possible to the production environment, but which sends some information to the visualisation tool. A visual programming tool's mapping component is typically very simple, allowing the end-user to choose what is displayed, but providing no abstraction recovery. Most of these tool's visualisation components are also very simple, and display only predesigned visualisations, although some (such as INCENSE [150] and CERNO-II [73]) do allow visualisers to design their own displays.

2.2.1   Visual Tools for Textual Languages

In this section, we describe visual tools for languages with a traditional textual syntax. We consider animated interpreters and various types of graphical debuggers.

Animated Interpreters

An animated interpreter displays the operation of an idealised interpreter for the target language. An animated interpreter may be used to understand the operation of an interpreter, or (especially where the language itself is complex) understand an interpreter's interaction with the target program. Several such systems have been implemented, including SEEPS visualising a Display Postscript interpreter [136], Lieberman's Three-Dimensional system [128] which displays the execution of programs in a LISP dialect, and TPM which animates PROLOG [70]. These systems generally illustrate the execution of the target program in great detail. For example, TPM displays all aspects of a PROLOG program's search tree, including backtracking and the incremental binding of logical variables via unification.

The views produced by animated interpreters are usually drawn automatically by the system and are not configurable by the end-user. They are usually generated in real-time as the target program is run. TPM is the exception to this, as its displays may be generated as the program runs, or built from a post-mortem trace. TPM also supports several different views. This flexibility, and its portable implementation, may be why TPM is one of the few program visualisation systems to find widespread use [173].

Graphical Debuggers

Graphical debuggers illustrate the execution of the target program in terms of the basic objects and operations provided by the target language. They perform no abstraction recovery, as they display the target program's call stack and data using views produced automatically. For example AMETHYST [153] can display all PASCAL's data types including records, arrays, and pointer structures. In this way AMETHYST can visualise most of the data structures found in PASCAL programs. Similar (although less comprehensive) systems have been implemented for other programming languages [167, 14], and are now becoming commercially available.

Graphical debuggers generally cannot produce animated or continuous displays. Like a conventional textual debugger, graphical debuggers wait until the target program's execution is suspended (typically due to a user interrupt or breakpoint) and then retrieve the data to be displayed from the target program's memory space. Graphical debuggers have been built by extending existing textual debuggers [194]. In such systems, the functions of the program component are carried out by the textual debugger, the mapping component provides an interface to the textual debugger, and the visualisation component displays the retrieved data. Sophisticated graphical debuggers are now available commercially [223, 14].

Debuggers can also display a program's control flow, either statically or dynamically. Flow of control through various functions, objects, or classes within the target program has been visualised by tracing a path representing the execution stack of the current thread through a graph describing the program's structure in systems such as GRAPHTRACE [116], GROOVE [193], and Cunningham Diagrams [55]. TRACK [22] provides a detailed graphical formalism for both specifying control flows of interest and monitoring the program's execution. The OBJECT VISUALIZER [165, 166] introduces several interesting displays which combine information about an object oriented program's control flow, data structures, and program performance, and has inspired several similar systems. Of course, static graphical representations of control flow have long been displayed by traditional program profilers such as GPROF [87].

Section 11.5.4 compares several object oriented graphical debuggers with the prototype system which we developed as part of this project.

Parallel Debuggers

Several graphical tools have been designed to assist debugging parallel or distributed programs. These include multi-process versions of control flow displays, which often include extra information about the

state of monitors, queues, and processes: for example, see PPT [162] and PARADOCS [172]. These are known as *correctness* debuggers in the parallel processing community, to distinguish them from the more numerous *performance* debuggers, which display information about the target program's performance. For example, PIE [126] and PARAGRAPH [216] can present various displays of the target program's effective parallelism, including Gantt charts and Kiviat diagrams as well as barcharts and histograms.

User Defined Displays in Debuggers

Although most graphical debuggers allow the user to select which data are to be displayed, they usually provide only one graphical presentation of the data. The mapping component of AMETHYST and the other systems described above simply queries the program component to determine the data to be displayed, and then sends this data to the visualisation component. Some systems do include support for a visualiser to define specialised pictures to display their data. In INCENSE [150], one of the earliest graphical debuggers and AMETHYST's direct predecessor, a visualiser defines *artist* procedures which map between the actual program data structures and the graphics to be displayed. These procedures recover abstractions by traversing the target program's data structures and generating calls to a graphics subsystem.

Several systems have retained the facilities for user-defined pictures but without requiring the user to write the code to generate them. VICK, a visualisation construction kit for SMALLTALK [24], includes a graphical editor which a visualiser can use to build a composite display by combining smaller display components. The individual display components are created by programming. Unlike most data visualisation systems, VICK produces continuous displays, as it uses TRICK (§2.5.2) to monitor the target program. CERNO-II [73], a graphical debugger for the object oriented PROLOG dialect SNART, uses a declarative language to specify the displays to draw. The FIELD [181] programming environment includes GELO [183], a multipurpose programming-by-example graphical layout tool, which can describe visualisations of program data structures.

2.2.2   Visual Programming Environments

Graphical debuggers and animated interpreters present information about the dynamic behaviour of the target program. Graphics can also be used to illustrate a program's static structure. Many interactive programming environments (such as the SMALLTALK environment [84]) maintain complete representations of the target program's structure, but do not present these graphically. More modern programming environments, however, are making increased use of visualisation and graphical manipulation [5].

PECAN [178, 179] is the first programming environment to make large-scale use of graphical displays. A syntax-directed editor is used to enter the target program in PASCAL. PECAN then provides several views which display the program's algorithmic structure using various visual syntaxes such as conventional flowcharts, Nassi-Schneiderman diagrams, and expression parse trees. Other PECAN graphical views display the program's source code, type structure, and symbol table. PECAN also includes textual views of the program's runtime stack and data.

FIELD [181, 180, 182] is a successor to PECAN, built as a collection of independent tools which cooperate via messages distributed from a central server. FIELD incorporates many specialised tools, including an annotation editor, several debuggers, cross-referencers, call graph and profile visualisers, as well as the GELO data structure display tool (§2.2.1) and the TANGO algorithm animation system (§2.3.1). All FIELD tools communicate by broadcasting events via the central server. For example, when a debugger notices that the target program has stopped at a breakpoint, it broadcasts messages telling other tools that the program has stopped and the location of the appropriate source code. Upon receiving these notifications, the editor highlights the current line in the program, and data structure displays are updated.

The SPE environment [89, 91] is a more recent system which is built using a framework for constructing multi-view environments. Like FIELD, SPE integrates a variety of different tools, and includes both graphical and textual views.

PECAN, FIELD and SPE all produce multiple views of the program under development, generally including source code text, graphical debugger-like diagrams of data structures, and statistical charts for

profile information [143]. The programming environments for the SELF programming language — SEITY [45, 46] and the SELF UI [197] — take the opposite approach, and produce only a single display of the whole program. The display is similar to that of a graphical debugger, but it includes program source code and can be edited to develop SELF programs. These systems try to provide an environment where the programmer thinks of the objects in the display as the ultimate representation of the program, rather than displaying information about a program hidden inside the environment — for example, objects are displayed as three-dimensional boxes. SEITY also uses cartoon-like animation to increase the impression of solidity [45].

2.2.3 Visual Programming Languages

Visual programming languages, such as PROGRAPH [54, 86], LABVIEW [13], and SERIUS [123], have a visual syntax, rather than a traditional textual syntax. A program in a visual programming language is an arrangement of figures on a page, rather than a stream of text. In one sense, visual programming languages and their programming environments perform a kind of program visualisation by default: these systems have an internal representation of the program, and present it graphically to the user. A visual language differs from a visual environment for a textual language because the language's visual syntax is the primary means of manipulating the program.

The *1994 Visual Languages Comparison* [95] describes an exercise where several visual languages were used to solve three small programming problems — the Sieve of Erastothenes, balancing a checkbook, and simulating a wagon wheel rolling down an incline. Solutions to all of the problems were implemented in at least two visual languages, with the Sieve problem being implemented in five, as well as the textual language C. Each of the solutions for the Sieve problem required roughly the same number of operators — around 13 — regardless of language. This included the C solution, which took up markedly less screen space than any of the solutions written in visual languages. The visual language solutions generally made the fine details of data and control flow more explicit than the textual solution.

Programmers have been slow to adopt visual languages. A variety of reasons have been suggested for this lack of interest — from the inefficiency of early visual languages and the amount of screen space they required to display programs, to programmers' fear that the increased clarity afforded by visual syntaxes would remove the esoteric mystique of their trade [53]. More obviously, since most programmers have not been taught visual programming languages, they would have to learn them before they could use them. Programmers would also have to learn the specialised editors and programming environments which visual languages require [98, 152].

SKETCHPAD and PYGMALION

The earliest visual programming language (and one of the first interactive graphical systems of any type) is Ivan Sutherland's SKETCHPAD [212]. SKETCHPAD programs are diagrams containing purely graphical symbols (the points and lines of Euclidean geometry) to which the user attaches constraints. These constraints are again purely graphical, requiring, for example, that two lines should be parallel, or should have the same length. SKETCHPAD programs are run whenever the user moves or alters an element of the diagram. SKETCHPAD then solves the constraints by relaxation, moving diagram elements to ensure the constraints are maintained.

David Smith's PYGMALION [196] is another influential early visual programming language. Like SKETCHPAD, PYGMALION programs have a purely visual syntax, although PYGMALION introduced the now ubiquitous *icons* — graphical signs which represent concepts other than their appearance. For example, PYGMALION includes icons which represent assignment and selection statements. Executing a PYGMALION program, as in SKETCHPAD, is primarily a graphical operation, in that the graphical picture of the program evolves to reflect the program's execution. For example, when a user-defined icon (similar to a user-defined procedure) is executed, the screen is cleared to show the icon's definition, and when the icon's execution has completed, the screen is restored to show the icon's original context, and execution proceeds in that context.

Systems like SKETCHPAD and PYGMALION naturally provide program visualisation. These systems' programs and data are graphical figures, and as programs are run, the graphics change to reflect the execution of the program. These visualisations provide no abstraction — all displays are in terms of the languages' visual syntax.

Visual Syntaxes for Traditional Languages

The early visual languages such as SKETCHPAD and PYGMALION, although highly graphical, are not particularly practical programming languages — SKETCHPAD programs are limited to diagrams with attached constraints, and, while PYGMALION does include more conventional control and data structures, its implementation is limited to very small programs. To create more powerful visual languages, visual syntaxes have been attached to existing textual languages. For example, TINKERTOY [67] provides a visual syntax for LISP, THINKERTOY [92] and MOLIÈRE [27] provide visual syntaxes for SMALLTALK, and C^2 [83] provides a visual syntax for C. These visual syntaxes use the semantic models of the underlying textual languages, so the resulting visual languages provide the full power of general purpose programming languages.

Most visual syntaxes for traditional languages are static, and quite similar to the flowchart or Nassi-Schneiderman diagram views produced by PECAN (§2.2.2). Visual syntaxes for traditional languages can, however, also incorporate dynamic animation of running programs. For example, the PICT flowchart language [83] provides execution visualisation using coloured highlighting to illustrate control flowing through the program flowchart.

Modern Visual Languages

Modern visual language research has focused upon designing visual languages from scratch, producing visual languages which do not depend upon existing textual languages. Some of these languages, in particular PROGRAPH [54, 86, 191] and LABVIEW [13] are in commercial use [41].

PROGRAPH and LABVIEW are both dataflow languages: a program is represented as a graph, within which nodes represent operations which transform data, and edges carry data between operation nodes. As with other visual syntaxes, this representation of a program makes the static algorithmic structure explicit. This is particularly important for dataflow languages, because a program's topology is a graph, which is difficult to describe using a conventional textual syntax. PROGRAPH and LABVIEW also provide procedural abstraction — a subprocedure can be defined by its own diagram, and then invoked by a single dataflow node in other diagrams. Procedural abstraction can be used to reduce the amount of extra screen space required by a visual program, because programs can be subdivided along natural boundaries so that every procedure can be displayed in a single screenful [53].

Visual languages can use visual syntaxes for purposes other than simply displaying the program's code. For example, a LABVIEW dataflow procedure can be displayed as a *front panel* (modelled on the front panel of a laboratory instrument) with dials, sliders, knobs and buttons which display data values as the program executes. PROGRAPH similarly provides *application builder editors* which can be used to graphically edit objects representing user interface components [191]. Visual languages can provide dynamic visualisation. For example, PROGRAPH's graphical views can illustrate control flow by highlighting the program's visual syntax. PROGRAPH's programming environment includes graphical debugger style views which illustrate the program's data structures, module organisation, inheritance hierarchies, and call graph. These displays, like LABVIEW's front panels, are updated dynamically as the program executes.

Some visual languages specifically address the visualisation of visual language execution. VISAVIS [171] is a visual functional language which includes higher-order functions. VISAVIS programs have been visualised in a manner similar to those of PYGMALION: as the computation proceeds, icons representing functions to be evaluated are replaced by the function's definition with arguments instantiated, and eventually the function's result is substituted into the calling context [170]. The visualisation models the definition of the VISAVIS language as a term-rewriting system, and can be controlled by controlling the precise rewriting strategy used.

The CHEMTRAINS language [15] is another rewriting system. A CHEMTRAINS program is a set of rewriting rules which are applied to a data store, the *arena*. Both CHEMTRAINS rules and the arena data store are defined graphically, and consist of graphical objects (text, boxes, ovals, and sketches) connected by unidirectional or bidirectional graphical links. When a CHEMTRAINS program runs, it applies the rules which change the contents of arena. Displaying the changing arena visualises the evolution of the program's data structures. This is in contrast to the visualisation in VISAVIS, which illustrates the rewriting of the program and so focuses on algorithmic structure.

Visual Programming and Program Visualisation

Perhaps the most important difference between visual programming and program visualisation is that visual programming focuses upon writing programs from scratch, while program visualisation focuses upon displaying existing programs. A visual program is usually defined by a particular visual representation, designed by the programmer when the program is written. A change to that visual representation generally implies a change to the program. In contrast, program visualisation displays do not generally depend upon the way the programmer wrote a particular program, and visualisations may be created, modified, and discarded without affecting the program.

In spite of these differences, visual programming and program visualisation obviously have much in common, as both are concerned with applying computer graphics to computer programs. Because of this commonality, it should be possible to combine both in a single system. Visual languages like VISAVIS and CHEMTRAINS which visualise the execution of visual programs exemplify one possible combination.

2.2.4 Visualisation for Maintenance and Reverse Engineering

Software maintenance and reverse engineering involve modifying programs after they have been placed into service [50, 233]. The modifications may be small localised changes, or they may globally reorganise a program's structure or even rebuild a program in a different programming language. Arnold's tutorial *Software Reengineering* [8] provides a general introduction to this area and reprints many important papers.

This section describes specialised visualisation techniques and systems which have been developed to support maintenance and reverse engineering. These techniques are able to display information about large commercial programs, which can incorporate millions of lines of code. Compared with the visualisation techniques described elsewhere in this chapter, visualisations for software maintenance are generally static views of a program's organisation and structure, because such views can be produced without any manual preparation of the target program.

Structural Visualisation

The most basic maintenance or reverse engineering views are module structure graphs and call graphs. For example, VIFOR [176] and CARE [130] display intermodule dependency graphs and structure charts for FORTRAN and C programs respectively — similar to the views produced by a programming environment. These systems parse a program's source code and then display the graphs.

A more powerful approach was pioneered by the C INFORMATION ABSTRACTOR (or CIA) [49]. The CIA parses target programs (written in C) and extracts structural and dependency information which is stored in a relational database. CIA collects a wide variety of information, such as the names of the files comprising a program and the #include relationships between them, the definitions of functions and variables and their locations, and the use of those functions and variables elsewhere in the program. To produce views, the user queries CIA's database to retrieve the information of interest, and then uses the stand-alone graph layout tool DAG [80] to visualise the results.

Any information in the database can be displayed, so CIA can display module dependency diagrams, structure charts, call graphs, and so on. The relational database means that CIA can store a large amount of information about the target program, while allowing the user to retrieve efficiently only that portion

of the information which is of current interest. Separating program analysis (filling the database) from visualisation means that independent tools can perform each function.

CIA has given rise to similar systems for different languages, for example CIA++ [88] for C++ and APAS [115] for ADA. A program information database and associated visualisation tools have also been incorporated into the FIELD programming environment (§2.2.2) [184].

Software Metrics

As well as visualising software structure, software metrics (such as complexity, volume of comments, or execution profiles) or development metrics (such as age of source code, or number of changes to a line) can be visualised.

SEESOFT [68, 69] was one of the first systems to visualise software metrics. SEESOFT's main display is a condensed view of all the source files of the program, drawn so that an entire program can fit onto a single screen. The display is coloured to visualise statistical information about the target program. SEESYS [12] extends SEESOFT to handle very large systems (up to a million lines of code). SEESYS uses a variant of *treemaps* [107] to present a large amount of information with a single display, and then animates these displays to illustrate the way a software system has evolved over time. SEE-ADA [195] takes a more basic approach to visualising software metrics. SEE-ADA displays call graphs and module dependence diagrams, and also uses colour to overlay program metric information onto these displays.

Reverse and Reengineering

Structural and metric visualisations indirectly assist reverse and reengineering by helping programmers find information about a program. More sophisticated tools can support the process directly, by producing editable visualisations which the user can manipulate graphically to restructure the program.

RIGI [148, 149, 232], for example, is a structural visualisation tool for C which displays the target program's module dependency graph — similar to CIA or CARE. In addition to browsing, RIGI allows the user to manipulate the dependency graph, by moving functions between program modules and creating new modules to group existing modules and functions. This is known as *modularisation* (or segmentation), and can be used to improve the program's structure by increasing cohesion within modules and decreasing coupling between them. After modularisation, the module structure should reflect the program's architectural design. Once the module structure has been rearranged, RIGI can rebuild the program source files to reflect the new structure.

COBOL/SRE [118, 119, 157] is similar in philosophy to RIGI, although it works with COBOL rather than C. COBOL/SRE can restructure the source code within function and data definitions, as well as moving whole functions between modules like RIGI. The most important difference between the two systems is that COBOL/SRE's display is based on textual source code, rather than dependency graphs. COBOL/SRE visualises programs by selecting and highlighting parts of programs based on a variety of *program slicing* techniques [222] which can precisely identify statements according to a variety of conditions. For example, COBOL/SRE can highlight only those statements which depend upon values of particular input variables (forward slicing), or those statements which determine the values of particular output variables (backward slicing).

The DESIRE [20, 21] system also uses program slicing to modularise C programs, but displays dependency graphs and call graphs as well as sliced source code. Both graphical and textual views can be used to manipulate the target program. DESIRE includes a connectionist knowledge base which can be used to locate candidate modules.

The SOFTWARE REFINERY [42, 135] is the most sophisticated reengineering system commercially available. Unlike the other systems described in this section, the SOFTWARE REFINERY is customisable, and has been adapted to handle programs in a variety of languages. The SOFTWARE REFINERY can be used for a variety of program manipulation tasks, including automatic translation between programming languages as well as program modularisation. The SOFTWARE REFINERY uses program visualisation in two ways — to display target programs, and to support its own customisation. To support reengineering,

the SOFTWARE REFINERY displays the call graphs, module dependency graphs, and sliced text of the target program. To support customisation, it displays the grammar rules and abstract syntax trees of the target programming language.

2.2.5 Summary

Visual programming tools are increasingly common, and several varieties (including graphical debuggers, visual programming environments, visual languages, and visual tools for program maintenance) are increasingly being used commercially. These tools typically visualise the target program directly at the level of the programming language, and have the advantage that the target program does not have to be specially prepared before it can be visualised. This is, of course, precisely what is required for debugging or analysing existing programs in detail. Because visual programming tools generally display low-level information, the amount of information presented can easily overwhelm a user interested in the higher-level aspects of the program [32].

2.3 Annotation

Producing higher-level views of a program requires that the abstractions within that program must be recovered by a PV system so that they can be displayed. The most common (and probably the most general) abstraction recovery technique is to modify the target program so that it automatically produces information at the required level of abstraction. Student programmers are taught to insert *write* statements into their programs at strategic places to display the control flow through the program and report the values of important variables. This strategy is also used by experienced programmers. The write statements can use arbitrary computations, and so can produce information about any abstractions within the program.

This technique can be adapted for program visualisation: the visualiser simply inserts calls to graphics primitives, rather than textual output statements [77, 120], so that when the program is executed these routines draw dynamic images of its operation. To make the results more permanent and accessible (an important concern when dynamic computer graphics devices were rare and expensive) the output can be recorded on film, as in Baecker's *Sorting out Sorting* [10], or videotape.

This approach has several drawbacks. The target program must be edited to insert the graphics calls, and the inserted calls obscure the original text of the program. To change the visualisation, the program must be edited again. This repeated modification can introduce bugs into either the program or the visualisation, and any abstraction recovery, graphical layout, or animation must be programmed explicitly.

To avoid these disadvantages, the program can be *annotated* or *instrumented* with *event markers*, rather than with direct calls to output statements. These annotations may be nothing more than stylised *write* statements or calls to PV system procedures which then notify the program visualisation system about the target program's actions, rather than directly producing graphical output. Programming environments can provide special support for these annotations, distinguishing them from the main text of the program by displaying them in the margin or in a distinguished font [181].

In most annotation-based PV systems, event markers are the only link between the target program and the visualisation system. This provides a measure of independence between the two, in contrast to graphical debuggers which may directly access the target program's memory. Provided annotations are maintained in correct positions, the rest of the program may be altered or even replaced, and the visualisation will continue to function. Similarly, the visualisation may be changed or replaced without reference to the target program.

This technique has many advantages. Any action of the target program that the visualiser considers significant can be monitored, provided the program's code is suitably annotated. If necessary, the structure of the target program can be arbitrarily modified to compute any information necessary for aggregation. No changes are required to hardware, the language processor or other software. Annotation

is relatively efficient, since the event markers are simply executed as part of the execution of the target program.

These advantages have corresponding limitations. Annotation is implemented by editing the program's code, and this is its greatest disadvantage: to annotate a program one must make semi-permanent changes to it. The annotations remain with the program, obscuring its source code. These effects can be reduced somewhat with programming environment support, but annotating a program requires an intimate knowledge of its structure and implementation. Event markers or print statements are very flexible monitoring tools because they can be positioned precisely to capture important parts of the program's execution. However, they generate misleading information if they are not positioned correctly.

In terms of the PMV model, annotation systems' program components are often simply the annotated target programs. Annotation systems' mapping components consist of the annotations in the programs which perform abstraction recovery. Some systems send information from annotations directly to the visualisation component, while others also include more complex mapping components, which can be used to transform the data generated by the annotations. Many annotation based systems employ sophisticated graphical animation systems as their visualisation components.

2.3.1   BALSA and successors

Marc Brown's BALSA and BALSA-II systems [33, 34] popularised the use of annotation in program visualisation. BALSA's target program is written in a dialect of PASCAL supporting independent modules. As BALSA is an algorithm animation system, BALSA simply terms the target program the *algorithm*. The algorithm must conform to BALSA's coding standards — it must be a single, self-contained module exporting several procedures with predefined names, and interesting event annotations must be added into its source code.

BALSA's annotations are written as PASCAL procedure calls. They have a name (the name of the procedure) and use the procedure's arguments to transmit information from the program. An executable BALSA system is built by a preprocessor which combines algorithm modules with the BALSA kernel and other modules which define visualisations. The annotation procedure calls are linked to *event routers* provided by BALSA's kernel. At runtime, as BALSA executes the target program, the annotation procedures call the kernel routers, which then invoke BALSA's visualisation component.

BALSA's Rendering Pipeline

Figure 2.2 illustrates BALSA's basic architecture, which Brown describes as an *object oriented pipeline*. The annotated algorithm's event markers generate *output events* when the algorithm is executed. These are converted by *adaptors* to *update events* which are sent to those *views* interested in the algorithm. A view consists of a *modeler* and a *renderer*.

Unlike the graphical debuggers described above, BALSA's views are not able to access their target program's data structure — views only communicate with the algorithm via events. A modeler uses the event notifications to construct a model of the program's data, and renderers use this model to draw and animate images using a low-level graphics package. The particular model built by a modeler depends upon the needs of the renderers that will use it: a model can contain any information received from the annotations in the algorithm.

Adaptors are used to translate between the events sent by the algorithm and those expected by the views. This is useful if one view is to visualise more than one algorithm and the algorithms do not generate the same events, and similarly if several different views are attached to a particular algorithm. Adaptors, modelers and views are written as modules in BALSA's PASCAL dialect and can perform any computation required for the visualisation.

BALSA's architecture may take more complex configurations. The data in a modeler may be shared between several renderers (see *Modeler-2* in Figure 2.2). Modelers may include *submodelers* which use the main modeler's data to construct their own more detailed model which is then used by further renderers. If the modeler's data is sufficiently complex, the code manipulating it may itself be annotated with event
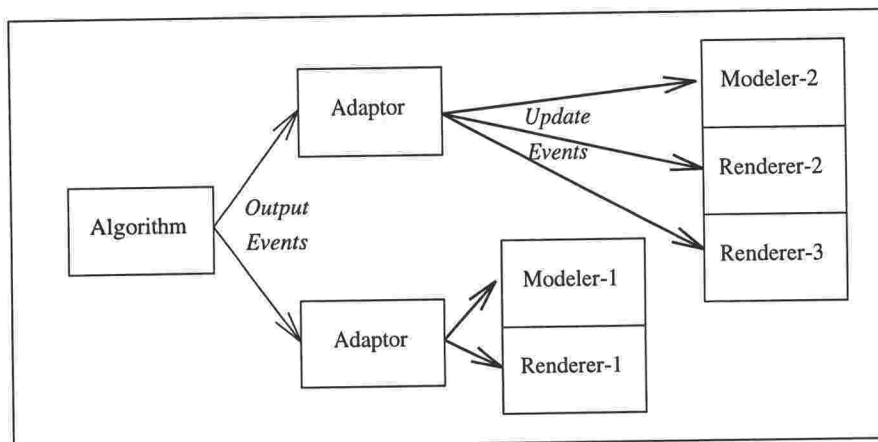
Figure 2.2: BALSA's architecture (from [33])

markers to allow fully independent views (i.e. further modeler/renderer combinations) to visualise this intermediate data.

BALSA was designed long before architectural models of program visualisation (such as PMV) were developed, indeed, BALSA's design has influenced the construction of the architectural models. BALSA's algorithms, adaptors and views fit the three components of the model, and its output and update events correspond to PMV's actions and changes.

BALSA's other features

BALSA is able to supply initial input to the target program, and allows the user to interact with the program being run. These features are managed by *input generators* — further PASCAL modules written by the visualiser. Input generators communicate with the target program by *input event* markers, which are similar to interesting event markers but which supply data to the target program. The programmer must replace any traditional I/O statements in the target program with input event markers. When an input event marker is executed, the input generator provides an appropriate input value.

BALSA also includes an interactive environment with the ability to run several algorithms in parallel, and support for recording, editing, and replaying interactive sessions.

BALSA's flexibility is bought at the price of complexity. A BALSA system is built by linking algorithms, renderers, modelers, adaptors and input generators using a preprocessor which processes the module constructs used to describe each component, and links the components to the BALSA kernel. Building and testing visualisations in BALSA involves a large amount of programming. This complexity does bring benefits. No other program visualisation system has been as ambitious in scope or provided a comparable environment.

Tango

TANGO [199, 201, 202] is a successor to BALSA. While BALSA's aim was to provide a powerful environment for algorithm animation, TANGO sought to reduce the complexity of constructing a visualisation, so that a program could be animated with no more expertise than was required to write it.

TANGO makes three main changes to the BALSA architecture. It is not a standalone program, rather it is integrated into the FIELD programming environment (§2.2.2). It uses FIELD's annotation editor to insert marks into the program, instead of editing code directly. Event messages are sent to TANGO's views via FIELD's distributed broadcast mechanism. Most importantly, the animated graphics are described using the *Path Transition* animation language [200] in place of simple graphics calls. This language describes both graphical figures that can be displayed, and paths that describe movements and transformation of these figures.

A WYSIWYG *dynamic graphics editor* DANCE, [203], can be used to generate much of an animation. Like a standard graphics editor, DANCE supports the design of graphical figures, but it also supports the direct entry of those figures' transition paths. The path transition animation definition subsumes the functions of BALSA's modelers, renderers, and graphics package.

TANGO has recently been extracted from FIELD and can function as a standalone program. XTANGO [208, 204] supports only a single view of an algorithm, and does not include the specialised annotation or animation editors, so it is used programmatically. Annotations are made with procedure calls as in BALSA. XTANGO has been used in teaching, and as it relies only upon the X window system and the C programming language it has been very widely distributed.

XTANGO is also one of the few program visualisation systems to undergo any sort of empirical evaluation. XTANGO was used to produce an animation of a pairing heap, and this animation was used as part of a lesson. The performance of students using the animation was compared against a control group who did not have the benefit of the animation. Unfortunately, probably due to details of the experimental design, the results were inconclusive [206].

Zeus

ZEUS [38, 36] is Marc Brown's second system after BALSA. Its main focus is upon extending program visualisation to include colour and sound. ZEUS's architecture is derived from BALSA, but simplified and written in MODULA-3. As in BALSA, the target program must be annotated with event markers and must conform to the system's structural requirements. ZEUS combines BALSA's modelers and renderers into unitary *views* and does not include any adaptors. It also supports input and has been used as the substrate for implementing the multi-view editor FORMSVBT [9].

The most interesting change from BALSA is that views in ZEUS are able to access the target program's data structures. This reduces the need for explicit modelers, but at the cost of increased coupling between the application and view. This is especially problematic because MODULA-3 is a parallel language and ZEUS is able to visualise parallel programs.

ZEUS has been extended with support for alternative graphics languages and 3D graphics [154]. It has also been extended to support some simple graphical debugger features — displays of variable's values and the source text of the executing program [37].

2.3.2   Other Annotation Systems

Many systems since BALSA have used annotations to recover abstractions for visualisation.

Anim

Jon Bently and John Kernighan's ANIM [17, 18] is a very simple program visualisation system — perhaps a case of 20% of the work providing 80% of the benefit. On first inspection, ANIM appears trivial: its main component is a simple script language which can describe pictures containing lines, text, boxes and circles. The target program is annotated to write a trace in this language, using the target language's standard output routines, so ANIM annotations are the target language's write statements. ANIM's utility comes from two factors: first, two flexible tools are presented to view the scripts, and second, the scripts can be manipulated post-hoc using standard UNIX tools.

ANIM's script viewers are MOVIE and STILLS, which (as their names imply) provide dynamic and static views respectively. MOVIE allows the script to be presented a various speeds, paused, and even replayed backwards. MOVIE can either connect to a running program using a UNIX pipe, or read a saved script file. STILLS is a TROFF preprocessor which can easily insert frames of ANIM animations into paper documents.

Because the script language is very simple, scripts can be manipulated using standard UNIX tools. For example, simple AWK programs can be used to interleave two separate scripts to generate algorithm

races, or even to simulate 3D visualisation. As a last resort, the script files can be fine tuned by hand, using a standard text editor. This has proved useful when the system is being used in practice, to present a program via a visualisation, rather than merely to demonstrate yet another program visualisation tool [18].

The Animation Kit

The ANIMATION KIT [133] is an extension to SMALLTALK for algorithm animation. The target program is annotated with interesting event markers, which the ANIMATION KIT represents using the change notifications of SMALLTALK's dependency mechanism [85]. These are used to indicate general points of interest in the program's execution and to monitor particular implementation variables. Views are programmed using an extended version of the standard SMALLTALK MVC library [122], and when the target program runs, the dependency mechanism ensures the views are updated.

Animus

ANIMUS [65, 30] is an extension to the THINGLAB constraint-based simulation system [28, 29] to handle dynamic simulations and program visualisation. ANIMUS extends THINGLAB's static constraints with trigger and temporal constraints which can be used to produce animations.

ANIMUS's target program is annotated with trigger constraints. A trigger constraint can be added to any object by the ANIMUS programmer, and specifies the graphical response to be performed when the object receives a particular message. The graphical actions may take place over a period of time, using ANIMUS's temporal constraints. THINGLAB's underlying constraint system can also be used to link the graphics to variables in the underlying program.

ANIMUS uses THINGLAB's basic object structure, and programs to be visualised must conform to this structure. ANIMUS's constraints are implemented by a preprocessor (§2.5.4) which transforms this structure into SMALLTALK without constraints.

The Illustrated Compiler

The ILLUSTRATED COMPILER (ICOMP) [7] is a single animated program (a PL/0 compiler) and an interactive environment for exploring that program, rather than a system for visualising programs. It is interesting because it is probably the largest visualised program presented in the literature.

ICOMP presents approximately twenty different views of the PL/0 compiler, covering its scanner, parser, code generator, and an interpreter running the compiled program. End-users can choose which views they wish to see, and control the execution of the compiler using the underlying facilities of the INTERLISP debugger.

The compiler is implemented in INTERLISP-D. The graphics are drawn using INTERLISP's window manager and graphics routines. The compiler source is annotated with *hookpoints* (LISP function calls) which act as annotation markers. The graphics routines are then invoked using INTERLISP's ADVISE facility. This allows *advice* — arbitrary LISP code — to be associated with a preexisting function. When an advised function is called, the advice is executed as a side effect. ICOMP uses advice to call the appropriate visualisation functions whenever a hookpoint is evaluated. A single function can have multiple pieces of advice, so multiple views can receive notifications from each hookpoint.

2.3.3 Direct Annotation

The annotation based systems described in the previous sections all require that the visualiser annotate the program, and then design and program the visualisation. This section describes several approaches to making annotation and animation of programs easier. Typically this involves three extensions to the tools presented above: a simple method of describing the annotations to be made (as in TANGO's annotation editor), a method of describing the visual effects to take place when the event marker is executed (similar

to, but more extensive than Dance), and a method of dynamically inserting the markers into the target program. We call the systems grouped in this section *direct annotation* system for two: reasons: the annotations and visualisations are defined using direct manipulation user interfaces, and the annotations are linked directly to the visualisation.

Gestural

Robert Duisberg's Gestural [66] system was probably the first to provide fully graphical specification of program visualisation. It is built on top of Smalltalk, and visualises standard Smalltalk programs. Gestural uses a dynamic graphics editor both to edit the graphical images used in animations, and to capture the user's gestures for moving or reshaping these images. A musical score-like editor can be used to combine several gestures into a single unit, for example allowing two images to simultaneously exchange position.

Once the gestures have been defined, they can be connected to the target program by "pointing and clicking" on some part of the program's text. Gestural then invisibly modifies the Smalltalk program to invoke the gesture when the annotation is executed. Annotations are shown by changing the font with which the program text is displayed. Similar manipulations can be used to bind the end points of the gesture to variables or expressions in the program. When the program is run, the modified program invokes the gestures and the animation is generated.

The Gestural implementation is very much a prototype, working with a single procedure in the context of a single object. Its graphical vocabulary is limited to black rectangles of varying sizes. Gestural provides only basic support for user defined data types — some limited handling of array accessing was built in. It does however demonstrate that simple and effective visualisations can be built without any textual programming.

Lens

John Stasko's Lens [147] is an extension to XTango which is very similar to Gestural. Lens also uses a dynamic graphics editor to specify images and manipulations, and an annotating editor to present the target program.

To design a Lens animation the visualiser first creates the graphical objects, each of which may be bound to a variable in the target program. Lens' graphical vocabulary is a specially chosen subset of Tango's but as it includes rectangles, circles, lines, and text it is much richer than Gestural's. Once the graphical objects have been defined, the visualiser may select the operations to be applied when the program reaches a particular point. Graphical objects may be moved, exchange positions, or change colour.

To display the animation, Lens invokes a version of XTango. The target program is run under the DBX [131] debugger, with breakpoints in the program wherever the user has annotated the source. Whenever a breakpoint is reached, Lens locates the associated annotation and sends commands to XTango to perform the appropriate animation.

2.3.4 Summary

Annotation is the abstraction recovery technique most commonly used in program visualisation, especially in algorithm animation systems. A good understanding of a program is required to annotate it, but given this understanding, annotation can be quick and easy to perform. Of course, annotation has several problems: the target program must be modified to insert annotations, annotations obscure the original program code, and annotations must be precisely positioned in the program. The annotation based approach is now quite mature, so continuing research into annotation based systems is now focused upon the overall usability of the systems, and the graphical details of the animations they produce, rather than their general architecture.

2.4   Mapping and Inference

Mapping based PV systems are an alternative to annotation based systems. Mapping based PV systems monitor the low-level actions of the program, and then use inference rules to determine the higher-level changes from the program's actions. Mapping can in principle avoid many of the problems of annotation — in particular, the visualiser need not annotate or modify the program. The program can be changed without affecting the visualisation, as there are no annotations which must be preserved.

Mappings can be defined declaratively. Declarative mappings, according to the proponents of this approach, should be simpler to specify than annotation based displays which depend upon the target program's procedural behaviour. Some declarative mappings can be inverted, in which case a single definition can provide both program visualisation and visual programming or editing.

2.4.1   Pavane

PAVANE [187, 185] produces 3D animations of parallel programs, and works within the parallel shared-dataspace language SWARM [187]. Animations are described by providing mappings between SWARM *tuple spaces* using SWARM rules. PAVANE begins with the program's state available in the *state space*.

The state space is first mapped into the *proof space*. This *proof mapping* must pass any relevant information about the program's data into the proof space. This space is called the proof space because PAVANE's authors believe that the properties of algorithms important in constructing their proofs should be presented in a visualisation. For example, a view of an array being sorted may distinguish array elements which are in their final position from those elements which will still be moved (§4.1.3).

The proof space is then passed through the *object mapping* into the object space. The object space contains idealisations of the graphical objects which will eventually be displayed. The animation is actually created by the *animation mapping* from the object space into an *animation space* which describes primitive graphical objects for rendering.

Although it appears to be based around a program's data, PAVANE can animate the control structure of the program. This is because a SWARM program's data and control information are both contained in the state space. The various mappings (especially the animation mapping) can access the previous versions of their input and output spaces, and can use these to accumulate historical information or detect changes which should be animated smoothly.

2.4.2   Trip

The members of the TRIP family of systems are similar in approach to PAVANE — they describe visualisations as mappings between different *structure representations*. The TRIP systems are written in PROLOG and manipulate structure representations as collections of asserted predicates in the PROLOG database.

TRIP1 [110] established the general architecture of these systems. Like PAVANE, visualisation is described by three mappings. First, a textual representation of the application's data is parsed into the *abstract structural representation*. This parse corresponds to PAVANE's proof mapping. The abstract structural representation is then mapped into the *visual structural representation*, a collection of predicates describing graphical objects to be presented to a constraint-based layout system. The visual structural representation is used to produce the actual images, in a similar manner to the PAVANE animation mapping.

TRIP2 [214, 137] extends TRIP1 with a *spatial parser* and *inverse structure mappings*, which allow the system to accept input. Diagrams produced by TRIP2 can be edited freehand. The spatial parser then recovers the visual structural representation, and the inverse mappings translate this back into the target language via the abstract structural representation. TRIP3 [144] uses graphical programming-by-example to build the mapping rules.

The various TRIPs do not provide the control visualisation or animation facilities of PAVANE, and their displays are only two-dimensional. TRIP's graphics system provides more layout support than PAVANE's

animation system, although PAVANE is better at 3D animation. Visualisations are probably easier to build in TRIP3 than PAVANE — PROLOG's DCG parser facility can be used to interface to any target language, and the visualisations can be defined graphically.

2.4.3 UWPI

The University of Washington Program Illustrator (UWPI) [96] is the most ambitious mapping rule based PV system yet built. It presents abstract visualisations of graph, list and array algorithms written in a PASCAL subset which provides integer variables and one and two dimensional arrays, but neither records nor pointers. UWPI statically analyses the target program to determine the abstractions it contains, and then uses the results to produce a visualisation.

UWPI's analysis proceeds in several stages. The target program is parsed, and stored as an abstract syntax tree. The tree is then traversed by a *statement pattern matcher* similar to a peephole optimiser, and then by a *subrange inferencer*. These look for clichés (recurring patterns) in the target program's use of variables. For example, the subrange inferencer may report that a variable's value ranges over the indices of an array, and the pattern matcher may detect that the variable is used to index into the same array.

The next stage, the *ADT converter*, uses this information to infer the way the program variables are used to implement abstract data types (ADTs). UWPI's rule base includes descriptions of about ten ADT's, including linked lists, queues, and directed graphs. Each variable is ranked against the ADT definitions and the most plausible ADT is chosen for that variable. For example, UWPI would conclude the variable described above is used as a pointer to the array's contents. UWPI's rules are ordered according to the generality of the ADT. If it is unable to deduce the precise type of a variable, UWPI assigns it a less specific type. In this way animations gracefully degrade when UWPI is presented with programs it cannot completely analyse.

Once all the variables' types have been identified, the *layout strategist* is invoked to generate a visualisation plan. UWPI determines the most important ADT and uses it for the backdrop of the visualisation. Other variables are then animated as pointers or lozenges moving over this background. In the array pointer example, the array would be the background and the integer would be animated as a pointer into the array. Once the visualisation plan is complete, UWPI interprets the target program and produces the visualisation. Variables in the target program are monitored by the interpreter and their values used to update the visualisation at runtime.

In spite of its analysis of the target program, UWPI is surprisingly efficient — it required fifteen seconds to analyse a small breadth-first search procedure, and visualised the procedure traversing a seven element graph in thirty seconds. The inferencer rule bases are resistant to small bugs in the target program, and can operate across families of algorithms — once a selection sort was successfully visualised, several other sorting algorithms could immediately be displayed.

The advantage of this approach is that the visualiser does not need to provide any specific information about the target program — UWPI's analysis attempts to recover abstractions automatically. The visualisation is independent of the names used in the target program, and the precise representation used for an ADT. For example, UWPI's PASCAL subset does not include Boolean types, and so Boolean values must be implemented by integers, the language's only scalar data type. UWPI is able to recognise several common integer representations of Booleans and visualise them appropriately.

Of course, knowledge about abstractions must be provided somewhere, and in UWPI it is embodied in the rule bases of the statement matcher, subrange inferencer, and ADT converter. UWPI thus shifts the burden from describing the abstraction structure of a particular program (either by annotating it or writing program specific mapping rules) to writing rules for recognising abstractions in whole classes of programs.

2.4.4 Summary

Program mapping has one large advantage over annotation: the visualiser does not have to annotate the target program. Writing data mapping rules for systems like TRIP or PAVANE does require detailed

knowledge about the target program, but is probably easier than inserting the corresponding annotations, especially as mapping rules are not sensitive to their position in the program. Writing abstraction detecting rule bases, as required for UWPI, is significantly more difficult, but should only need to be done once for each implementation of any abstraction.

Apart from this, neither annotation or mapping seems to have an advantage over the other in the *kinds* of visualisations that they can produce. The most sophisticated mapping system described, UWPI, is, however, able to generate abstract visualisations of some programs without any user or visualiser intervention. It is difficult to see how this could be performed with an annotation based approach.

2.5 Program Monitoring

This section surveys techniques used for monitoring the execution of the target program. We include the traditional hardware-based approaches used in debuggers, theoretical techniques using reflexive languages, and ad-hoc solutions which modify the program to generate information about its actions. Our presentation is oriented towards the use of monitoring techniques for supporting program visualisation. More detailed information is available from the comprehensive surveys which have appeared in the last decade [48, 156, 169, 205].

Although we present each technique in isolation, many actual systems use a combination of techniques. This is for reasons of efficiency, as no two techniques gather precisely the same information or impose the same overheads. One common tradeoff involves monitoring *definition* as against monitoring *use*. All calls to a particular function can be monitored easily by monitoring that function's definition. Monitoring every call site would produce the same information but is obviously more complicated. Conversely, if we wish to monitor a single call to the function, monitoring the definition will provide a large amount of spurious information, and may impose overheads upon other calls to the monitored function. This definition vs. use distinction applies to many other constructs, including data type definitions and variables, method definitions and message sends, class definitions and object instances, and whole objects or classes (including all their associated methods and variables) and particular messages or variables.

2.5.1 Hardware Monitoring

Computer programs to be monitored must eventually be executed by some hardware processor. By monitoring the operation of the hardware it is possible to monitor the program's execution without modifying the program in any way.

Monitoring facilities provided by general purpose processors (such as the VAX [61], 8086 [139] or 68000 [188] families) include instruction, memory access and timer interrupts, and special purpose internal registers. The interrupts cause the processor to execute a monitoring system routine whenever a particular condition occurs — when the target program executes an instruction, accesses a particular part of memory or after a specified time interval. The monitoring routine can then examine the execution state of the target program, generate action notifications, and then resume the target program. For example, trace interrupts can be used to invoke a debugger, thus single stepping the program; memory access traps (whether monitoring single addresses or entire pages) can be used to monitor data structures, and timer interrupts can be use to collect profile information. An internal register can count the number of times the CPU performs a particular operation, or record the utilisation of the CPU's functional units.

The main advantage of hardware-assisted monitoring is that the target program does not have to be changed in any way. Using some suitable interface, the programmer activates the monitoring and then analyses the information. It is also quite efficient for that precise subset of actions which can be monitored directly in the hardware. The disadvantage is that the monitoring is limited to those facilities provided by the hardware and supported by the operating system, and these vary widely from system to system. For example, the CADR LISP machine [145] can monitor one contiguous 32K block of memory, while 80386 family chips are able to monitor only four addresses, but these addresses may be anywhere in memory [139].

Unless the language processor is also extended to provide information about the program's translation, information will only be reported in terms of assembly language.

2.5.2   Postprocessing the Executable Form of the Program

Rather than modifying the hardware, similar results can be obtained by modifying the program's executable form – that is, postprocessing the language translator's *output*. Neither the hardware nor the language software need themselves be modified.

This technique is used by many debuggers (along with hardware monitoring), because the program can be changed *after* the language processor has completed its work, even after the target program has been loaded and run. Breakpoints are typically implemented by changing the program's memory image to insert a call into the debugger — when program execution reaches the breakpoint, the debugger is entered. By changing the compiled program, breakpoints can be inserted and removed without retranslating the program. The ACID [226] debugger makes particularly flexible use of this technique, as it provides breakpoints, coverage analyses, and execution profiles by editing the binary representation of the target program.

A disadvantage is that the only information this technique alone can provide is that contained in the program binary. A common solution is to modify the language processor to include source level information about the program in the binary file. Many language systems can generate symbol tables which can be used to direct the changes to the binary, allowing the user to work in terms of the source code.

This approach is not limited to "real" binary files to be executed by "real" hardware. Interpreters must maintain an internal representation of the target program — similar effects can be achieved if this can be modified. Structurally reflexive languages (§2.5.5) allow a program to modify its own internal representation directly. For example, the standard PROLOG debugger provides procedure spypoints by modifying PROLOG's representation of the procedure [31]. An encapsulator [164] or indirection [121] object is a SMALLTALK object which displaces an object within the target program, intercepts all the messages intended for the original object, and eventually forwards them to the original object. Encapsulators may be used for several purposes, depending upon how they process intercepted messages: they can enforce mutual exclusion or atomic transaction constraints as well as providing information to a monitoring system.

This technique has also been used directly in PV systems. For example, TRICK [23, 22] monitors SMALLTALK methods by replacing their internal representations with specially constructed wrappers which notify the visualisation system then resume the original methods.

This technique has the following advantages: the program's internal form can usually be modified without direct user intervention, and without having to reload or recompile the target program. If the unmodified program representation can be kept alongside the modified representation (as in TRICK and encapsulators), modifications can be hidden from queries about the program's structure by referring such queries to the unmodified version, and the modifications can be reversed easily when monitoring is no longer required.

2.5.3   Modifying the Language Processor

Working only with low level representations of the target program (either modifying the binary or using hardware facilities) has the problem that any information presented will be in those low-level terms, rather that in terms of the original source program.

More radically, the language processor may be changed to monitor the target program directly: if the processor is a compiler, to produce a program which monitors itself when run; if it is an interpreter, to monitor the programs it interprets. The translation or interpretation performed by the language processor is extended to include the monitoring. For example, a tally variable may be updated whenever a procedure is called, or a monitoring procedure may be called whenever a variable is referenced.

Several control flow visualisation systems have used interesting variations of this technique. GRAPH-TRACE [116] modifies the message dispatch function of the STROBE LISP dialect to build graphical traces of process execution histories. Cunningham and Beck [55] modified SMALLTALK's debugger [84] to collect similar information. The AMETHYST graphical debugger, UWPI, and most animated interpreters use custom interpreters specially designed to produce information about the target programs' control flow and changes in its data values.

This approach does not require any modifications to the target program or hardware, but of course the language translator itself must be modified. The changes in the processor output may be visible to lower-level tools. Overall, this is quite an efficient technique because the monitoring can use the information about the target program computed by the translator both to optimise the monitoring and to present source-level information.

2.5.4 Preprocessing the Program Source

The target program can be preprocessed before being sent to the language translator, and the preprocessor can insert statements to generate monitoring information as the program is run.

This technique has been used to build debuggers (or program visualisation back ends) for several languages. For example the portable SCHEME debugger PSD [113] uses this technique to trace the execution of a SCHEME program. Program source preprocessing is also used to support PROLOG visualisation in TPM (§2.2.1), and C++ visualisation in GROOVE [193].

Translating the target program's source has several advantages. It is easy to implement, as no major software or hardware components need to be modified. This visualiser need only specify what is to be monitored, and does not perform the modifications directly. The preprocessor itself can be portable between different target hardware or language implementations, since it works completely with the target language. The main disadvantage is that since the program is modified its behaviour may be altered, and care must be taken to display the original program to the user.

2.5.5 Reflexive Languages

A reflexive language is one in which a program can affect its own computation [134]. Reflexive languages work at both the *base* level — performing computation about the program's problem domain, and the *meta* level, which has the base level as its domain. The *base language* is that subset of the reflexive language concerned with the base level: the full reflexive language includes the base language and a *reflexive extension*.

The reflexive extensions of a language may be *structural* or *computational*. A program in a structurally reflexive language can inspect and alter its own structure. For example, a LISP program can dynamically create or delete functions, and inquire about the existence and status of variables. Other structurally reflexive languages include PROLOG and SMALLTALK.

In a structurally reflexive language, a program can alter its own structure but not the language's semantic model. While structurally reflexive languages cannot directly monitor the target program's execution, they can be used to support other monitoring techniques. In particular, a structurally reflexive language can easily modify programs to support dynamic monitoring. For example, PROLOG and LISP program tracers often use these languages' structurally reflexive extensions to add tracing statements to their target programs (§2.5.2).

A computationally reflexive language usually includes a structurally reflexive subset, but a program is also able to dynamically manipulate its own execution. In these languages, it is possible to alter the semantics of base-level programs as they are executing, typically by writing or extending a *meta-circular* interpreter — an interpreter for the extended language written in itself. A program visualisation system requires information about the target program and its execution — if the target language is computationally reflexive, this information can be obtained easily. For this reason, several program visualisation systems have used reflexive languages: the visualisation system and target program are

written in the same reflexive language and the visualisation system uses the reflexive facilities of that language to monitor the target program.

For example, the CLOS object system [112] includes a reflexive extension, the META OBJECT PROTOCOL or MOP [114]. CLOS is effectively implemented by a set of standard objects supplied by the MOP, and these can be altered or replaced to change CLOS's behaviour. The MOP's *slot* and *method* objects, which implement CLOS's variables and methods respectively, can be altered so that any slot accesses or function calls are reported to a visualisation system [94].

2.5.6 Summary

All these methods of monitoring programs have been used with program visualisation systems. Each provides information about the target program with an associated cost.

Modifying the target program (typically to perform annotation) is probably the most common method, it is certainly the simplest as it requires no specialised supporting hardware or software. It is quite intrusive, however, because the visualiser must manually perform the modifications. Automatically modifying the program's source, or, especially in a structurally reflexive language, its internal representation, is almost as easy to implement, and has the advantage that the visualiser does not have to modify the target program directly. Modifying the program's executable form is also popular, especially if existing tools, such as textual debuggers, can be employed.

Using a reflexive language, altering the language processor to provide monitoring, or using specialised monitoring hardware are progressively more complex options, but each of these provides further benefits. These methods can monitor an unmodified target program, and, since the language's implementation is cognisant of the monitoring, the monitoring can be carried out very efficiently. Hardware monitoring is the extreme end of this continuum: suitable hardware can monitor all aspects of the target program's execution without altering the program's behaviour or performance in any way. However such hardware is usually prohibitively inconvenient.

48. The best book on programming for the layman is "Alice in Wonderland";
but that's because it's the best book on anything for the layman.

Alan Perlis, *Epigrams On Programming* [168]

3

Abstraction

> *Abstraction: the decision to concentrate on properties which are shared by many objects or situations in the real world, and to ignore the differences between them. It is my belief that the process of abstraction, which underlies attempts to apply mathematics to the real world, is exactly the process which underlies the applications of computers in the real world.*
>
> C. A. R. Hoare, *Notes on Data Structuring* [99]

Abstraction is central to both programming and visualisation. In programming, we concentrate on the essential features of the program to be written, and map these to the implementation programming language or computer. In visualisation, we wish to draw attention to the important features of the data being displayed, and deemphasise the inconsequential.

Abstraction is crucial to programming because of the complexity inherent in even small programming tasks. Programs should not be constructed as monoliths: rather they should be designed piecemeal, ideally as a set of components each allowing consideration in isolation. The design of a program is a collection of abstractions that limit the complexity that must be considered at any time.

A program exploratorium should both illustrate and alleviate this complexity. It should illustrate this complexity because complexity is part of the very nature of programs, but it must also mitigate this complexity because any nontrivial program will otherwise be too difficult to comprehend.

This chapter attempts to answer some high-level questions. How is abstraction used in programming? How is abstraction used in program visualisation? How can a program exploratorium take cognisance of the abstractions within a program? The presentation here is necessarily somewhat idealised; following chapters discuss pragmatic details.

The first part of this chapter (Sections 3.1 and 3.2) investigates the rôle of abstraction in program visualisation with reference to the previous work reviewed in Chapter 2. The second part (Sections 3.3 to 3.6) surveys various programming paradigms with respect to their suitability for supporting the visualisation of the abstractions in programs' designs. Section 3.7 then presents a novel scheme for exploring programs based upon displaying the abstractions contained within them. Finally, Section 3.8 summarises the chapter.

3.1   Abstraction in Visualisation

We consider that three separate uses of abstraction can be distinguished in program visualisation systems:

1. **Visual abstractions** present information visually. Every program visualisation system, even the most basic graphical debugger (§2.2), uses visual abstractions to produce the graphics to be presented to the user.

2. **Program abstractions** capture the important ideas in the target program's design. A program's design is a structure of abstractions, and these abstractions are essential to understanding the program, so we call these abstractions "program abstractions". Program abstractions are independent of visual abstractions, and can usually be displayed in many different ways, using different visual abstractions. Program visualisation systems which illustrate higher-level aspects of a program (such as algorithm animation systems, §2.3, §2.4) must somehow recover abstractions from the program, so that they can display the important ideas inherent in the program's design.

3. **Aggregate abstractions** provide information about a program's structure or performance. Aggregate abstractions often synthesize information ranging across the whole of the program (thus their name), in contrast to program abstractions, which represent the design of a single program component. Aggregate abstractions, like program abstractions, are independent of a particular visualisation, and can be displayed in different ways. Parallel programming tools (§2.2.1) and visual programming environments (§2.2.2) typically display aggregate abstractions that summarise a program's performance.

We have found these categories a useful aid in understanding program visualisation, even though they are not rigidly defined. A particular visualisation will always use visual abstraction, but whether it in addition uses program or aggregate abstraction may be a matter of opinion.

The following subsections present each of these categories in turn, by discussing a series of visualisations representative of those in the literature [37, 152, 173]. The example illustrations have been generated using our Tarraingím program visualisation system, described in Chapter 5.

3.1.1   Visual Abstraction

Sorting algorithms were one of the earliest subjects of program visualisation [10, 39]. They are often visualised by displaying the data to be sorted, typically an array of numbers. As the sort progresses, the elements in the array are moved or exchanged, and the animation system updates the display of the array's elements.

Figure 3.1 shows three illustrations taken from a dynamic visualisation of a sort. Two views (the sticks histogram view and dots scatter plot view) present graphical illustrations of array element values plotted against their position in the array, and the third vector view simply lists the values of the array elements, in the order of their position in the array. In the figure, the array is unsorted — the dots and sticks views appear random, and the list of numbers in the vector view is in no particular order.

All these views display the same information about the contents of the array, but each view presents this information differently. The graphical views display the shape of the data especially well and, as the sort progresses, give an impression of the properties of various different algorithms. Precise values of individual elements can be more easily read from the textual view.

Each view is a separate *visual abstraction* of the array — a mapping from a lower level (the array in the program) to a higher level (the illustration). For example, the sticks view maps each array element to a filled rectangle. The element's value is mapped to the rectangle's height, and its position in the array is mapped to the rectangle's horizontal position in the view. The dots view is similar — an element's position is represented by the horizontal position of a square dot, while the dot's vertical position represents the element's value. The vector view simply maps each element's value into one or more character glyphs.

Visual abstractions can illustrate the behaviour of an algorithm, as well as the data it is manipulating. Several behavioural views of quicksort are illustrated in Figure 3.2. The call tree view illustrates the tree
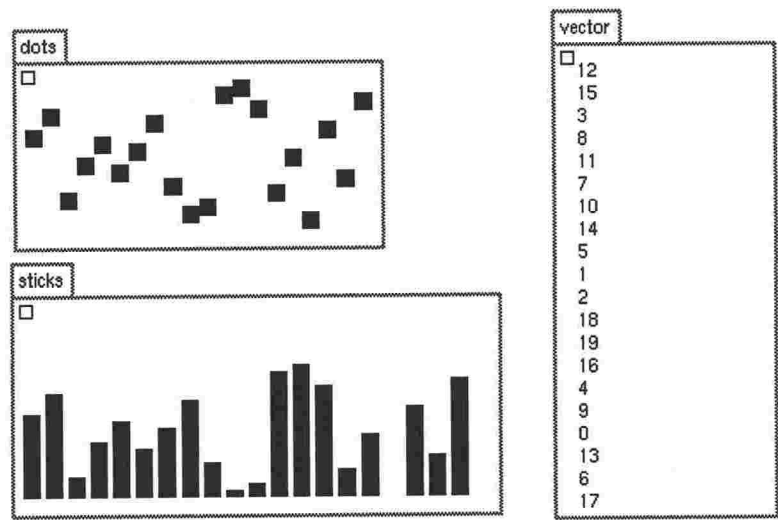
Figure 3.1: The Data within an Array

of recursive calls made by quicksort while sorting the array. Each box in the call tree view represents a call to quicksort, the numbers in the box being the bounds of the partition to be sorted. The partition view shows the progress of the sort — a curve is drawn to connect bounds of partitions of array elements imagined across the bottom of the view. The trace view displays a textual list of the recursive calls to quicksort, and the swaps of array elements. Each of these views is a visual abstraction of the underlying behaviour of the sorting algorithm.
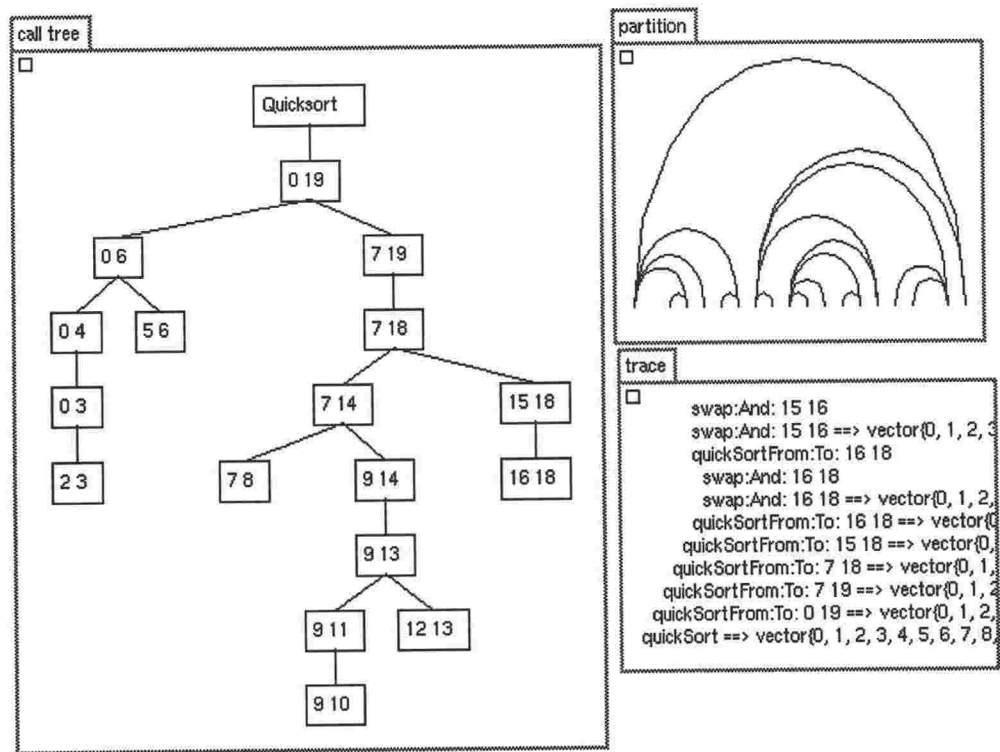
Figure 3.2: The Behaviour of Quicksort

3.1.2   Program Abstractions

The array data views in Figure 3.1 all display the same information about the array: the values of the array's elements. These views display an abstraction of the array, rather than the array itself. The array is being used as a sequence — an abstract list of numbers. Details of the actual array, such as the exact types, sizes, or addresses of the array elements, are irrelevant to the sequence abstraction and are omitted from the views. The behavioural views in Figure 3.2 similarly display essential information about the algorithm's execution: the partition or swap operations performed by quicksort.

The sequence is an example of a *program abstraction* — an important idea in the internal design of the program. Each view in Figures 3.1 and 3.2 embodies a different visual abstraction of the same underlying program abstraction. Note that these views could alternatively be described as different visual abstractions of the array, i.e., visual abstractions of a concrete programming language construct. We introduce the idea of an underlying program abstraction (the sequence) because all these views present the same subset of all the possible information about the array.

A program abstraction can be seen from two different perspectives: from its *interface* and from its *implementation*. Seen from its interface, an abstraction appears as a single unit which can be understood in isolation. An abstraction's implementation describes how the interface is realised. The views in Figures 3.1 and 3.2 rely only upon the sequence interface, not on any particular implementation details, so they can be drawn for any kind of sequence, or any implementation of the quicksort algorithm sorting a sequence.

Some implementations (such as arrays) are *primitive*: they are built into programming languages. Others are *constructed* by combining simpler abstractions. A linked list can be built from link records, or a hash table from an array. A sequence can then be used to implement a stack in a parser, or to contain the terms representing a polynomial.

In this way, a program is built as a hierarchy of abstractions. A few key abstractions, such as arrays, numbers, and characters, are supported directly by the programming language, while the bulk of the program consists of abstractions constructed by the programmer from the language primitives. This abstraction hierarchy describes part of the design of the program.

Figure 3.3 illustrates these multiple levels of abstraction. The stack abstraction view shows a push down stack of integers. Each stack element is displayed by the length of a horizontal bar. The stack currently contains seventeen elements, with the top element (numbered 0) drawn at the top of the view.

The stack implementation view shows how the stack is implemented. This view has three subordinate views. The first of these, labelled components, contains icons for the main components of the implementation: a parent which supplies the stack's operations (§5.4.4), an array holding the stack's contents, and an integer index. The other two views display these components. The contents view displays the contents array as a sequence in the same way as the bar view from Figure 3.1: the height of the bar gives the value of the array element, and elements are drawn from left to right. The index view illustrates the value of the index component as an index into the contents array.

The contents implementation view shows the primitive vector that implements the stack's contents array. This vector also contains a parent component, and components for each array element.

Several interesting features of the stack implementation can be gleaned by comparing these views. For example, the value of the index variable is the size of the stack. The contents array contains twenty elements (the stack only seventeen), so presumably when an element is removed from the stack it is not removed from the array, but the index is adjusted so that it is inaccessible to the stack operations. This can be seen quite clearly by watching the evolution of the display as elements are added to and removed from the stack. Finally, the order of items in the stack is the reverse of that in the array. The stack is "pushed down" the screen, so that the element most recently added is on the top of the stack. Elements are stored in the array in its natural order (drawn left to right), so that the top stack element can be anywhere in the array (actually one element to the left of the index), and the first (leftmost) array position holds the bottom stack element. To traverse the stack in its natural order, the portion of the array containing valid stack elements must be traversed in reverse order.

The stack abstraction and stack implementation views present the interface and implementation perspectives of the stack abstraction, and the stack's contents component view and the contents implemen-
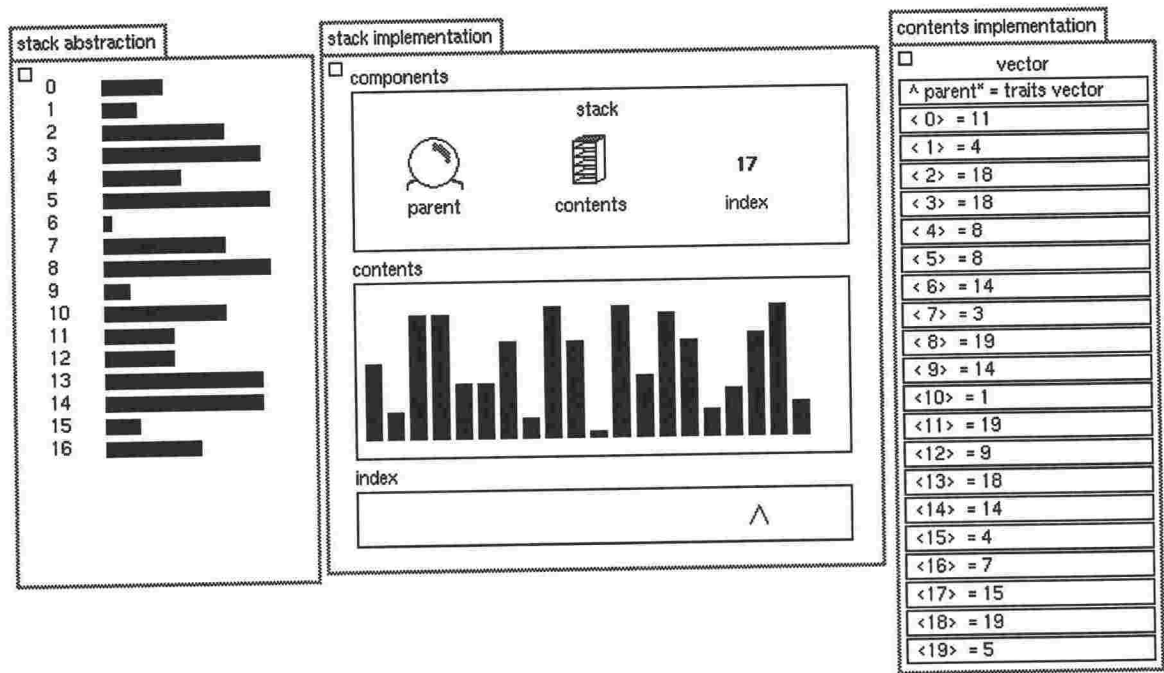
Figure 3.3: A Stack Abstraction and Implementation

tation view present the corresponding perspectives of the stack's component array. Note that these two perspectives display different information about an abstraction. In contrast, different visual abstractions of the same program abstraction present the same information in different ways.

A program visualisation system should be able to display both perspectives of program abstractions at any level in the program's design. In this way, the user can compare displays of abstractions and their implementations to see how the program is constructed from a hierarchy of abstractions.

3.1.3 Aggregate Abstractions

Many program visualisation systems produce views of programs which are neither displays of program abstractions nor displays of the program's code and data. Often these views focus upon the *form* of the program, as against views showing program abstractions which display the program's *content*. Such views may display the structure of the target program or performance information, and often use statistical data reduction and visualisation techniques to display large amounts of information. We say such views display *aggregate abstractions*, since they display information collected from large parts of the program. Views of aggregate abstractions are more common in programming tools or environments (§2.2.2) than in graphical debuggers or algorithm animation systems.

Figure 3.4 illustrates several views of aggregate abstractions. The abstraction structure view displays the relationships between abstractions within a small recursive descent parser. The parser uses a lexical analyser (lexer) and a stack; the lexer uses an input stream; and the stack (as in Figure 3.3) uses an array (vector) and an integer (smallInt). The operation profile and read/write profile views display some performance information about the stack. The operation profile displays a histogram of the number of times the stack has performed each operation, and the read/write profile shows how many times each element of the array implementing the stack has been read (light bars) or written (dark bars).

Figure 3.4 also shows an important point about aggregate abstractions: like visual abstractions, aggregate abstractions are often expressed in terms of program abstractions. For example, the operation profile view displays performance information about the stack program abstraction, and the abstraction structure view explicitly displays the structure of program abstractions making up the target program.
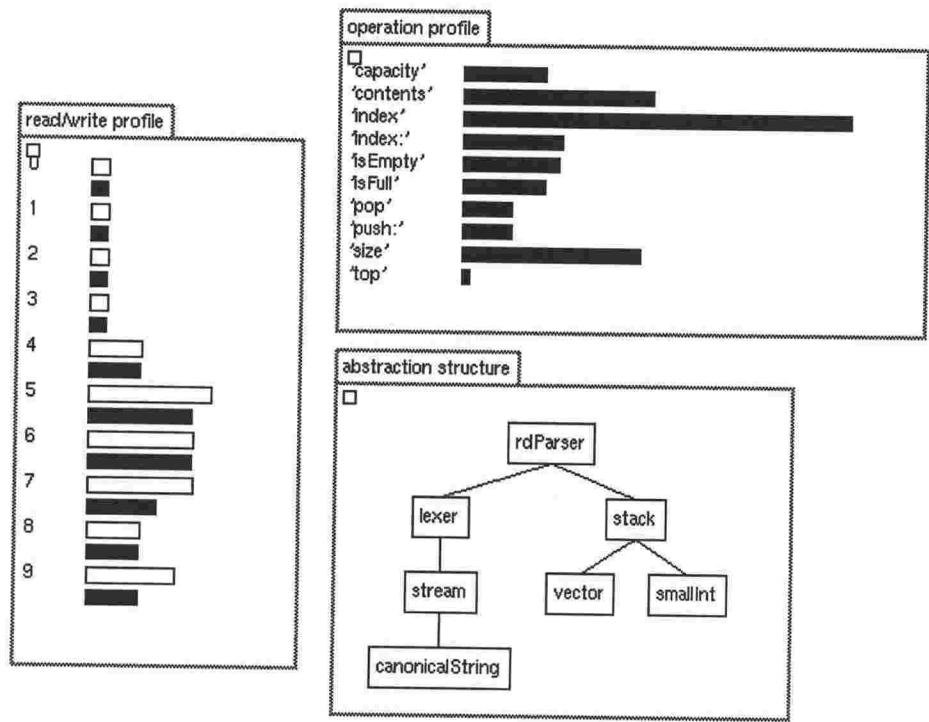
Figure 3.4: Aggregate Abstractions

3.1.4 Summary

This section has discussed three kinds of abstractions used in program visualisation: visual abstractions, program abstractions, and aggregate abstractions.

All program visualisation must by definition involve visual abstraction, and the graphical techniques required are quite well known. Visualisations of aggregate abstractions are similarly well understood, with graphical profilers and performance monitors becoming commonplace (§2.2.1).

Both visual abstractions and aggregate abstractions belong to the context of a visualisation: they are not part of the program itself. Program abstractions represent the design ideas within the program and are independent of any particular visualisation. Thus they are the key to portraying a program in terms of its design in a program exploratorium.

The focus of the remainder this chapter (and indeed the remainder of this thesis) we call *abstract program visualisation*: the explicit use of program abstractions in program visualisation. From this point, where "abstraction" is used unqualified it refers specifically to program abstraction. The following section examines abstract program visualisation techniques in more detail.

3.2 Visualising Abstractions in Programs

All the example illustrations in the previous section can be produced by many of the existing methods of program visualisation. This includes those views especially illustrating program abstraction, such as the views shown in Figure 3.3. This section begins by evaluating this previous work, to determine the extent to which it supports visualisation of program abstractions, drawing upon the detailed survey contained in Chapter 2. It goes on to introduce an improved approach, based upon explicitly recognising program abstractions from the target program's structure.

3.2.1 Previous Approaches

Visual Programming Tools

Visual programming tools such as graphical debuggers (§2.2) typically provide very little support for program abstraction, as they are designed to work at the programming language level. Although more advanced systems such as Incense (§2.2.1) allow user defined visualisations of the program's data structures, there is no differentiation between visual and program abstraction. New views are defined by artist procedures which directly traverse the program's data structures and simultaneously build an illustration. Writing these procedures requires a detailed knowledge of the program's implementation on the part of the visualiser.

Annotation

Annotation-based algorithm animation systems such as Balsa and Anim (§2.3) provide some support for program abstraction by allowing the visualiser to choose the events which are communicated from the program to the visualisation system. Views are defined in terms of these events, rather than by the details of the underlying program.

Event annotations can be chosen to reflect a consistent abstract model of the program, although no such discipline is enforced by these systems. Any program annotated according to the same convention is compatible with such a view, and conversely, any view which understands the event annotations will be compatible with that program. In any case, annotations must be inserted by the visualiser to describe the abstractions within the program, and the visualisation system can only display those abstractions previously identified by annotation.

Direct annotation systems (§2.3.3) provide even less program abstraction support. The annotations in systems like Gestural are defined by explicit reference to the target program's code, and operation parameters are derived directly from the target program's data. By simplifying the connection between program and visualisation, a direct approach makes defining visualisations much easier, but the resulting definitions are very tightly coupled to the details of the target program.

Mapping

Mapping systems use declarative mappings to link the program and the visualisation (§2.4). The mapping is written as a set of rules in a functional or logical style, and describes the visualisation in terms of the target program's text. This is generally simpler than the two previous approaches because the definition is declarative and the target program does not need to be modified.

Some mapping systems (such as Trip §2.4.2) allow several mappings to be composed to define a visualisation. This provides some support for program abstraction, as one mapping may be used to define an abstract model of the program, and another to portray that model. The model represents program abstractions in the same way as the choice of events in an annotation system. As with the other approaches, a program abstraction must be described by the visualiser in terms of its implementation before it can be visualised.

Program Analysis

UWPI (§2.4.3) uses general rules to analyse the target program. This analysis produces a description of the abstractions within the program without user involvement. When the program is executed, these descriptions are used to extract both data and behaviour information to control the visualisation. As far as program abstractions are concerned, UWPI is like an extended mapping system with general rules describing many possible implementations of an abstraction, rather than just the implementation used within a particular target program.

Summary of Previous Approaches

Although they differ in detail, the process to construct an abstract visualisation is roughly the same for each method. The visualiser must inspect the target program and understand how the abstractions to be visualised are implemented. The visualisation then is defined in terms of this implementation. We call this a *bottom up* visualisation.

Several systems (including BALSA (§2.3.1), TRIP (§2.4.2), and UWPI (§2.4.3)) include an intermediate stage which can be used to model program abstractions. This allows the visualiser to explicitly identify information about program abstractions, so that several views of the same abstraction can be constructed without each view's definition requiring the visualiser to reexamine the target program. Any intermediate model is still defined in terms of the details of the abstraction's implementation.

3.2.2 Top Down Visualisation via Explicit Abstractions

Bottom up visualisation assumes that information about the abstractions in a program's design is only *implicitly* present in its implementation. Program abstractions must be embodied somewhere within the program. If a program uses a stack, it must contain an implementation of the stack, that is, data structures to contain the stack's elements, and statements to implement operations to push and pop data, determine if the stack is full, and so on. Unfortunately, the relationship between an abstraction and its implementation is not necessarily obvious. For example, a stack may be implemented by a block of memory and a pointer variable manipulated by several widely separated parts of the target program. The stack's data may not be easily distinguishable from other uses of the underlying memory, and similarly its operations may be spread throughout the code of the whole program.

If, however, the abstractions in a program could be identified *explicitly* from its text, visualisations could be constructed *top down* — working from the definitions of abstractions in the program to their implementations. A program visualisation system could simply inspect the program to determine the abstractions it contained. The information required by views of these abstractions could be acquired by relying upon the explicit definition of the abstractions, rather than by reverse engineering the abstractions' implementations.

Consider the stack described above. If abstractions are represented explicitly in a program's structure, the stack can be detected simply by inspecting the abstractions the program contains. When a view needs information about the stack, the stack's definition can be consulted to determine how that information can be retrieved, rather than the implementation being accessed directly.

Abstractions of both Data and Behaviour

Most PV systems are biased towards visualising either the data or the behaviour of the target program. For example, graphical debuggers typically visualise a program's data (§2.2), as do data-mapping systems such as TRIP and PAVANE (§2.4). In such systems, if the behaviour of the program is to be displayed it must be inferred by comparing changes between successive states. Annotation based systems like BALSA (§2.3) are biased towards displaying the behaviour of a program — large data structures (such as arrays to be sorted) are handled piecemeal by sending a series of events to inform the visualisation systems about each element.

Abstract program visualisation requires information about both code and data. Many useful views of programs show the behaviour of the programs (see Figure 3.2), and views of data benefit from receiving incremental notifications of the changes in the data. Therefore it is important that any explicit model of abstraction used by a PV system encompass both the program's data and code.

3.2.3 Abstractions, Paradigms, and Languages

Explicit representation of abstractions within the target program requires consideration of a model of abstraction within the target program, as well as a model of program visualisation *per se*. A graphical

debugger is designed to display programs written in a particular programming language. Therefore, the model of visualisation it uses must be closely related to the programming model adopted by that language. Algorithm animation systems, on the other hand, are less strongly related to the target program. Systems such as ZEUS or ANIM (§2.3) can visualise many different types of programs, written in different languages, provided the programs can be annotated to produce events. The annotation model of visualisation, which treats the target program solely as a source of interesting events, is unrelated to the programming language model. A model of program visualisation which is based upon explicit representation of abstractions within the target program must be coupled to the model of abstraction used within the program, in the same way the design of a debugger is related to the language being debugged.

In discussing abstraction in programming, we use *paradigm* to mean a conceptual model of abstraction which underlies the design and expression of programs. Different paradigms emphasise different types of abstractions. Take, for example, procedural decomposition, object orientation, and logic programming, which we consider as three separate paradigms. A program written using procedural decomposition is seen as a layered hierarchy of procedure definitions, where each procedure implements a single task, whereas an object oriented program is seen as a collection of communicating objects, and a logic program is seen as a collection of facts and inference rules.

Programming languages are the most visible artifacts of programming paradigms, and often act as exemplars for a paradigm. PASCAL, for example, exemplifies the recursive structure of code and data, SMALLTALK exemplifies object orientation, and PROLOG exemplifies logic programming. A paradigmatic programming language provides constructs which allow the abstractions of a particular paradigm to be written elegantly. An abstract program visualisation system based on a paradigm should be able to recognise these language constructs and use them to identify the abstractions within the program.

The obvious problem which arises with this approach is that the use of a paradigmatic programming language cannot guarantee that a program will be written according to a particular paradigm. For example, consider a visualisation system such as PECAN (§2.2.2) which displays structured flowcharts [62]. If the programming language provides structured control statements (such as case, while and repeat) this task is quite simple: the target program's control flow graph can be derived easily from its abstract syntax, because the language's control statements explicitly represent the program's control flow. If the language only provides goto statements, a much more extensive analysis is required to find the program's flow graph [3].

Even in a language with structured control statements, a program's control flow can only be determined from its abstract syntax if the program is actually written in a structured style. An unstructured program can be written in a structured language, using structured statements to emulate gotos by using a loop around a case statement. In this case, an abstract syntax based analysis would appear to work, but its results would be misleading. The analysis would reveal only the loop and the case statement, that is to say, it would show that clauses of the case statement could be executed in any order.

If, therefore, a program is written in good style, and the constructs provided by a paradigmatic programming language are used to express the abstractions important in the program's design, a program visualisation system should be able to use the programming language's constructs to recover the target program's abstractions. If a suitable paradigm to express program abstractions can be identified, a visualisation system should be able to display a program's abstractions top down, without the use of bottom up techniques such as annotations or mapping rules.

It follows from the above discussion that the choice of paradigm is very important. The paradigm determines what kinds of abstractions can be represented explicitly in the target program, and thus what can be identified easily by the visualisation system. For this reason, the following four sections are devoted to the evaluation of four paradigms with respect to their support for abstract program visualisation. The paradigms we have chosen to evaluate are procedural decomposition (§3.3), structured data types (§3.4), abstract data types (§3.5), and object orientation (§3.6). These paradigms are well known to computer science. All are mentioned in Dijkstra, Hoare and Dahl's *Structured Programming*[1] [56] first published

---

[1] *Structured Programming* contains three separate essays. The first, Dijkstra's *Notes on Structured Programming* [62] describes procedural decomposition. The second essay, Hoare's *Notes on Data Structuring* [99], describes both structured

in 1970, and thus are at least twenty-five years old. As will be seen from the discussion, we favour the object oriented paradigm.

The descriptions of the paradigms and the examples are not original but draw in particular upon the more comprehensive surveys found in Collberg's thesis [51] and Stroustrup's wonderfully-named *Sixteen Ways to Stack a Cat* [211].

3.3   Procedural Decomposition

Procedural decomposition is one of the oldest established programming paradigms. Its evolution is usually traced to Dijkstra, who, quoting the doctrine of *Divide and Rule* [62], realised that procedures could be used to organise abstractions within programs. Following this approach, a program is designed by breaking the original problem into several smaller subproblems which can be solved independently. This subdivision is captured by writing a procedure which calls subprocedures for each subproblem. The decomposition terminates when each subproblem is sufficiently small that it can be implemented by writing a single procedure. The process of repeated subdivision is known as *stepwise refinement* [228], as each subdivision incrementally refines a high-level abstract specification towards a concrete and executable program.

Procedural decomposition captures procedural abstractions: a procedure is used to implement a particular task. The requirements of the task form a specification for the procedure. Provided a procedure meets its specifications (is implemented correctly) its implementation details are unimportant as far as users of the procedure are concerned. As described by Dijkstra, a procedure at a given level of the decomposition can be considered as a program for a virtual machine — its subprocedures being the virtual machine's instructions. In the same way that a program written in a high-level language can be understood without knowledge of the machines upon which it may run, a procedure can be understood (or visualised) regardless of the implementations of the subprocedures, if their specifications are understood.

3.3.1   A Procedural Program

Figure 3.5 presents a simple example of a procedural program written in an idealised Pascal-like language, and structured using procedural decomposition alone. The example program reproduces its input with lines reversed, like the unix filter rev, then prints the number of lines read. The main task is implemented by the reverse procedure. It is decomposed into three tasks: initialising the data structures, performed by the initialise procedure; processing each input line, performed by the handle_line procedure; and printing the number of lines read, performed by a single write statement in reverse.

The handle_line procedure counts and reverses each line. To perform the reversal, it accumulates characters into the contents array using the index variable index. When an entire line has been processed, characters are read out from the array in reverse order. The subtasks of moving characters in to and out of the array are implemented by the charin and charout procedures — note that bounds checking has been omitted to keep the example to a manageable size. The variable lines accumulates the number of lines read.

This program captures the structure of the task quite well: each procedure corresponds to a subtask in the decomposition.

3.3.2   Visualising Procedural Programs

Procedural programs can be visualised in various ways. Information about procedures can be gathered by analysing the static structure of the program, or by monitoring procedure calls as they occur (§2.5). Programming environments may display call graphs showing the relationships between the procedures comprising a program, or graphical profiles which show the amount of a program's execution time spent

---

and abstract data types. Finally, Dahl and Hoare's *Hierarchical Program Structures* [57] describes the essence of object oriented programming.

```
var
 contents: array[80] of char;
 index: integer;
 lines: integer;

procedure charin (c: char);
 begin contents[index] := c; index := index + 1; end;

procedure charout returns char;
 begin index := index - 1; return contents[index]; end;

procedure initialise;
 begin index := 0; lines := 0; end;

procedure handle_line;
 begin
  while (not eoln) do charin(read) od;
  while (index > 0) do write(charout) od;
  lines := lines + 1;
 end;

procedure reverse;
 begin
  initialise;
  while (not eof) do handle_line od;
  write('Reversed: ',lines,' lines\n');
 end;
```

Figure 3.5: A Procedural Program using a Stack

in a particular procedure. The trace view from Figure 3.2 and the operation profile view from Figure 3.4 are examples of views that can be constructed easily by monitoring procedure calls. Unfortunately most views of procedural programs display aggregate abstractions rather than program abstractions: illustrations of program abstractions based solely upon procedural decomposition seem very hard to find.

In visualising programs, the data a program manipulates is at least as important as the functions a program performs (§3.2.2). Classical procedural decomposition doesn't concern itself with data, so data structures will be invisible to a program visualisation system based upon procedural decomposition. For example, in Figure 3.5, there is nothing in the code to indicate that the contents array and the index variable can be understood together as part of the implementation of a stack, while the counter variable lines is unrelated to either.

3.3.3 Summary

Procedural decomposition models the tasks a program performs. Although task-based information is sufficient to produce useful aggregate views, because this paradigm gives little attention to the data a program manipulates, it is not a good candidate to capture abstractions for a program exploratorium.

3.4 Structured Data Types

A strict procedural decomposition does not provide a satisfactory foundation for program visualisation, principally because it does not take account of the data manipulated by the program. This is a general

problem with the paradigm, so in the 1960's attempts were made to find methods of structuring a program's data as well as its code. These resulted in the idea of *Structured Data Types (SDTs)*, first developed in languages such as PASCAL.

A structured data type builds a new data representation from a set of primitive types and a set of type constructors [99]. Primitive (or unstructured) types represent atomic values taken from a particular domain, such as integers, floating point numbers, or characters. Structured types, constructed from arrays, records, sets, and unions, are produced from the atomic types using type constructors. The structure of these types is recursive, as type constructors may be applied to other constructed types as well as to primitive types.

3.4.1 Structured Data in Programs

Figure 3.6 presents a version of the example from Figure 3.5 using an explicit stack implemented with a structured data type, also using an idealised PASCAL-like language. The stack is embodied in the stack named type, which is constructed as a record containing the contents array and the index variable. The charin and charout procedures from Figure 3.5 now become push and pop procedures which operate upon the stack, and the stack is initialised via the initialise_stack procedure. A global variable s embodies the actual stack manipulated by the program. The lines variable is not part of the stack's implementation, so it is not part of the structured type.

The push, pop, and initialise_stack procedures have an argument (stk of type stack) which represents the stack to be operated upon, so these procedures can be seen to implement stack operations. This is not enforced by the language, and the program can also access the stack directly. This is illustrated by the handle_line procedure in Figure 3.6, which simply inspects the data structure to check whether the stack is empty.

3.4.2 Visualising Structured Data

Structured data types have been visualised successfully in several graphical debuggers, the earliest probably being INCENSE (§2.2.1). A structured type is generally displayed by decomposing the type according to its type constructors. Primitive types can be drawn either with text strings, or with simple graphical representations similar to user interface widgets, and constructed types are illustrated by combining displays of their elements. The structure of the view parallels the structure of the data type being displayed. To keep views up-to-date as the program runs, its data must be monitored and the views redrawn whenever the data changes.

This is illustrated in the stack implementation view of Figure 3.3. The stack record contains two fields, contents and index, displayed in separate subviews in the stack implementation view. The contents implementation view similarly displays each component of the contents array implementation.

Unfortunately, decomposition of structured data types does not suffice to produce displays of program abstractions, such as the stack abstraction view from Figure 3.3. To create such a display, the visualiser must know how the representation is used to build a stack (whether the index variable denotes the last stack element or first empty space, and the direction in which the stack grows in the array), and how modifications of this representation affect the abstraction. This information is distributed among the procedures manipulating the abstraction: it is not explicit in the program, and is certainly not contained in the definition of the structured type.

3.4.3 Summary

For abstract program visualisation, using structured data types remedies the major problem of procedural decomposition: structured types can be used to group and identify the target program's data structures. Unfortunately, as a structured data type is only a well organised concrete implementation, it does not provide much assistance for visualising the program abstraction it implements.

```
type
  stack = record
    contents: array[80] of char;
    index: integer;
  end;

var
  s: stack;
  lines: integer;

procedure initialise_stack (var stk: stack);
 begin
  stk.index := 0;
 end;

procedure push (var stk: stack, c: char);
 begin
  stk.contents[stk.index] := c;
  stk.index := stk.index + 1;
 end;

procedure pop (var stk: stack) returns char;
 begin
  stk.index := stk.index − 1;
  return stk.contents[stk.index];
 end;

procedure initialise;
 begin initialise_stack(s); lines := 0; end;

procedure handle_line;
 begin
  while (not eoln) do push(s,read) od;
  while (s.index > 0) do write(pop(s)) od;
  lines := lines + 1;
 end;

procedure reverse;
 begin
  initialise;
  while (not eof) do handle_line od;
  write('Reversed: ',lines,' lines\n');
 end;
```

Figure 3.6: A Program using a Stack Data Structure

3.5 Abstract Data Types

Abstract Data Types (ADTs) [93, 99, 100] were proposed in the 1970's as a technique for building truly abstract data structures in programs. ADTs use procedures to provide abstract interfaces to data, thus giving data the benefits of procedures' abstraction.

There are two mutually reinforcing perspectives on ADTs [125]: from a procedural perspective, an

ADT is a type, and a group of procedures sharing common data of that type; from a data type perspective, an ADT is a set of values taken from some domain and a set of operations upon those values. The most important feature of an ADT is expressed in both definitions: an ADT organises both procedure and data structure. The presentation of ADTs in this thesis draws heavily upon the work of Liskov and Guttag [132] and Meyer [140].

An ADT has an *interface* which is a list of operations that can be performed upon its instances. An implemented ADT must also have an internal description of how that ADT is implemented in terms of lower level concepts. An ADT implementation consists of a *representation* and a set of procedures. The representation is a structured type containing the information required by each instance of the ADT, and the procedures perform the operations defined in the ADT's interface upon the representation.

The most important feature of paradigmatic ADT style is the separation of ADT's interfaces and implementations. This is known as *information hiding* (also *privacy* or *encapsulation*). The implementation of an ADT should only be available within its definition: in the rest of the program, the ADT should only be manipulable via operations in its interface.

Supporting information hiding is the focus of modular programming languages such as MODULA-2 [230] and ADA [6]. These languages divide programs into *modules* (called *packages* in ADA) which may be compiled separately and which provide local namespaces. To allow one module (a *client*) to use an ADT defined in another module (a *supplier*), the supplier module declares that some of its local definitions are available for *export*; the client then *imports* those definitions. Any declarations which are not exported are hidden inside that module. An ADT is implemented by a module which exports procedures corresponding to the ADT's operations, and hides the representation type.

An informal specification of the interface of a stack ADT is shown in Table 3.1. Each operation supported by the ADT is listed, and the operation's behaviour described.

Stack Interface

constructors	
new_stack	Create a new stack
mutators	
push(stack, e)	Push element e onto the stack.
pop(stack)	Remove the top element from the stack.
accessors	
top(stack)	Return the top element on the stack.
size(stack)	Return the number of elements in the stack.
isEmpty(stack)	Return true if the stack is empty.

Table 3.1: Stack ADT interface

ADT operations can be grouped into several categories [132, 140]. Operations which create an ADT instance (new_stack in the table) are known as *constructors*; operations which change the value of an ADT instance (push and pop) are known as *mutators*; and operations which retrieve information from an instance but do not change its value (top, size, and isEmpty) are known as *accessors*.

A program using ADTs may be designed using stepwise refinement, as discussed in Section 3.3. The overall design strategy remains divide and rule, but ADTs are used as the main design element rather than procedures. The decomposition may be focused upon data, rather than function, but often both data and function are considered simultaneously. The resulting program is structured as a hierarchy of ADTs. Rather than implementing all ADTs directly in the programming lang

3.5.1 An ADT Program

Figures 3.7 and 3.8 revisit the stack example, this time using ADTs. These figures use an idealised MODULA-2-like language, with a syntax as similar as possible to the previous examples in this chapter. The most important feature of this example is that the program is now in two separate modules: the definition of the ADT (Figure 3.7) and its use (Figure 3.8). The stack definition is similar to the SDT version in Figure 3.6 — a record type, named stackrep in Figure 3.7, holds the stack's contents and index. The stackrep type is not exported outside the ADT definition module; rather an opaque type named stack is exported. The stack module's clients can use variables of type stack to refer to stacks, but cannot manipulate them directly because a stack's full definition (the stackrep type) is visible only within the stack module.

```
module stack;
  export new_stack, push, pop, isEmpty, stack;

    type
       stack = stackrep;

    stackrep = record
        contents: array[80] of char;
        index: integer;
      end;

    procedure new_stack returns stack;
     var stk: stack;
     begin
      stk := new(stackrep);
      stk.index := 0;
      return stk;
     end;

    procedure push (stk: stack, c: char);
     begin
      stk.contents[stk.index] := c;
      stk.index := stk.index + 1;
     end;

    procedure pop (stk: stack) returns char;
     begin
      stk.index := stk.index − 1;
      return stk.contents[stk.index];
     end;

    procedure isEmpty(stk: stack) returns boolean;
      begin return (stk.index = 0); end;

  end;
```

Figure 3.7: A Definition of a Stack ADT

The main program (Figure 3.8) is also a separate module. It imports the stack module, but is otherwise similar to the previous versions.

The ADT exports the constructor procedure new_stack, the mutator procedures push and pop, and the

```
module main;
  import stack;
  export reverse;

    var
        s: stack;
        lines: integer;

    procedure initialise;
      begin s := new_stack; lines := 0; end;

    procedure handle_line;
      begin
        while (not eoln) do push(s, read) od;
        while (not isEmpty(s)) do write(pop(s)) od;
        lines := lines + 1;
      end;

    procedure reverse;
      begin
        initialise;
        while (not eof) do handle_line od;
        write('Reversed: ',lines,' lines\n');
      end;

end;
```

Figure 3.8: A Program using a Stack ADT

accessor function isEmpty. This accessor is required because the handle_lines procedure has to determine whether the stack is empty. Previous versions of the procedure examined the stack's representation directly, but this is no longer possible because the data structure is encapsulated within the ADT's definition.

3.5.2 Visualising ADT Programs

Implementation data can be retrieved directly from SDTs (§3.4.2). By analogy, abstract data can be retrieved from ADT instances by calling accessor operations which return that information. Because the data returned is abstract and independent of a particular implementation, the PV system does not have to reinterpret any implementation data structures. This requires that the PV system is able to call procedures defined in the target program when information is required, rather than simply inspect its memory.

The behaviour of ADTs can be monitored by a similar analogy; rather than indiscriminately monitoring procedures, we monitor those procedures implementing the ADT's operations. Because ADT's implementations are hidden behind their interfaces, all operations upon abstractions must be performed through their interfaces. By monitoring the procedures exported from an ADT, we are assured of monitoring all the computation performed in the context of that abstraction.

Since an ADT's representation is essentially a SDT and a collection of procedures, implementation views of an ADT can be produced by inspecting the state of the SDT (§3.4.2) and monitoring the procedures (§3.2).

3.5.3 Summary

ADTs contain both data and code, and so avoid the largest disadvantages of the previous approaches; that procedural decomposition only describes control flow, and that SDTs only organise data. Unlike structured data types, ADTs are abstract: they hide their implementations behind their interfaces, and therefore ADTs can be visualised top down, by working from their interfaces to their implementations. Abstract views of ADTs can retrieve the information they need by calling accessor operations in the ADT's interface, and can detect changes in instances by monitoring operations applied to that instance.

The encapsulation of abstract data types, observed by the programmer and supported by the language, ensures that ADT instances can be considered in isolation. In particular, all operations upon an abstraction can be detected by monitoring the interface of the instance representing that abstraction.

Abstract data types surpass both procedural decomposition and structured data types for reifying abstractions within programs. ADTs should be sufficient to explicitly represent all the types of program abstractions shown in Section 3.1 in a target program.

3.6 Object Orientation

The term *object oriented* (abbreviated *OO*) was coined by Alan Kay at the Xerox PARC Learning Research Group as part of the design effort that produced the SMALLTALK language and programming environment [111]. This and several related neologisms were introduced to signify the paradigm developed by the group, in much the same way as the relational database community replaced traditional terms such as *file*, *record*, and *field* with *table*, *row*, and *column* [58].

The most important of these new terms is of course *object*, used in a specialised sense to mean a runtime program component — a dynamically allocated memory record containing data and procedures. Others include *class* for an object type definition (similar to a structured record declaration), *method* for function or procedure definition, and *message* for procedure call, from which comes *message sending* for procedure calling.

The object oriented paradigm is an extension of the ADT paradigm. A class can be considered an ADT implementation, and objects which are instances of that class correspond to instances of the ADT [140]. In this thesis we prefer the object oriented terms when our discussion is within that paradigm, and the more traditional terms otherwise.

Identity and State

Objects have *identity* and *state*. An object's identity is a unique distinguishing mark assigned to each object when it is created, and an object's state is a set of variables. Two objects otherwise containing the same state can be differentiated by comparing their identities. If objects are implemented simply as dynamically allocated memory records, an object's identity could be the address of the record holding its state.

The variables comprising an object's state often include references to other objects. The topology of an OO program is then a dynamic graph (the *object graph*), with nodes representing objects and edges references between objects. Each node is labelled with the unique identifier given to its object when it was created.

Message Sending

Message sending is the OO analogue of procedure calling. Methods (procedure definitions) are attached to particular objects. Like a procedure call, a message send may take arguments, but unlike procedures, all messages have one distinguished argument, the *receiver*. This is the object to which the message is sent, and where a method implementing the message, if any, will be found. The receiver argument is usually implicit in method definitions in OO languages, and methods execute in a scope containing the receiver and its local state.

Responding to a message is a two stage process. The receiver is searched for a method with the same name (*selector*) as the message; such a method implements the message. If a suitable method is found, it is executed. The particular method chosen depends upon the dynamic type of the method receiver, so message sending is known as *dynamic dispatch*. This is in contrast to the overloading (a.k.a. *static dispatch*) common to PL/I, ADA and other languages, where the actual function called depends upon the *static* types of the supplied arguments.

The set of messages implemented by an object is known as the object's *protocol*, and the object is said to *understand* these messages. Depending upon the type discipline adopted, some messages may not be successfully dispatched — that is, a message may be sent to an object which does not understand it. This causes a run-time error, known as an *undefined selector* exception, which is handled similarly to an array bounds exception in structured languages.

Encapsulation

An object's local state and behaviour (local method definitions) are *private*: they are only accessible from methods scoped within the object. This is similar to the information hiding provided by modules, and ensures a barrier is maintained between the implementation of an object and its use.

SMALLTALK enforces encapsulation on a per-object basis, that is, a method may access only its receiver's private components. The encapsulation provided by other languages is significantly weaker, granting access to all instances of an object's class (e.g., CLU [132]), or instances of both the object's class and other nominated classes (EIFFEL [141] and C++ [210]).

Inheritance

Object oriented programming includes inheritance. Inheritance allows new types of objects to be defined as extensions to the definitions of existing objects, by adding new local state and adding or replacing method definitions. The preexisting class is called the *superclass* and the new class the *subclass*. When an object of a subclass receives a message, both the subclass and the superclass are searched for matching messages, with the subclass taking precedence. In this way code can be reused easily, as common definitions can be placed in a superclass, and then inherited by more specialised subclasses. More importantly, inheritance can be used to ensure that several subclasses share the same interface (the superclass' protocol should be common to all subclasses).

3.6.1 An Object Oriented Program

An object oriented version of the stack example is presented in Figures 3.9 and 3.10. These figures are written in an idealised OO language with a general style similar to the earlier examples. This is similar in many respects to the ADT example presented above (§3.5.1) — the basic data structure is unchanged, and the program is again in two parts, separating the definition of the stack from its use. The stack itself is an object, with public push, pop and isEmpty methods, and a private contents variable and index array. The main difference is that there are no types defined explicitly, as the stack class is both a type and a module. The methods do not manipulate an explicit stack argument, rather, the receiver is passed implicitly. No extra stackrep type is required, as the class construct itself encapsulates the contents and index variables. The stack class in Figure 3.9 performs the functions of both the stack type and the stackrep type from Figure 3.7.

The differences between the ADT main program (Figure 3.8) and the OO main program (Figure 3.10) are more subtle. The main program is now an object, rather than a module. A variable s containing a stack is declared and all operations upon the stack are performed by sending messages to this variable.

3.6.2 Visualising Object Oriented Programs

Most of the benefits of ADTs for visualisation apply equally to objects. In particular, objects, like ADTs:

```
class stack;

private
  contents: array[80] of char;
  index: integer;

public
  method new returns stack;
   begin
    index := 0;
    return self;
   end;

  method push (c: char);
   begin
    contents[index] := c;
    index := index + 1;
   end;

  method pop returns char;
   begin
    index := index - 1;
    return contents[index];
   end;

  method isEmpty returns boolean;
   begin return (index < 0); end;

end;
```

Figure 3.9: A Definition of a Stack object

- organise both code and data.

- are abstract, in that they separate their interface and implementation.

- are easy to identify in the program.

Objects can therefore be visualised in much the same way as ADTs, by working top down from their interfaces to their implementations (§3.5.2). The messages received by an object can be monitored to detect the operations performed upon the abstraction represented by that object, and messages can be sent to objects to retrieve their abstract state. Implementation views can be built by illustrating an object's local state and monitoring its methods.

ADTs are quite adequate for representing abstractions in programs, however, several technical features of OO languages provide advantages over their modular counterparts for program visualisation:

- Classes combine modules and types.

- Objects uniformly organise the entire program.

- Inheritance supports common interfaces between objects.

We discuss each of these points in turn.

```
class reverser;

 private
   s: stack;
   lines: integer;

   method initialise;
    begin s := new stack; lines := 0; end;

   method handle_line;
    begin
     while (not eoln) do s.push(read) od;
     while (not s.isEmpty) do write(s.pop) od;
     lines := lines + 1;
    end;

 public
   method reverse;
    begin
     initialise;
     while (not eof) do handle_line od;
     write('Reversed: ',lines,' lines\n');
    end;
end;
```

Figure 3.10: A Program using a Stack object

Classes, Modules, and Types

Object classes define types, and simultaneously encapsulate those types. Modules, in contrast, provide encapsulation control only [231], they can contain types and procedures to implement ADTs, but also have other uses. Booch [25] groups ADA modules into four categories.

1. Groups of related structured type and constant definitions.

2. Groups of related procedure definitions.

3. Abstract Data Types.

4. Abstract State Machines.

An example of the first category is a module defining the values of configuration parameters for a compiler implementation, and of the second, a module containing a library of mathematical functions [51]. These two categories illustrate one important use of modules: to organise software into components to solve problems such as code library management and separate compilation.

The third and fourth categories use modules to represent abstractions. An Abstract State Machine module is roughly equivalent to a single ADT instance. It exports procedures which operate directly upon the data hidden within the module; there is no explicit type. If multiple instances are required the program must contain multiple modules. The main program module in Figure 3.8 could be considered an Abstract State Machine. The stack ADT module from Figure 3.7 unsurprisingly belongs to Booch's ADT category.

Note that modular languages in practice impose no restrictions upon the relationships between modules and types. In particular, a module could well be used to implement two or more of Booch's categories.

A module could group procedures and structured data types together with an abstract state machine, or could contain several ADT definitions. A PV system visualising a modular program would therefore have to be sensitive to the various types of modules, especially the difference between ADTs and Abstract State Machines — one with an explicit type and one without — and to the possibility that modules may in practice combine several categories. A class, in contrast, associates each module with a single type, and so roughly corresponds to a modular program in which every module implements an ADT.

Uniformity

A pure object oriented language (such as SMALLTALK [85] or EIFFEL [141]) uses objects uniformly to organise programs; even the language's basic data types (such as integers and Booleans) are provided by objects. Objects can also be used to represent algorithms and procedures, as well as data structures [79].

For example, a compiler front end may be made up of a lexical analyser, a parser, and a symbol table. The lexical analyser may itself be implemented using an input stream, and the stack by an array and an integer index variable (see Figure 3.4). The symbol table, stack, array, and integer variable are essentially data structures, while the lexer and parser are essentially procedures. All of these (including the lexer and parser) can be represented using objects, whereas only the symbol table, stack, and possibly the array would be represented as ADTs in a modular language.

Inheritance

The use of inheritance supports a common vocabulary of operations across different types of objects. In modular languages, each abstraction is typically designed and implemented independently. Although there is no reason why similar ADTs should not be designed in similar ways and have similar interfaces, there is no support for this within most modular languages. Since inheritance is an important part of object oriented programming, similar objects are very likely to have similar interfaces.

Since an abstract view of an object depends only upon that object's interface, an abstract view should be able to display any object with a suitable interface.

3.6.3 Summary

Object orientation is essentially an extension of the ADT paradigm: a class implements an ADT, with objects as the type's instances. Like ADTs, object orientation should provide a good vehicle for representing abstractions for program visualisation.

In comparison with modular languages, object oriented languages use a single language construct (the object) to represent program abstractions. Object oriented scope and encapsulation rules promote the independence of objects, and inheritance and message passing ensure that common names may be used to denote similar operations while distancing the use of those operations from their implementations.

For these reasons, an object oriented language should provide more support for abstract program visualisation than a modular language, provided that the target program is written in a suitable paradigmatic style.

3.7 A Model of Abstract Program Visualisation

We have developed the *Abstract* Program Mapping Visualisation model (APMV) to describe a design for an abstract program visualisation system which displays the program abstractions present in a well-written object oriented program. The APMV model is based upon the PMV model, and it consists of the same three components, however it differs from that model in two important respects. First, each component of the model is no longer monolithic. The program is viewed as a collection of abstractions represented by objects. Each abstraction can be visualised independently, using independent mappings (which we call *strategies*) and displays (*views*). Second, the components are connected bidirectionally. In

the PMV model, the program component reports the program's actions to the mapping component which forwards these as changes to the visualisation component. In the APMV model, the program's actions are monitored and changes generated, but views can also send messages to objects in the target program (via the mapping component's strategies) to recover abstract information directly. These message sends are known as *callbacks*, as the visualisation system "calls back" into the target program.

The APMV model is illustrated in Figure 3.11 (compare with the PMV model, Figure 2.1).
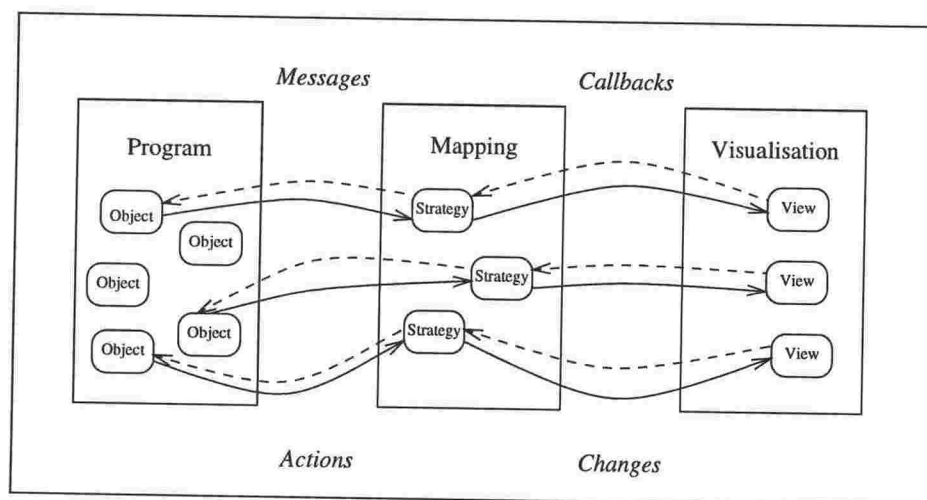
Figure 3.11: The APMV Model

3.7.1 Program Component

The program component contains the target program, and connects it to the mapping component. Most importantly, the program component monitors the actions of objects in the target program, as directed by the mapping component's strategies. The actions that can be detected depend upon the target programming language, and may include the receipt and return of messages, and the reading and writing of variables.

The program component also gathers structural information about the objects in the target program, and provides this information to the mapping and visualisation components. The mapping component uses this information to decide how best to monitor the objects, and the visualisation component uses this information to allow the user to choose which objects to display and to present language level views.

3.7.2 Mapping Component

The mapping component mediates between the program and visualisation components. Each view in the visualisation component is linked to an object in the program via a particular mapping. We call these mappings *strategies* because their main task is to determine how an object should be monitored. Strategies can also adapt the view's callbacks to suit the target object, or adapt the object's changes to suit the target object.

Views and strategies are complementary. A view's purpose is to display an image and handle user interaction. A strategy supplies information to a view, using actions reported by the program component to send changes to the view, and routeing the view's callbacks to the target object. This separation of concerns between views and strategies factors out common behaviour and allows both views and strategies to be reused.

Strategies are written by the visualiser in a general purpose programming language, and then associated with views. When a view is created, its strategy inspects the target object, and requests that the program component monitor the target object's actions.

Consider the dots and sticks views of sequences from Figure 3.1. Both views could be used to display an array (a type of sequence) using a suitable strategy, such as monitoring low-level array operations. A linked list is another type of sequence, and should be able to be portrayed by sequence views, but a linked list must be monitored differently from an array. The dots and sticks views can be used unchanged to display a linked list, provided the array strategy is replaced by another more suited to linked lists, such as monitoring high-level sequence operations.

Strategies receive notifications of actions from the program component. The actions may be processed by the strategy, passed unmodified to the target program, or ignored. A strategy may even send messages to objects in the target program before forwarding changes to the view. Callbacks are received from the view, and may be processed similarly: sent directly to the target object, delayed, or ignored (although the strategy must at least manufacture a value to be returned to the view). Most simple strategies relay the actions detected by the program component directly to their views, and forward their view's callbacks directly to the target object. More complex processing is used principally to provide aggregate abstractions, or adapt the view to the target object.

Strategies can be used to support views of aggregate abstractions. For example, the operation profile from Figure 3.4 can be implemented by a strategy which monitors the target object and updates a profile database object to record the target object's actions. The strategy sends its view changes referring to the profile database, rather than the target object, and redirects its view's callbacks to the profile database also. The view is effectively attached to the database profile object, not the actual object in the target program. The view is unaware that it is displaying an aggregate abstraction (the profile) rather than a program abstraction (the target object). If the database is stored as a sequence, it can be displayed by any sequence view.

A view must be able to understand the changes it receives from its target object, and similarly it must only send callbacks the target object understands. This is similar to type compatibility in programming languages, and is based upon the target object's interface. Strategies can be used to ensure that views are compatible with their target objects, by translating the target object's changes so that they are understood by the view and translating the view's callbacks so that they are understood by the target object.

3.7.3 Visualisation Component

Views are the subcomponents of the visualisation component of the APMV model. Each view corresponds to an individual visual abstraction, typically appearing in a single window on a bitmapped display, and displays a particular object in the target program, known as that view's *target object*. An object can be displayed in several different views, giving multiple illustrations of the same object. Views also provide the user interface for the visualisation system.

Views are written by the visualiser using a graphics system and programming language. The visualiser also specifies a strategy to be used by the view. When a user requests a view of a target object, a new view is created from the visualiser's definition, and this in turn causes a new strategy object to be instantiated for that view.

A view communicates with its target object indirectly, with its strategy as an intermediary. A view receives changes from its strategy, and these describe the actions of the target object of interest to the view. A view may respond to changes in several ways: *static* views display static pictures of their target object and simply ignore any changes (they typically employ a strategy which does not generate any changes); *batch* views use callbacks to retrieve the target object's current state, and then update their display; and *incremental* views use information from the changes to update their display.

A view can also send callbacks to its target object via its strategy. A callback is a message (i.e. a message selector and one or more arguments) which the strategy sends to its target object. The message is executed by the target object, and the result returned to the view. Views may send callbacks for several purposes: to gather initial information about a view's target object as soon as the view is created; to gather information in response to changes the view receives; and to perform operations upon the target object in response to user input.

3.7.4   Callbacks and Changes

Callbacks and changes form a complementary pair. Callbacks are initiated by views, while changes result from actions within the target program. Callbacks may be dispatched at any time with respect to the target program (we say they are *asynchronous*), while the reverse is true of changes, which are generated at precise points in the target program's execution (they are *synchronous*). Callbacks are typically sent from a view via a strategy to a single target object, while each target object many send changes to many strategies and views.

The following table summarises this relationship.

Callbacks	Changes
Originate from view	Originate from program
Asynchronous	Synchronous
Many to one	One to many

3.8   Summary

This chapter has presented three important ideas: that program abstractions are important in program visualisation; that explicit representation of program abstractions is very useful in constructing visualisations; and that object orientation should be able to provide this representation.

We began by identifying three separate kinds of abstraction used in program visualisation, two of which (visual abstraction and aggregate abstraction) concern only the presentation of the program and are well understood. The third, program abstraction, represents the design ideas within the program, and is crucial for understanding any moderately complex program.

Existing program visualisation techniques do not support program abstraction well, since they rely on the visualiser to provide information about the abstractions within the target program. We proposed a novel approach, based around the explicit representation of abstractions within a program. This approach in principle allows visualisations to be constructed top down, working from identifiable definitions of abstractions contained within the program, rather than their implementations.

We reviewed several models of abstraction in programming to determine their suitability for visualising program abstractions, and determined that those paradigms based upon abstraction of both code and data provide the best support. In particular, objects contain both code and data while maintaining a strong encapsulation barrier between their interface and implementation.

We then described the APMV model, a novel model of abstract program visualisation derived from the PMV model. The APMV model visualises the program abstractions within an object oriented target program.

This scheme can be summarised in the following principle:

> The *pictures* we draw correspond to the *abstractions* in the *design,* which are the *objects* in the program.

The various aspects of the APMV model will be discussed in detail in the subsequent chapters of this thesis. Chapter 4 examines the rôles of strategies within the model, and the following chapters describe the Tarraingím program exploratorium prototype, which we have built as a proof-of-concept of the model.

38. Structured Programming supports the law of the excluded muddle.

Alan Perlis, *Epigrams On Programming* [168]

4

Abstract Program Visualisation

The APMV model visualises the target program by using objects to represent program abstractions. This chapter investigates several pragmatic considerations in the realisation of that model. In particular, it focuses on the information required from an object in the target program to produce an abstract view, and the two mechanisms used to retrieve this information: callbacks and changes.

An abstract program exploratorium should be able to produce views at many levels of abstraction: showing an object in the target program as an abstraction, showing how that abstraction is implemented in terms of other abstractions, and showing its realisation in the programming language. This chapter is phrased in terms of producing an abstract view of an object. This is because the recovery of abstractions from the target program is the key to the APMV model. Implementation views, or views of objects at other levels of abstraction, can be drawn by treating each of the object's components as separate abstractions, and combining this information to create the view.

The first section (§4.1) discusses the representation of program abstractions in the structure of the target program. The second and third sections discuss the use of callbacks (§4.2) and changes (§4.3) to retrieve the abstract information from a running program. These two sections discuss issues the visualiser must consider when writing strategies, and depend upon the assumption that the target program does not involve object aliasing. Section 4.4 discusses this assumption, and describes how it may be lifted.

4.1   The Design of the Target Program

The APMV model determines a program's design abstractions from its structure. This section discusses the relationship between this approach and the detailed design of the target program.

This section begins by considering the design of the objects (§4.1.1) and operations (§4.1.2) within the target program. Section 4.1.3 investigates visualising information related to a program's correctness, and Section 4.1.4 describes how abstract information is retrieved by callbacks.

4.1.1   Modelling Abstractions

The objects represented by the target program's structure should represent the abstractions the programmer considered important. These abstractions are not necessarily those the user wishes to see. For example, the user may wish to visualise the lexical analysis performed by a compiler. The design of the target compiler may distribute the lexical analysis throughout various parts of the compiler, and may not

incorporate a separate lexer to visualise. In general, a program can be designed in more than one way. The programmer may have chosen one design (e.g., a parser without a separate lexer), while the user of the animation system may assume another.

The advantage of using the target program's structure to capture program abstractions is that information about these abstractions can be obtained easily. This advantage applies only to those abstractions explicit in the program's design (§3.2.2). Program visualisation implies a relationship between the target program and the resulting illustrations. The user of a program exploratorium is investigating a real, concrete, implemented program.

Views which present an idealised picture of the target program, or illustrate alternative designs are also very useful, especially if they can be displayed alongside views illustrating the program as it actually is. Such views can be provided within the APMV model, by using strategies to provide aggregate abstractions. Since strategies are written using the full power of a programming language, almost any aggregate abstraction can be constructed, given sufficient effort on the part of the visualiser. The closer the aggregate abstractions are to the program abstractions represented within the target program, the easier the strategies to model them should be to write.

4.1.2 Modelling Operations

The APMV model relies upon the program abstractions the user wishes to see being represented as objects within the target program, or being synthesised as aggregate abstractions. The important operations upon those abstractions must similarly be represented as messages sent between corresponding objects.

Consider visualising Quicksort[1]. Figure 4.1 shows Quicksort implemented with a single procedure acting on an array, using array element accessor and assignment operations to sort the array. The APMV program component can monitor the program's actions in calling these operations, plus the recursive invocations of the Quicksort procedure.

```
procedure Quicksort (l, r: integer);
var ...;
begin
  v := a[r]; i := l − 1; j := r;
  repeat
    repeat i := i + 1 until a[i] ≥ v;
    repeat j := j − 1 until a[j] ≤ v;
    t := a[i]; a[i] := a[j]; a[j] := t;
  until j ≤ i;
  a[j] := a[i]; a[i] := a[r]; a[r] := t;
  Quicksort(l, i − 1);
  Quicksort(i + 1, r);
end;
```

Figure 4.1: Quicksort [35]

Sorting algorithms are usually analysed in terms of array element comparisons and exchanges [192]. Similarly, views of Quicksort require information about changes in the array in terms of these operations, rather than low-level array accesses. In Figure 4.1, all comparisons are carried out by the $\leq$ and $\geq$ operators. Exchanges are not explicitly identified as they are performed by sequences of several array assignments. The primitive array assignment operations can be identified easily, and important element comparisons can be detected if the program component's monitoring distinguishes comparisons of array elements from those of indices. Unfortunately, this approach cannot detect that the assignments were part of an implicit exchange operation.

---

[1]The Quicksort code examples in Figures 4.1, 4.2, and 4.3 are adapted from Brown [35] and are written in PASCAL.

Of course, if an Exchange operation is provided to swap array elements, the Quicksort procedure may be rewritten to use it (see Figure 4.2). This makes the exchange operations explicit, so Exchange actions can be detected within the program and forwarded as changes to views.

```
...
begin
  v := a[r]; InitLeftPtr(l − 1); InitRightPtr(r);
  repeat
    repeat IncLeftPtr(i) until Compare(a[i],v,'>=');
    repeat DecRightPtr(j) until Compare(a[j],v,'<=');
    Exchange(a[i],a[j]);
  until j ≤ i;
  Exchange(a[i],a[j]);
  Exchange(a[i],a[r]);
  Quicksort(l, i − 1);
  Quicksort(i + 1, r);
end;
```

Figure 4.2: Quicksort with Explicit Exchange operations [35]

This option is discussed by Brown, but is dismissed in favour of annotating the target program. In the following extracts, his *entities* and *deltas* are essentially our *program abstractions* and *changes*:

> Given a properly modularised Smalltalk program, one just needs to specify how the objects and messages map into entities and deltas ... which could then be monitored automatically.
> ...
> It is tempting to believe that such a strategy is a panacea. However algorithms from textbooks and journals are given in "straight-line" code; they are not broken into procedures.
> ...
> The approach we have taken ... is to annotate algorithms with "events" rather than forcing the algorithmatician to radically proceduralize his algorithm to encapsulate each meaningful operation. This approach minimizes the changes to the algorithm, since the algorithm is augmented, not transformed.

> Marc Brown, *Algorithm Animation* [33].

Brown rejects using explicit procedure calls to represent abstract operations for two main reasons: the target programs in which he is interested, examples from standard textbooks, are not written in that style, and inserting annotations is less effort than restructuring the target program. As described in the previous chapter (§3.2.3), in this study we are willing to assume that the target program is written in paradigmatic style, so that its abstractions are explicit in its text, and we wish to avoid the need for post-hoc annotation or restructuring. Operations which are explicitly present in the target program can be visualised easily within the APMV model, and strategies can be employed by the visualiser to synthesize particular operations if that is required.

This is a design trade-off related to the different application domains of Brown's algorithm animation and our program visualisation. Algorithm animation addresses small programs which contain great procedural complexity: the detailed operations occurring in these programs are very important in their visualisation. Abstract program visualisation addresses moderately sized programs made up of program abstractions, where each abstraction, considered individually, is not particularly complex. Presenting serviceable views of all the abstractions in the program is more important than presenting very detailed views of any particular abstraction.

4.1.3   Proof Properties

Part of Brown's annotated Quicksort is shown in Figure 4.3. As well as Exchange annotations, it includes another annotation, InPlace, which does not correspond to any operation in the target program. Rather, this annotation describes a property of the execution of the algorithm. When it is reached, the i'th array element has reached its final position in the sorted sequence — thus it is "in place". BALSA's views use colour to differentiate between those elements which are in place, and those elements which are yet to reach their final positions.

Cox and Roman [186] consider that algorithm animation should illustrate those aspects of an algorithm which are important to its correctness — the algorithm's *proof properties*. Their PAVANE system includes an explicit stage (the proof mapping, §2.4.1) to extract this information from the program. When visualising Dijkstra's shortest path algorithm, for example, they have rules identifying the current shortest path so that it can be highlighted in the display.

```
    ...
        t := a[i]; a[i] := a[j]; a[j] := t;
            Event( Exchange, i, j );
        until j ≤ i;
        a[j] := a[i]; a[i] := a[r]; a[r] := t;
            Event( Exchange, i, j );
            Event( Exchange, i, r );
            Event( InPlace, i );
    ...
```

Figure 4.3: Fragment of Annotated Quicksort [35]

The InPlace annotation in Figure 4.3 captures a proof property of Quicksort — that after the (sub)array to be sorted has been partitioned, the pivot element is then in place. Whereas annotations such as Exchange correspond to abstract operations within the program, proof annotations such as InPlace (and PAVANE's proof mapping rules) correspond to assertions about the program [133]. The Exchange annotation indicates that an operation has just been executed, whereas the InPlace annotation indicates that a particular condition now holds.

Proof properties describe correctness properties of a program's design, rather than the abstraction structure of that design. They are thus much closer to aggregate abstractions than program abstractions. They are unlikely to be parts of the structure of the program[2], although, like aggregate abstractions generally, they are usually expressed in terms of program abstractions. In Figure 4.3, the InPlace annotation describes a property of the sequence abstraction being sorted — it does not depend upon that sequence being implemented as a PASCAL array.

Proof properties can be visualised within the APMV model, using the techniques employed to support aggregate abstractions. Strategies can use callbacks and changes to monitor objects in the program and evaluate their proof properties. For example, the array abstraction could be queried and each element checked to determine if it was in place. These strategies may not be particularly simple, for example, determining which array elements are in place involves first sorting all the elements and then checking the position of each element. Alternatively, Section 4.3.4 describes how the APMV model can be extended to support annotation directly.

4.1.4   Retrievability

Views send callbacks to objects to retrieve information about those objects' state. Callbacks can only be sent to accessor messages defined by objects. Unfortunately, objects do not necessarily provide suitable

---

[2]Some languages, such as EIFFEL, provide syntax for expressing assertions and loop invariants within the program text. The InPlace annotation is really an assertion about the Quicksort algorithm, so in such a language perhaps it could be captured in the program.

accessors. Consider a more basic version of the stack presented above (Table 3.1) which provides only new_stack, push, pop, and isEmpty operations. This interface is sufficient to provide an unbounded stack: the target program can push elements onto the stack and pop them off in LIFO order. The stack must maintain a record of all the elements it currently contains. Unfortunately, these elements cannot be retrieved through the stack interface without altering the stack. All elements can be accessed in turn by a series of pop operations, but as each pop removes an element from the stack, at the end of the series the stack will be empty. If this technique is used to recover state information from a program, the visualisation will have rather strange effects.

The minimal stack's interface does allow some information about its state to be retrieved without side effects: the isEmpty operation returns true if there are no elements in the stack. An abstract view of a minimal stack could display only an indication of whether or not the stack is empty, ignoring the number, type, or values of the stack's elements, since this extra information is not available through the stack's interface. This perspective of a stack is similar to the approach of many soft-drink dispensing machines, which do not display the number of cans of product remaining, but simply illuminate an indicator lamp when the supply of a given product is exhausted [142].

If this minimal view is not acceptable, a more complete view can of course be constructed by examining the stack's implementation. The stack's interface and encapsulation can be ignored, and its representation accessed directly, as in a graphical debugger. But such a view illustrates the stack's implementation, not the stack abstraction.

An aggregate abstraction can also produce a more informative view. For example, a strategy could monitor all the push and pop messages the stack receives, and use this information to build a model of the stack's contents.

4.1.5 Summary

The APMV model exploits a correspondence between the target program's structure and its design, a correspondence which we believe will be found in many well-designed object oriented programs. To be visualised easily, the target program must be written in a style which uses the language's structuring facilities to capture its design, and that design must explicitly represent the abstractions and operations which the visualiser and user consider important. The APMV model can also be used to visualise abstractions which are not well represented in the target program, as strategies can be used to build aggregate abstractions which model the abstractions the user wishes to see.

4.2 Callbacks

Callbacks are messages sent from views via strategies to objects in the target program. They are used for several purposes, including providing initial information to a view, updating a view after its target object has changed, and executing commands from the user. Strategies can also send callbacks themselves; this is particularly useful for constructing aggregate abstractions.

The use of callbacks is an important strength of the APMV model, as it allows the program visualisation system to employ the definitions of abstractions within the target program. There are two particular difficulties with this approach, however. First, since the callbacks may be sent while the target program is running, their execution must be synchronised with the target program. Second, because strategies (and through them, views) rely on the results callbacks return, any errors in the target program can affect the correctness of the visualisation.

4.2.1 Synchronisation

Callbacks are essentially message sends, and will invoke methods in the target object in the same way as messages sent from within the target program. Like any other sends, they will work correctly only when the target object is able to receive them. Unfortunately the target program could be modifying the

target object at the same time as that object receives a callback. Callbacks can therefore be sent only to objects which are *quiescent* (not executing any messages), and once a callback is executing, the target program must be prevented from modifying any objects used by the callback.

Consider a linked-list object while a new member is being added to the list. The representation data structure of the list will pass through several inconsistent states — it is possible the list could be temporarily circular. A callback sent to the list object at this time is unlikely to return a correct result, indeed it may loop continually around the list or cause the entire computation to abort. A similar situation can occur if the target program attempts to access the list while the list is being modified by a callback (presumably a callback sent in response to a command from the user). Thus callbacks must be synchronised so that they do not interfere with the target program's execution, and vice versa.

When the PV system needs to send a callback to some object in the program, it must first determine whether it is safe to do so, i.e., whether the target object is quiescent. If so, the callback can be performed directly, otherwise it must be delayed until it is safe to execute. The target program can be prevented from interfering with an executing callback simply by suspending the target program until the callback has returned.

The safety of a callback can be determined in various ways:

- The target object may be sufficiently simple that its state is always consistent.

- The active threads of control within the target object can be monitored.

- Synchronous callbacks can be sent only when it is certain the target object can receive them.

Simple Target Objects

Some objects, such points, rectangles, and arrays, are sufficiently simple that their representation can never be in an inconsistent state. Callbacks can always be sent to such objects, without any need for further synchronisation.

Monitoring Object's Activity

The program component can monitor objects in the program to detect when they receive messages and when those messages return. The PV system can thus keep track of objects actively processing messages, i.e. those objects which have received messages which have not yet returned (§4.3). If a callback is sent to an active object, it must be delayed until the object is no longer active.

Delaying callbacks if any messages are active within the target object is a conservative approach: it will avoid any errors but may delay the callback unnecessarily. If both the callback and all active messages are accessors (i.e. they do not change the internal state of the target object), their execution can be interleaved without any ill effects. This is a common situation in concurrent systems where multiple reader processes may access a shared data structure.

Synchronous Callbacks

If a view uses callbacks to update its display, then presumably it can only respond to changes when callbacks can be sent to the target object. Rather than forwarding changes to the view and then synchronising the resulting callbacks explicitly, a strategy can forward changes only when the target object is able to receive callbacks. We call such callbacks *synchronous*, as they sent as a direct response to a change detected in the target object.

Strategies which use the program component to monitor the target object's actions can implement this approach simply. A strategy only sends changes to its view in response to a *top level* action in the target object: either a message received by an object known to be quiescent, or the return of the last active message send within an object.

Initialising Views

Callbacks used to initialise views pose a unique problem. The techniques for synchronising callbacks described above depend upon information about the recent execution history of the target object — in particular, whether any threads of control are active within the object. This information can be discovered by the program component, provided the target object is being monitored. When a view is first attached to an object, it will not have been monitored, so execution information is not available. In this case, the callback must be delayed (and the view initialisation postponed) until the required information can be gathered.

4.2.2   Dealing with Errors

A program visualisation system based on the APMV model makes a useful debugging tool, because it can display the target program in terms of the important design abstractions within it. This gives the user precisely the information needed to find and correct bugs in the target program. The APMV model can present multiple views at multiple levels of abstraction, so the user can see different perspectives of the program. For example, an abstraction can be directly compared with its implementation. Since the APMV model automatically monitors the program, these abstract and dynamic views can be produced without the user needing to modify the program or write mapping rules.

Unfortunately, like any other software system, an APMV model system can contain errors due to incorrect design or programming. Compared with a more conventional debugger, the APMV model introduces two additional sources of error. First, the system's interchangeable subcomponents (views and strategies) can contain bugs. Second, mismatches between correct subcomponents and correct target programs can cause errors if the subcomponents use callbacks (§4.1.4, §4.2.1).

This section first describes how the user can detect all these kinds of errors using visualisations produced by an APMV model visualisation system. The section then describes how the user can diagnose errors, either by locating them accurately in the target program, or by determining that they are caused by the visualisation system itself. The section concludes by discussing our experience with using our prototype APMV model system as a debugger.

Detecting Errors

An APMV model visualisation system can detect some kinds of errors and bring them directly to the user's attention. Whether or not an error can be detected depends upon the characteristics of the particular error, not upon the location of the error. For example, an error in the target program, in the visualisation system, or in a subcomponent can raise an exception, such as an arithmetic overflow trap, a failed array bounds check, or a message lookup error. A callback can succeed, yet return a value which cannot be correct, for instance a negative integer as the value of the sum of an array of small positive integers. A callback can simply not return within a specified time. Views are able to catch exceptions, inspect callbacks' return values, and use timers to check that callbacks return, and so detect these kinds of errors. If a view can provide useful information despite the presence of an error, it should continue to operate after informing the user of the error, otherwise, it should cease operation.

Some errors can only be detected with detailed semantic knowledge about the design and implementation of the target program — knowledge that a visualisation system cannot always have. For example, a callback can return a perfectly plausible result, of the correct type and magnitude, but which is simply wrong. Since a visualisation system cannot in general have the knowledge to detect these kind of errors, it cannot bring them directly to the user's attention. The target program and associated displays will presumably continue to run, as no error has been detected to interrupt them. The user can, however, detect these errors by watching the displays produced by the visualisation system, and actively building up a mental model of the program from the information presented by those displays. Since the visualisation system has not detected any errors, it will continue to function, and its displays will show the effects of the error. The user can eventually notice an anomaly in a view affected by an error, and investigate further — either to diagnose the error if one is present, or to correct their mental model of the target

program if there is no error — even though the visualisation system cannot bring these kinds of errors directly to the user's attention.

Diagnosing Errors

When the user becomes aware of an error or anomaly, they may not be able to determine whether the error or anomaly is a real bug in the target program, a symptom of an error in the visualisation system or one of its subcomponents, a mismatch between a subcomponent and the target program, or the result of misunderstanding the program. If the anomaly is a result of a bug in the target program, the user may not be able to determine precisely where the bug is located from a single view. For this reason, many programming environments and graphical debuggers provide multiple views which can be used to highlight parts of the program which are particularly suspect [143] (see also §2.2.1). The APMV model uses multiple views to display the program at different levels of abstraction, so the user can identify bugs by comparing the displays of abstractions and their implementations.

Imagine a doctor examining a patient with a suspected fractured leg. The doctor will use several different techniques to locate the injury. For example, the doctor will inspect the skin around the suspected fracture for signs of bruising, ask the patient if the leg will support their weight, and attempt to feel the contour of the leg bones. If these high level tests indicate the leg is likely to be fractured, the doctor will use a lower level test such as X-ray photography to check the state of the bones directly, and will take several X-ray photographs from different directions to be sure of locating the fracture. Similarly, the APMV model's multiple views can be used to identify bugs in the target program. When one view indicates an anomaly, the user's natural course of action is to open one or more other views to display the anomaly from different perspectives.

Consider debugging a target program using a linked list, where, due to a bug, the list stores only every second element inserted into it, rather than every inserted element. Perhaps by observing anomalous views of other parts of the program, the user may suspect that there is a bug in the list. The user can then request alternative views of the list in order to locate and diagnose the problem. An abstract view of the list will show the list elements clearly, focusing on element values, so that the user will easily be able to see the effects of the bug — that only every second expected element is stored in the list. A language level view of the list will show the list's implementation in detail, including link records, pointers between links, and so on. This detail can obscure the elements' values, but once the presence of the bug has been confirmed from the abstract view, the language level view can be used to isolate the cause of the bug in the list's implementation.

The APMV model's multiple views enable the user to detect errors in the visualisation system without any special vigilance. When presented with an anomalous view, the user will naturally open one or more additional views on the object displayed in the anomalous view, and each of these views will use different system subcomponents. By comparing these views, the user can determine whether the visualisation system or the program is erroneous.

Consider again an abstract view of a linked list which displays only every second expected list element. Seeing only this view, the user cannot know whether the view is correct and somehow every second element has been omitted from the list, or whether the list's contents are correct and the view is erroneous. Upon detecting the anomaly in the view, the user can request additional views of the list, such as a language level view, or perhaps a library view at the same level of abstraction as the list view, but with a simpler graphical design and a simpler implementation. If the additional view shows all the expected elements in the list, then the abstract linked list view is more likely to be at fault, but if the additional view shows only half the expected elements, the fault is more likely to lie in the target program.

Mismatches between subcomponents (views and strategies) and the target program can be identified in the same way. For example, a third possible reason for the problems in the linked list view is that the program and the view are correct in isolation, but the view does not operate correctly when displaying that particular target object. Perhaps the view sends a callback message which the list object implements by returning every second list element. Alternative views which use different callbacks would display the list correctly, and the user could detect the location of the error by comparing the views.

Our Experience

We have found the prototype APMV system Tarraingím (described in the following chapters of this thesis) to be a useful debugging tool in practice. For example, we visualised various sorting algorithms, using implementations described in Segewick's *Algorithms* [192]. These algorithms are written in PASCAL, and use arrays indexed from one. The Tarraingím prototype is written in SELF (§5.4) and, as a consequence, uses arrays (known as vectors) indexed from zero. We overlooked this difference in transcribing Shellshort and the resulting program ignored the first element of the SELF vector being sorted.

We visualised Shellsort using Tarraingím, first using a textual collection view (the vector view from Figure 3.1). Since the error was very small, affecting only one element, it did not stand out in the textual view. We cross-checked this result using a graphical view (the dots view from Figure 3.1) as an alternative to the textual view. This graphical view made the dynamics of the algorithm very clear, and it was immediately obvious that the zeroth element of the vector was not participating in the sort.

Using Tarraingím as a debugger also tended to expose bugs in its implementation, in particular, bugs involving subcomponents and callbacks. For example, one of the first programs we debugged involved a circular doubly-linked list object. Displaying this object with an abstract graphical IpView (see Figure 10.39) repeatedly caused both Tarraingím and the target program to crash. The cause of this bug was not obvious.

We therefore tried various different visualisations of the list to isolate the bug. First, we used low level object views to check the implementation of the individual objects making up the list. These views suffered no errors, and their displays showed that the list's implementation seemed to be functioning correctly. These views also showed that the pointers within the list were often drastically rearranged whenever a new element was added. Second, we displayed the whole list with a simpler textual abstract view. We expected that if there was an error in the list object, this view would trigger the same error as the graphical abstract view, but against our expectations the textual view performed correctly.

At this point, we hypothesised that the error must lie within the IpView, rather than the linked list object. To confirm this, we tested the list with an IpView, several low level object views and an operation trace view attached to the list, and a second operation trace view attached to the IpView. As expected, the program crashed again, but this time the alternative views were still visible, if no longer operating. The low level object views showed that the list's internal pointers were being rearranged at the time of the crash, and the operation trace views showed that a callback message had just been sent from the IpView to the list.

By combining the information displayed in these views, we could finally identify the bug — the IpView was sending a callback to the list object while that object was actively processing another operation. The list object's internal structure was in an inconsistent state, so the callback caused the system to loop, triggering the crash once the available memory was exhausted. In short, this bug was caused by a mismatch between the view and the target object — in particular, a synchronisation error. Once diagnosed, the problem was solved by ensuring the IpView used a synchronous strategy which only sent callbacks when its target object was quiescent (§4.2.1). The simpler textual abstract view used a synchronous strategy thus avoiding the problem.

4.2.3 Summary

The APMV model uses callbacks to retrieve abstract information from the target program. Callbacks are effective because they use the target program itself to provide information at the correct level of abstraction. Unfortunately this dependence upon the target program can cause some practical problems with callbacks. In particular, callbacks' execution must be synchronised with the target program, where possible by using synchronous callbacks sent in response to change notifications. Views sending callbacks must also be aware that the target program may contain errors which can affect both the success of the callbacks and the veracity of any results returned.

4.3 Changes

A change is an event in the execution of the target program of interest to a view. The program component monitors the target object as directed by the view's strategy, and provides raw information about the program's actions to the strategy. The strategy can process this information and forward changes to its view. The strategy used by a view determines the changes that view receives.

Some views' displays are directly determined by the changes they receive. For example, trace views (as in Figure 3.2) display a textual list of the changes they receive. One trace view may use a strategy which forwards all the target object's actions as changes to the view; another may forward only those actions which modify the target's local state; and a third only those actions whose message selectors begin with the string foo. The difference in strategies will be reflected directly in the information displayed in each view.

Many views do not depend on the details of the changes they receive. For example, an *activity view* illustrates whether its target object was active or quiescent. This type of information is also required to synchronise callbacks (§4.2.1). These views only need to keep track of whether their target object has received a message which has not yet returned. Details of the message, such as its name and argument values, are unimportant to such a view.

Batch views completely redraw themselves when their target object changes, using callbacks to retrieve information about their target's current state. These views need more information about their target object than just its activity, but less than a full trace. A strategy should send a change to a batch view when it detects an action that modifies its target object. Like activity views, batch views do not inspect the details of the changes they receive: they simply use the changes as a signal that their display should be redraw.

An incremental view, in contrast, is redrawn by detecting the way in which its target object has changed and redrawing only those parts of its display which are invalid. Like a batch view, an incremental view does not need be notified about every operation performed by its target object, but it requires precise information about the target object's changes — it needs to know how the object has changed.

Monitoring Plans

The program component needs to know how to monitor the target program. A strategy supplies this information to the program component as a *monitoring plan*. A monitoring plan is a set of instructions that specifies which objects are to be monitored by the program component, and which actions of those objects are of interest.

A strategy computes a monitoring plan when it is initialised, sends the plan to the program component, and then relays the actions it receives from the program component as changes to its view. A strategy monitoring all the actions of a particular object would work in this way.

A strategy may also inspect the target object when building its plan. For example, to monitor all of an object's variables, a strategy could examine the target object, determine its variables, and then build a plan monitoring assignment actions to each of these variables.

A strategy may require more selective monitoring than the program component can provide. In this case, a superset of the required actions must be monitored, and the strategy must check each action, and only forward changes for those actions which its view requires. For example, the program component may not be able to monitor only the target program's actions whose message selectors begin with foo. A strategy could request that the program component monitor all the actions of the target object, and only forward changes for those actions meeting the criterion.

The Rest of this Section

The remainder of this section discusses issues in the design of strategies regarding the monitoring plans they produce and the changes they forward to their views.

Section 4.3.1 discusses strategies which monitor modifications to their target objects. Sections 4.3.2 and 4.3.3 describe how the actions monitored by a strategy can be chosen to deliver only the information required by a view. Section 4.3.4 then describes alternative strategies which do not depend solely upon the program component monitoring the target program. Section 4.3.5 presents an example showing how these strategies can be used to improve the efficiency of the monitoring system. The discussion of changes in summarised in Section 4.3.6.

4.3.1 Monitoring Modifications

Many views depend on their target object's state. Such views must be updated whenever that state is modified. A strategy can detect operations which alter its target object, and then send changes to inform its view.

Mutators vs. Accessors

Messages can be categorised as either accessors (which return information about the object but do not change it) or mutators (which change the object) by analogy with the classification of ADT operations (§3.5). Constructor messages are not generally sent to objects since the object to be constructed does not yet exist. If they are sent directly to objects (as in SMALLTALK's new or SELF's clone) they can be treated as accessors which return a new object as a side-effect. Since accessors do not alter the target object, they can be ignored by views interested only in modifications to the object's state.

To ignore actions involving accessors, a strategy must be able to determine whether a message is an accessor or a mutator. Unfortunately, making this decision *ab initio* requires a detailed global analysis of the target program. But since this distinction is part of the program's design, methods in the program can be marked to indicate whether they are accessors or mutators [101].

Several programming languages support this distinction. EIFFEL, for example, differentiates between routines declared using the function keyword that should not change the object's state, and those declared using the procedure keyword that are unrestricted [141]. C++ similarly allows accessor member functions to be marked const [210]. Alternatively, If this information is not provided in the program, the visualiser can explicitly identify accessors when writing the strategy.

Immutability

Some objects are immutable, that is, their state never changes. An immutable object has no mutators: all its messages are accessors. In many object oriented languages, integers, floating-point numbers and characters are immutable; some languages include immutable versions of other types such as strings, symbols or records. Since an immutable object cannot change, its does not need to be monitored. Objects can also be marked as immutable in some languages. This is the case in C++, which uses const to declare immutable objects[3] (all const objects' member functions must also be const).

An object may be immutable when considered as an abstraction, having only accessor messages, but its implementation may modify local state, presumably for reasons of implementation efficiency. For a simple example, consider an object providing a read-only interface to a large external database. To avoid repeated database accesses for frequently-retrieved entries, the interface object may cache them in its local state. This cache, and the mutable state used to implement it, is hidden within the object and invisible outside its interface.

Whether such an object can be visualised as immutable depends upon the level of abstraction of the view. If the view displays an interface perspective of the object, the object can be considered immutable; but if the view displays an implementation perspective, the object must be treated as mutable, and its actions monitored.

---

[3]C++ const objects can be dynamically initialised while remaining const, provided the initialisation is performed in a constructor.

Summary

Both accessor messages and immutable objects can be identified to the visualisation system via the programming language or information supplied by the visualiser. If a view only needs to be notified when its target object's state changes, a strategy can simply ignore accessor messages and immutable objects when building a monitoring plan.

4.3.2   Operation Granularity

Operations in the design of an object oriented program are represented as messages sent between objects in the program's structure. For example, a Quicksort procedure sends messages to read and to write the array to be sorted, exchange two elements, and recursively sort a partition. When the target program executes, this task structure is reflected in the dynamic structure of its actions. In particular, message sends can be nested. For example, the main Quicksort message makes a nested recursive call to Quicksort to sort each partition, and the exchange operations make nested calls to individual element operations.

Many views do not need information about all of these operations. A view displaying Quicksort partitions (say Figure 3.2) needs information only about the recursive calls, while a view analysing the algorithm may be only interested in the exchanges.

We say that each of these messages implements different *granularity* operations. The idea of granularity is similar, but not identical, to that of the level of abstraction in procedural decomposition [62]. Operations of different granularities do not form independent layers, unlike operations structured by procedural decomposition (§3.3). Rather, large granularity operations are used in addition to smaller granularity operations, and operations of different granularities may belong to the same level of abstraction. For example, the Quicksort operation is implemented by using exchange operations, which in turn use element indexing operations. A client of an array may use the large granularity Quicksort operation to sort the array, but must also use smaller granularity operations to initialise each element of the array and to retrieve the sorted information.

The visualiser can take account of messages' granularities when designing strategies. Strategies can safely ignore large granularity messages, if the large granularity messages use smaller granularity messages to do their work. Alternatively, small granularity operations can be ignored, provided they are called from other operations which are sent as changes to the view. For example, a view displaying the internal operation of Quicksort may wish to ignore the actual Quicksort operation, and update itself using the smaller granularity exchange operations. A suitable strategy would send exchange actions as changes to the view, and ignore Quicksort actions. The exchange operations will presumably call single element array operations, but the view does not have to be notified of these actions, as their behaviour is subsumed by the exchange operations.

4.3.3   Choice of Actions

The program component monitors the actions of objects in the target program. Each message send in the program can cause two actions to be sent to a strategy: a *receipt* action when the message is received, and a *return* action when its execution is completed. A strategy can forward either or both of these as changes to a view. A monitoring plan can request that the program component monitor either or both of these actions.

Receipt Action

Responding to a change resulting from a receipt action allows a view to react as soon as a message is received by the target object. The target object's state cannot have been affected by the execution of the message. A view can then send callbacks to retrieve old values from the object. This information is useful when implementing incremental views, allowing them to erase any old values displayed (§6.4). New values can sometimes be retrieved from the arguments of the message causing the change.

Return Action

Views can alternatively respond to the final action, that is, they can be notified when messages have completed execution. This has several advantages. First, the change notification can record whether the operation has completed normally or exceptionally, so the view can react appropriately. Second, when an action is complete, any changes to the target object caused by the message must also be complete. The view may now send callbacks to retrieve the new (i.e. current) values from the target object, so this is often used by batch views. Finally, the value returned from a message send is only available once that send has completed. Of course, a view which is updated in response to final actions will only be updated once the detected operation is complete.

Both Receipt and Return Actions

Responding to both receipt and final actions provides a view with all the information mentioned above. Two types of views in particular need this information: control flow views and smooth animations.

Control flow views (such as execution traces, call trees and call graphs) present information about message sends and method activations. The precise shape of the call tree is very important for such views, and in particular, the nesting structure of message sends. Both actions are therefore necessary, receipt actions to signal that a particular method has started executing (any further receipt actions indicating a nested message send) and completion actions to signal that the current method has returned.

Smooth animations can be used to display operations. When the view receives the change from the receipt action, the animation is started, and it is completed when the change from the matching return action is detected.

4.3.4 Alternative Strategies

The APMV model is designed with the idea that strategies receive actions from the program component, and that these actions are generated by monitoring the target program. The APMV model is flexible enough that alternative strategies, which may not conform strictly to this model, can also be used. In this section we briefly discuss some alternative strategies:

Null The simplest possible monitoring strategy is not to monitor the target object at all. Static views use null strategies (§3.7.3), that is, they use a callback to obtain an initial display which will not change. Static views are obviously quite efficient since the target object need not be monitored, and can be also used to provide a snapshot of an object at a particular time.

User Request A view can be updated only when the user explicitly requests it. This strategy is in practice similar to the null strategy described above, except that the view will occasionally send callbacks to retrieve the current state of the target object. Of course, such callbacks are not synchronous, and so must be synchronised with the target program (§4.2.1).

Polling This strategy is very similar to the user request strategy, but rather than a view being updated irregularly according to the user's request, callbacks are sent at regular intervals. Like the user request strategy, the callbacks are not synchronous.

Local State Change As well as detecting the messages an object receives, many monitoring systems can detect changes to the target program's memory — that is, to an object's local state. Depending upon the monitoring system, monitoring memory may be more efficient than monitoring messages. A view can be updated in response to state changes rather than the messages it receives. When a change in the target object's implementation state is detected, a callback can retrieve its abstract state. This strategy's weakness is that it cannot detect the details of the way an object has changed, since it does not take cognisance of the operations within the program. It is suitable for batch views, but not incremental views (§3.7.3).

Caching Many strategies, both those which monitor the program's actions and the alternatives described in this section, can benefit by caching information retrieved from the target object. A polling or local state strategy, for example, could cache the target object's state. When information is retrieved from the target object it is compared with the cache. If they differ, a change notification is forwarded to the view, and the cache is refilled with the new information. If they match, no change notification needs to be generated as the current display will still be correct.

Annotation In BALSA and ANIM (§2.3), annotations are implemented as extra procedure calls added to the program by the visualiser. Annotations can be used within the APMV model in much the same way: the visualiser can insert an empty method into the program for each type of annotation, and code may be annotated by sending those messages. The monitoring system will detect actions resulting from these messages, which can then be processed by views. If required, a custom strategy can be used to discard all other actions.

Inference A strategy can perform arbitrary computations on the actions it receives before forwarding changes to views — indeed the changes a strategy produces may bear only a tenuous relation to the actions it receives from the monitoring system. A strategy can embody a set of inference rules acting on the program's code or data to reconstruct program abstractions which are not accessible from the program's design (§2.4). This is very similar to the way strategies can be used to build aggregate abstractions.

The APMV model was not really designed to support strategies such as these, although it is flexible enough to admit them. If the target program is designed in accordance with the model, annotations and inference rules should not be required to identify abstractions and operations. These strategies can, however, be accommodated within the basic structure of the model, facilitating a hybrid approach to program visualisation, involving monitoring, inference, and annotation. We believe that a hybrid approach is worth study, even if only because it provides a neutral basis for comparison of the various component techniques. Such a study is not an aim of this project, so we have not pursued it.

4.3.5   Efficiency

Annotation based systems describe a program's abstractions to the visualisation system in a form which facilitates efficient visualisation. When annotating a program, the visualiser identifies only those events required by the view. This avoids generating spurious events, drastically reducing the amount of information the PV system must process.

For example, an annotated Quicksort sorting a fifty-element array generates about five hundred exchange events in ANIM (§2.3.2). Naïvely monitoring the array object alone could easily generate five thousand events, and a system processing all these events would presumably run ten times as slowly.

The number of changes a view receives determines the efficiency of the system. If a view is notified of a change after every action of its target object (rather than every action which actually results in the view's display changing) the view can be redrawn unnecessarily. This may slow the execution of the visualisation system, but will not alter the information presented in the view.

Consider the quicksort trace view illustrated in Figure 4.4. This shows all the actions of Quicksort executing upon a two-element vector[4]. The two elements of the vector begin in reverse order; one exchange operation is required to sort the array.

The operations appearing Figure 4.4 are explained in Table 4.1. The trace lists the receipt and completion of messages in the target program; completion actions display a return value after the "≫≫" symbol. When quickSort is called, the vector's size is checked (actions 2 and 3), a partition is found (actions 4 to 9), the two elements are exchanged (actions 10 to 19) and then the computation gently unwinds.

A BALSA-style annotated Quicksort (such as Figure 4.3) would generate one Exchange event, possibly an InPlace event, and perhaps two events to mark a Quicksort's call and return. ANIM, being a more basic system, would generate only one Swap event.

---

[4]Figure 5.9 contains the SELF source code for the Quicksort which was used to generate Figure 4.4 in our Tarraingím system.

Quicksort protocol

size	Returns the size of the vector to be sorted.
quickSort	Quicksorts the vector.
quickSortFrom: f To: t	Quicksorts the partition between f and t.
at: e	Returns vector element e.
at: e Put: v	Assigns v to vector element e.
swap: a And: b	Exchanges two vector elements.

Table 4.1: Quicksort protocol

```
quicksort trace

□ 1   quickSort
  2     size
  3     size »» 2
  4     quickSortFrom: 0 To: 1
  5       at: 0
  6       at: 0 »» 1
  7       at: 0
  8       at: 0 »» 1
  9       at: 1
 10       at: 1 »» 0
 11       swap: 0 And: 1
 12         at: 0
 13         at: 0 »» 1
 14         at: 1
 15         at: 1 »» 0
 16         at: 0 Put: 0
 17         at: 0 Put: 0 »» vector{0, 0}
 18         at: 1 Put: 1
 19         at: 1 Put: 1 »» vector{0, 1}
 20       swap: 0 And: 1 »» vector{0, 1}
 21     quickSortFrom: 0 To: 1 »» vector{0, 1}
 22   quickSort »» vector{0, 1}
```
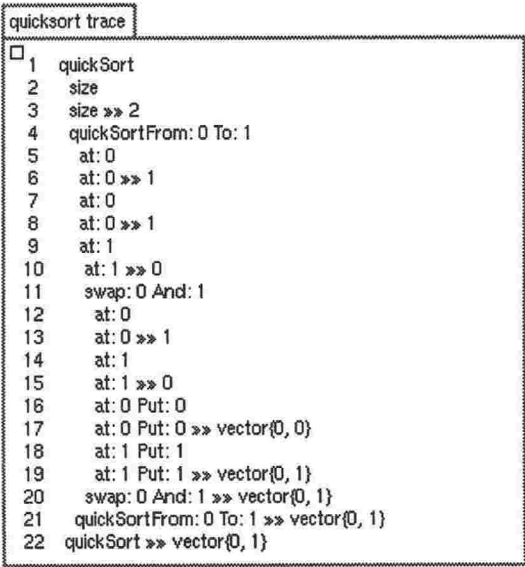
Figure 4.4: Quicksort Trace View

Unfortunately, the trace in Figure 4.4 contains twenty-two separate actions. Assuming all actions take the same time to process, a naïve APMV visualisation could run between five and twenty times slower than an annotated visualisation, without considering any overhead imposed by the use of a monitoring system rather than annotations. This is unacceptable. The techniques described in this section can, however, improve this in a variety of ways:

1. Accessor messages (size and at) can be discarded (§4.3.1). This saves twelve actions — 2 and 3, 5 to 10, and 12 to 15.

2. Large granularity messages, quickSort, quickSortFromTo and swapAnd can be discarded (§4.3.2). This eliminates six actions — 1, 4, 11, 20, 21, and 22.

3. Eliding the swapAnd event is a little counterproductive, but the nested smaller granularity actions can be ignored (§4.3.2), which will prune the eight deeper actions 12 to 19.

4. Either all receipt actions or all return actions can be ignored (§4.3.3). This removes eleven actions.

5. The visualiser can write a custom strategy to capture only those actions actually required — perhaps receipts and returns of quickSortFromTo and returns from swapAnd.

Using one or more of these techniques ensures that only three or four events, roughly corresponding to the annotations, will be generated. Unlike an annotation-based system, the actions can be identified without modifying the target program, or even inspecting the code of Quicksort itself.

The trace view in Figure 4.4 is generated using a very simple strategy: all the target object's actions are monitored. The same is true for the graphical views of Quicksort in Figures 3.1 and 3.2. Customising the strategy as outlined in this section can increase the speed of the display, but is not necessary to produce visualisations. This is an important feature of our approach. The complex strategies described in this section are principally concerned with the efficiency of the program visualisation system. In general, they do not involve the correctness of the visualisation, and so they do not have to be used. An inefficient preliminary visualisation can be constructed using simple, conservative strategies, which can be replaced by more optimised strategies as more information is gathered about the program's behaviour, or better performance is required.

4.3.6 Summary

The APMV model uses changes to send views abstract information about the execution of the target program. Changes are effective because they use the target program itself to provide information at the correct level of abstraction. Visualisers writing strategies can use several techniques to filter the actions the strategies receive, so that views are sent only those changes they really require.

4.4 Aliasing

Aliasing within the target program causes problems for the APMV visualisation model. The model relies upon monitoring an object's actions to detect all the changes to the program abstraction implemented by that object. In the presence of aliasing, a program abstraction can be changed without sending messages to its corresponding object.

The *Geneva Convention on the Treatment of Object Aliasing* [102] includes a good survey on aliasing in object oriented programming.

4.4.1 Aliasing in Object Oriented Programs

Aliasing can cause problems for the APMV model whenever a program abstraction is implemented by more than one object in the target program. We call the set of objects implementing such an abstraction an *object complex* [146] (also known as a *demesnes* [225] or an *island* [101]). One of the members of the complex, the *head* object, provides the interface to the whole abstraction, and thus to all member objects of the complex.

Aliasing causes problems for program visualisation whenever an alias to a complex's member object exists outside the complex. Messages can be sent to that member object via the alias, without reference to the head object. These messages can modify the member object, and thus the program abstraction implemented by the whole complex. A view of the abstraction receiving changes from a strategy monitoring the head object would not be notified of this modification, because the head object was not involved.

Consider the stack example from Section 3.1.2. The stack is implemented by the stack object, and two components: an integer and an array, stored as variables named index and contents in the stack object. The stack object is the head of the complex, which also contains the integer and the array. If a reference to the contents array exists from outside the stack object (that is, if the array is aliased) the contents of the stack can be modified by sending a message directly to the array object, without sending a message to the stack object itself. A strategy monitoring the stack abstraction would not detect this message, since it is not sent to the stack head object.

Object oriented languages provide encapsulation: an object's private local state is not accessible from outside that object (§3.6). The member objects of a complex can be stored within the head object's local state, but this protection is not strong enough to protect the members from aliasing. A method attached to the head object can return a reference to a member, bypassing the head object's encapsulation. An object complex can also include objects created outside the abstraction the complex is implementing, and these objects may have been aliased before they become members of the complex. An object's

encapsulation barrier protects only that individual object, and that object's private local state: the members of an object complex are not effectively encapsulated.

For example, references to all the elements of a stack must have existed outside the stack before the elements were pushed onto it. If these references are retained outside the stack, the stack elements will be aliased. Alternatively, a stack operation (such as top) can directly return a reference to a stack element, and this immediately creates an alias.

4.4.2   Managing Aliasing in Program Visualisation

The problem with visualising programs in the presence of aliasing is not the aliasing *per se*, but the possibility that an abstraction implemented by an object can be changed without a message causing that change passing through the object's interface.

Many aliases cause no problems for the APMV visualisation model. Immutable objects (§4.3.1) can be freely aliased, because they cannot be modified. Aliases to an object complex's head object cause no problems, since they do not bypass the head object's interface. The members of an object complex may freely alias one another, provided they are accessible from outside the complex only via the complex's head.

Aliasing only affects changes produced by monitoring the program's actions. Callbacks sent to the head of an object complex will return correct values from the abstraction, since the methods in the target program used by callbacks will correctly retrieve the current values from the members of the complex, assuming the callbacks do not use caching, (§4.3.4). Aliasing does not affect strategies which do not rely on monitoring the target program's actions, such as polling (§4.3.4). Unfortunately, such strategies cannot produce information about abstract changes in their target objects.

Monitoring-based strategies can be adapted to deal with aliasing. If the visualiser can determine the members of an object complex, a strategy can be written that monitors the whole complex, rather than just the head object. If a complex member is modified, and the modification is not the result of a message sent to the head object, the strategy can notify its view that its target object has changed, and the view can be redrawn using a callback to gather information.

4.4.3   Summary

The APMV model does not cope well in the presence of aliasing in the target program. If an alias is used to subvert an object's encapsulation barrier, any changes caused by that alias will not be detected by program component's monitoring of the target program.

Strategies can mitigate the effects of aliasing. For example, strategies which rely only upon callbacks are immune to aliasing. If all the objects participating in the target object's implementation can be monitored, any actions caused via aliases can be detected.

1. One man's constant is another man's variable.

Alan Perlis, *Epigrams On Programming* [168]

5

A Program Exploratorium Prototype

The previous two chapters have described the APMV model of program visualisation, and how this model might be used to design a program exploratorium. This chapter introduces the design and implementation of Tarraingím, a proof-of-concept prototype we have built to explore that model. The first section describes the requirements of an abstract program exploratorium (§5.1). Section 5.2 discusses the architectural design decisions behind Tarraingím, and Section 5.3 introduces Tarraingím's detailed design, based upon an object oriented framework. Section 5.4 introduces the SELF programming language upon which Tarraingím is based.

5.1 Requirements

Tarraingím is a program exploratorium based upon the APMV model. As such, it attempts to fulfill the aims of the project as a whole (§1.2). In particular, Tarraingím provides multiple views of programs, illustrating both their code and their data, at various levels of abstraction. Views are created on demand, to allow dynamic exploration of the target program's design and execution. Target programs do not have to be modified to be visualised by Tarraingím, although the design abstractions to be visualised must be explicitly represented in the target program's object oriented structure, or synthesised as aggregate abstractions.

This section describes Tarraingím's requirements. The three components of the APMV model are considered in turn: program, mapping, and visualisation.

5.1.1 Program Component

The program component must provide information about the target program to the mapping component, and indirectly to the visualisation component. This includes static information about the structure of the target program, and dynamic information which must be gathered by monitoring the target program as it executes.

The actual target program and target programming language implementation can be seen as outside the program exploratorium proper. The program component provides an interface to these external facilities.

The Target Program

The program component must provide an interface to the target program. We assume that the target program is written in good object oriented style (§3.2.3). This requirement is easier to meet if the program is written in a language supporting such a style, i.e. an object oriented language. To support the production of realistic examples, the language implementation must be capable of running medium-sized programs reasonably quickly. This component must also route callbacks sent from views or strategies to appropriate target program objects (§4.2).

Static Information

The program component must provide static information about the structure of the target program. If the target programming language is structurally reflexive (§2.5.5), some of this information can be supplied via the language's reflexive extension.

Dynamic Information

The program component must provide dynamic information about the target program's actions, monitoring the program's execution in accordance with monitoring plans produced by strategies (§4.3).

Monitoring the target program incurs a run-time cost in two ways. First, the act of monitoring a program directly slows its execution. Second, the information gathered by monitoring has to be processed by the visualisation system, even if only to be discarded, and this also imposes an overhead. The monitoring system should seek to minimise this cost by detecting only the information actually required by strategies and views.

In particular, monitoring needs to be carried out on a per-object basis. A view is typically attached to a single target object and information should be gathered only about the particular objects of interest to views. A medium-sized program could easily contain several thousand objects, most of which will not be of interest to the user at any given time, and thus should not be monitored. For example, if the monitoring system always monitored all the instances of a class, information about most of the instances would have to be disregarded. Information should be about the use of the class (the instances), rather than the class definition itself (§2.5).

Similarly, only those actions of monitored objects actually of interest to strategies and views should be monitored. A particular view may be interested only in assignments to an object's local state, or only a particular set of messages (§4.3): only these assignments or messages should be forwarded to the mapping component.

As the user can request views of any object at any time, objects in the program must be able to be monitored dynamically.

5.1.2 Mapping Component

The mapping component consists of strategies which connect objects from the program component to views in the visualisation component. Strategies promote the reuse of views, insulating views' visual abstractions from the details of supplying the information to be displayed. Unlike the program or visualisation components, the mapping component does not rely on facilities external to the visualisation system.

The main requirement of the mapping component is that it must be must be able to express all the strategies described in Chapter 4.

5.1.3 Visualisation Component

The visualisation component is responsible for Tarraingím's user interface. Views actually produce graphical displays, and respond to input from the user. The actual mechanics of display and input handling will

be managed by a graphics and user interface system external to the PV system proper: the visualisation component must provide an interface to this external service.

Graphical Output

The graphics system must support multiple windows which can be created and manipulated interactively. Views must be dynamically updated to reflect the current state of the program (based upon information about the program received from the mapping component). Graphics system design is not a focus of this project, and so any modern output system should suffice.

Graphic Design

Devising new graphical designs for program visualisation is also not a focus of this project: indeed, we are happy to display more-or-less "standard" illustrations of program abstractions using basic graphic design techniques. We are not concerned with the application of novel techniques, such as colour, sound, 3D, or virtual reality, for their own sake [38, 117]. New presentations are continually being developed [37, 76], and ideally Tarraingím should be able to display these.

User Interface

The user interface must allow end-users to manage the display windows, navigate through the target program, alter the visual properties of views, and provide direct input to program abstractions displayed in views. The required facilities (adjusting window layout and handling locator input devices) are conventional and are provided by most interface systems.

5.2   Architectural Design

This section discusses several architectural issues in the design of Tarraingím.

1. Should the visualisation system and target program share an address space, or should they use different spaces?

2. Should the visualisation system use the same programming language as the target program, or should two (or more) languages be used?

3. Which programming language(s) should be chosen?

4. How should the target program be monitored?

5. How should the graphics be produced?

6. How should the nominally concurrent tasks of handling user input, updating displays, and running the target program be scheduled?

This section addresses these architectural issues in the order given above. Although this order is not arbitrary, neither is it a simple sequential progression, as these choices are not independent. For example, the choice of the target programming language (and the particular implementation of that language) partially determines the design of the program component of the visualisation system, which must monitor the target program. The requirements of the program component's monitoring influence the choice of programming language. Similarly, user interface facilities may be more accessible from one language than from another.

5.2.1   Address Space

The partition of a program into several separate address spaces greatly affects the subsequent design of the program. For a program visualisation system there are essentially two choices: either the target program and visualisation system cohabit within a single address space, or the program and visualisation system are split over two or more spaces.

Multiple Address Spaces

Using several separate address spaces produces a flexible architecture. For example, the system can easily be distributed and executed on several different machines. There are several disadvantages to this design, however. The connection between the various address spaces, used to transmit actions and callbacks between the target program and visualisation system, can become a bottleneck. Objects in the target program have to be *pickled* (packaged for transfer or storage [155]) to be transmitted to the visualisation system, and similarly a callback's arguments must be pickled before being sent to the program. Objects and arguments must then be unpickled on receipt.

Using multiple address spaces raises the question of precisely how the program is partitioned between the various spaces. Should the mapping subsystem be placed with the target program (the easier to construct and control monitoring plans), placed with the visualisation component, divided between the two, or allocated a separate address space of its own?

A Single Address Space

A single address space design does not promote a clear separation between the various components of the system, nor does it facilitate the distribution of the system between several different machines. An error in the target program can directly corrupt the visualisation system, for example by overwriting the visualisation system's data structure.

A single address space design does have several advantages. In particular, it allows a more flexible internal design of the visualisation system, as the various components can communicate easily. Callbacks may be sent by simple procedure calls to the target program, without pickling or passing information across address spaces.

Tarraingím's Architecture: A Single Address Space

Tarraingím is designed to be a testbed for the APMV model, rather than a production-strength program visualisation system. The virtues of distribution and reliability provided by the use of multiple address spaces are thus less important than the simplicity of the single address space model. In particular, views and strategies (which must be programmed by the visualiser to produce displays) will be easier to write if they do not need to deal with multiple address spaces.

Tarraingím thus uses a single address space containing both the visualisation system and the target program.

5.2.2   Programming Language

The first question that must be addressed with regard to the choice of programming language is whether a single language will serve for both the target program and visualisation system, or whether the demands of the target program and visualisation system are better served by using several different languages.

Multiple Languages

Using different languages for the target program and the visualisation system provides several advantages. The target program can be written in a language best suited to capturing abstractions, while the rest

of the visualisation system can be written in language with good user interface and graphics facilities. For example the ZEUS system (§2.3.1) animates algorithms written in MODULA-3, but ZEUS views are written in the OBLIQ scripting language [154].

Using two or more languages requires that objects and communications exchanged between the target program and PV system have to be translated or shared between the different languages, similar to the pickling which is required if more than one address space is used. This is not a major problem for an event-based system like ZEUS, because the communication is in terms of named events with simple arguments. The APMV model uses callbacks and changes whose arguments can be complex objects, and the PV system must be able to inspect the structure of the target program.

A Single Language

Using only a single language is obviously not as complicated as using multiple languages. In particular, if only a single language is used, communication between various parts of the system is facilitated. The language's structural or computational reflexive facilities (if any) can be used very easily by the visualisation system. A visualisation system written in the same language as the programs it visualises should be able to be used reflexively to visualise its own execution.

If a single language is chosen, it must be suitable both for representing the abstractions in the target program and implementing the visualisation system. It must have sufficient performance to produce animated graphics while simultaneously monitoring the target program.

Tarraingím's Architecture: A Single Language

Tarraingím is implemented using only one language — that is, it illustrates programs in the language in which it is written. The choice of a single address space and a single programming language minimises the distance between Tarraingím and the target program. Ideally, a program exploratorium would merge programming and visualisation into a seamless whole: using one language and one address space should make this easier to achieve.

Using a single language and single address space also considerably reduces the amount of effort required to manage the communication between the system's components. This simplicity of design and implementation is quite important in a prototype.

5.2.3   Choice of Programming Language

Given that a single language will be used to implement Tarraingím and serve as the target language, this section evaluates possible languages. The language must meet the requirements of the program component described above (§5.1.1): it must be object oriented, provide suitable graphics facilities, and support dynamic monitoring of medium sized programs (§2.5).

This section reviews four languages and their available implementations: C++, CLOS, SMALLTALK, and SELF. We did not consider designing our own specialised programming language or reimplementing an existing language to provide the structuring and monitoring support required by the APMV model.

C++

C++ [210] is the most widely used object oriented language. It is a hybrid language, as it adds object oriented facilities to the structured language C. It is strongly typed, and includes definitions of the mutability attributes of functions and objects (§4.3.1). Because C++ is typically implemented using traditional compilation technology, it is an efficient language, but fine-grained dynamic monitoring is difficult to implement without compiler support (§2.5.3).

CLOS

CLOS [112] was developed as part of the Common Lisp standardisation effort to unify several existing Lisp object systems. Like C++, CLOS is a hybrid language. CLOS has the advantage that is fully structurally and computationally reflexive (§2.5.5), so programs can be monitored within the language. CLOS uses multiple dispatch and does not really provide encapsulation.

SMALLTALK

SMALLTALK [85] was the language which really popularised the object oriented paradigm (§3.6). Unlike C++ and CLOS, SMALLTALK is a pure OO language: it is not derived from any preexisting language, and all computation is modelled as message sending. SMALLTALK is structurally reflexive (§2.5.5), and the SMALLTALK's reflexive facilities have been used to build several monitoring systems (§2.5.2).

SMALLTALK is typically implemented by sophisticated dynamic compilation [60]: to the user a SMALL-TALK system appears to be interpreted. GNU SMALLTALK, a recent, freely-available implementation, uses a naïve bytecode interpreter implemented in C++; this is an order of magnitude slower than commercial SMALLTALK implementations, however, the interpreter can be modified very easily (§2.5.3).

SELF

SELF [218] is a language designed as a successor to SMALLTALK. SELF is simpler than SMALLTALK, which is quite a feat since SMALLTALK is itself a small language. SELF is based upon prototypes rather than classes, and subsumes both method invocation and variable access into message sending [213]. SELF includes inheritance, although between objects rather than classes, and also supports dynamic inheritance and delegation. Like SMALLTALK, SELF is structurally reflexive, but dynamic monitoring should be easier to implement because of SELF's minimal design.

Tarraingím's architecture: SELF

The most widely used language considered above is C++. Unfortunately, C++ appears to be the most difficult of these languages to monitor dynamically. In contrast, CLOS programs can be monitored easily, by taking advantage of the reflexive features of the language. CLOS, like C++, is a hybrid language: they can be used to support many other programming styles beside object orientation. These are also both rather large and complex languages.

This leaves SMALLTALK and SELF from this list above. We chose SELF over SMALLTALK for several reasons. SELF's use of prototypes rather than classes (§5.4.4), and message sends for variable accesses (§5.4.2), significantly simplifies the language and so should make monitoring easier. SELF's encapsulation support is more flexible than SMALLTALK's — individual methods and variables can be declared either public or private. Finally, the SELF compiler is freely available.

Tarraingím is thus implemented in the SELF programming language, and visualises programs written in that language. Section 5.4 contains a brief tutorial introduction to SELF.

5.2.4 Monitoring the Target Program

The program component must monitor the actions of the target program. Section 2.5 categories techniques for monitoring programs as follows:

1. Hardware monitoring.

2. Postprocessing the executable form of the program.

3. Modifying the language processor.

4. Preprocessing the program source.

5. Reflexive languages.

Special purpose hardware (§2.5.1) was not available to this project: even if it were, we would not wish to use it since it is not generally available. We prefer to avoid modifying a language compiler or interpreter (§2.5.3) for essentially the same reason: we do not wish our work to depend upon a special-purpose language implementation.

A fully computationally reflexive language (§2.5.5) would be ideal. SELF, although structurally reflexive is not computationally reflexive, so it cannot directly monitor programs' execution.

The two non-invasive techniques are pre- and post-processing the target program. Preprocessing requires parsing the target program source code and producing a modified version with monitoring statements added (§2.5.4). In a structurally reflexive language, postprocessing can be performed very easily, as the language's reflexive facilities can be used to modify the target program to detect the required actions as it executes (§2.5.2).

Tarraingím therefore uses postprocessing to monitor the target program, transparently modifying objects to be monitored so that they report their actions to the program component. Since the modification is implemented and controlled within SELF, this is effectively the same as using the structural reflexive facilities of the language to implement a computationally reflexive metasystem. The program component performs the modifications automatically and camouflages them so that the user is unaware that the target program has been modified.

5.2.5   Graphics System

A program visualisation system would ideally use a specialised graphical animation system [154, 200]. The exploratorium we are building does not require great graphical sophistication (§5.1.3). All the languages discussed above, and SELF in particular, can provide access to graphics libraries. Tarraingím uses the X window system [189] since it is widely available and well supported locally. Since X is a network-based window system, more than one user can use Tarraingím simultaneously.

5.2.6   Process Design

A program exploratorium must handle several tasks concurrently. It must simultaneously execute the target program, update animated views, and obey user commands. Concurrency in a program can be handled in two ways: either by emulation using a serial program, or supported directly in a parallel or concurrent program.

An essentially serial program can emulate concurrency by iterating through several tasks. For example, the target program may be executed until a monitored action occurs, then suspended while strategies and views are executed. User input can then be handled, and finally the target program can be resumed. This is essentially the scheme used by BALSA (§2.3). Programming in this style is complicated, but the details of the emulated concurrency can be precisely controlled.

Alternatively, if the language or runtime system supports multiple processes, each task can be executed within its own process. The resulting system will be simpler, as scheduling logic is not distributed throughout the program. On the other hand, several potentially fatal global program conditions (deadlock, shared resource protection) must be dealt with.

Like SMALLTALK and several LISP variants, SELF provides lightweight processes, and Tarraingím uses these threads to schedule the concurrent tasks. The target program runs in a foreground process, calling Tarraingím as a co-routine when an action must be sent to a strategy. Concurrent background processes handle user input, animation, and any other asynchronous processing. The additional care required to synchronise the various processes can be isolated within the core of the system, and the resulting view and strategy definitions are simpler than in the single-process case.

5.2.7 Summary

Tarraingím is implemented in SELF and visualises programs written in that language. The target program is monitored by using SELF's structurally reflexive facilities to automatically modify the target program, so that it notifies the visualisation system whenever an action of interest to a view occurs. Both the target program and Tarraingím execute in a single address space, using multiple lightweight processes to handle asynchronous requests from the user. Graphical input and output is handled by the **X** window system.

5.3 An Object Oriented Framework

The previous section has described Tarraingím's architectural design. This section describes Tarraingím's internal design, which is based on the APMV model. The design is an *object oriented framework* consisting of three *subsystems* (Monitoring, Strategy, and Display), each corresponding to one part of the model (see Figure 5.1).

The Tarraingím system itself consists of libraries of objects which are used within the framework, providing both a program visualisation environment and an extensible program visualisation kit.

5.3.1 Frameworks

An object oriented framework [109, 108] is an abstract description of an object oriented design. Tarraingím's framework specifies how the objects from the various subsystems collaborate to visualise a program. The figures in this chapter describing the framework present run-time arrangements of objects, and the relationships between them.

A framework is classically described as a static specification of several abstract classes, and the dynamic arrangement of their instances. A program is built from a framework by replacing the abstract classes with concrete objects.

A framework is accompanied by libraries of concrete objects. By using these objects within the framework, it can be put to immediate use without programming. A framework can be extended by writing new objects which meet the abstract specifications. The library objects serve to describe the operation of the framework, and as examples when extending it.

The first frameworks to be widely distributed were part of SMALLTALK. The use of frameworks then spread to other user interface systems such as MACAPP [190]. Frameworks have also been developed for programming environments (for example MVIEWS [89, 91]) and program monitoring systems (such as BEE++ [40]).

5.3.2 Framework Objects

Most of the objects in Tarraingím's framework belong to one of the three subsystems (see Figure 5.1).

	Tarraingím		
Model	Program	Mapping	Visualisation
Framework (subsystems)	Monitoring	Strategy	Display
Objects	controller encapsulator	watchers	views
	events		

Figure 5.1: Model and Subsystems

Monitoring

The monitoring subsystem implements the *Program* component of the model. This subsystem monitors the target program to gather the information about the program's actions required by the rest of the visualisation system.

Two main kinds of objects are used in this subsystem. The target program is actually monitored by encapsulators. The programming style within encapsulators is severely restricted. The interface to encapsulators is therefore provided by controllers, which also route the information encapsulators gather to the rest of the system.

This subsystem is covered in detail in Chapters 8 and 9.

Strategy

The strategy subsystem corresponds to the *Mapping* component of the model. It is responsible for linking the monitoring subsystem to the display subsystem. This involves translating the *actions* of the program (detected by the monitoring subsystem) into *changes* to be applied to the display. This is primarily carried out by watcher objects, described in Chapter 7. Different kinds of watchers embody different strategies for performing the mapping.

Display

The display subsystem implements the *Visualisation* component of the model. This subsystem has three tasks, of which the most important is to draw the graphical images making up the visualisations. It also provides the user interface for the rest of Tarraingím, and handles any user input to the visualisations.

The main object in this subsystem is the view, described in Chapter 6. A view implements a display of a particular object in the target program. Different kinds of views provide different displays.

Events

Event objects are essentially typed data packets. They are used to carry information around the Tarraingím framework, and do not belong to any particular subsystem.

Most events are *up events*, which are sent from controllers to watchers to describe the actions of the target program. Watchers forward events to views to describe the changes required in the view. Views send *down events* which are eventually routed via watchers to the objects in the program. Down events are used to implement callbacks, and may be either queries to determine the state of objects in the program, or user commands to alter the objects.

The same event object can represent both a concrete action of the target program (when it is sent from a controller to a watcher) and a high-level change (when it is sent from a controller to a view).

The implementation of events and their distribution around the framework are covered in more detail in Section 8.3.

5.3.3 Framework Arrangement

The dynamic arrangement of Tarraingím's framework is illustrated in Figure 5.2. Objects are arranged in a pipeline. An encapsulator monitors an object in the program. A controller packages the encapsulator's execution information into event objects and sends them to a watcher. The watcher processes the events before finally sending them to the view, which produces the graphical output. Some views are able to accept user input — if so, they will generate events and pass them in the opposite direction along the pipeline.

The watcher, view and event objects are abstract. Tarraingím's library includes many different concrete implementations of these objects, and the visualiser can write new versions to extend the system. The controller and encapsulator objects are concrete, and suitable for all types of visualisation. They are not replaced by the visualiser.
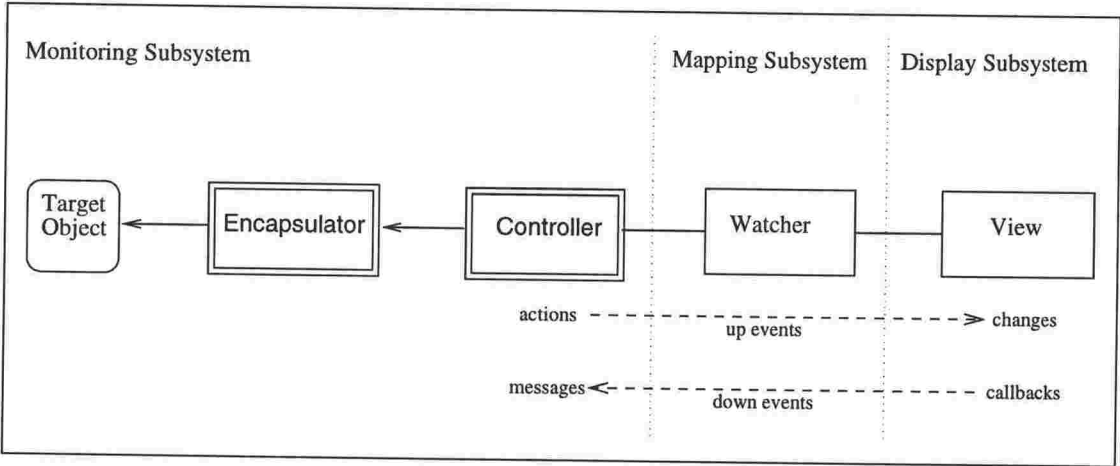
Figure 5.2: Tarraingím's Framework

Multiple Views

Figure 5.2 shows the visualisation of one object. Multiple objects can be displayed using several parallel pipelines. Each visualised object will have its own encapsulator, controller, watcher, and view. The pipeline can be generalised into a tree, allowing multiple views to display one object. Each view has its own watcher, while all views and watchers monitoring a single object share a single controller-encapsulator pair (see Figure 5.3).
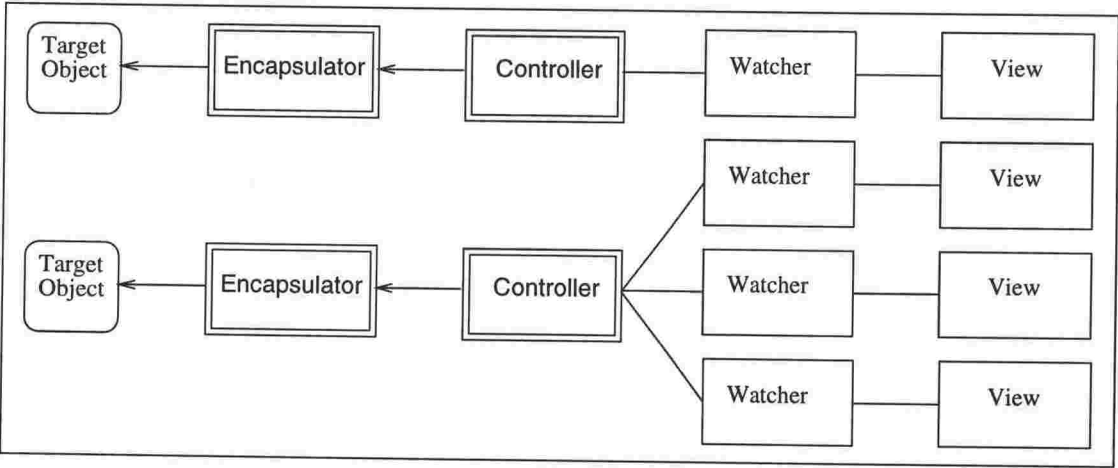
Figure 5.3: Parallel Pipelines

Composing Views and Watchers

A view can contain one or more independent *subviews*. These may display different representations of the same object, or information about several related objects (see Figure 5.4). Similarly, watcher objects can rely on *subwatchers* when implementing complex strategies. Subwatchers can also be used to allow a single view to monitor multiple objects.
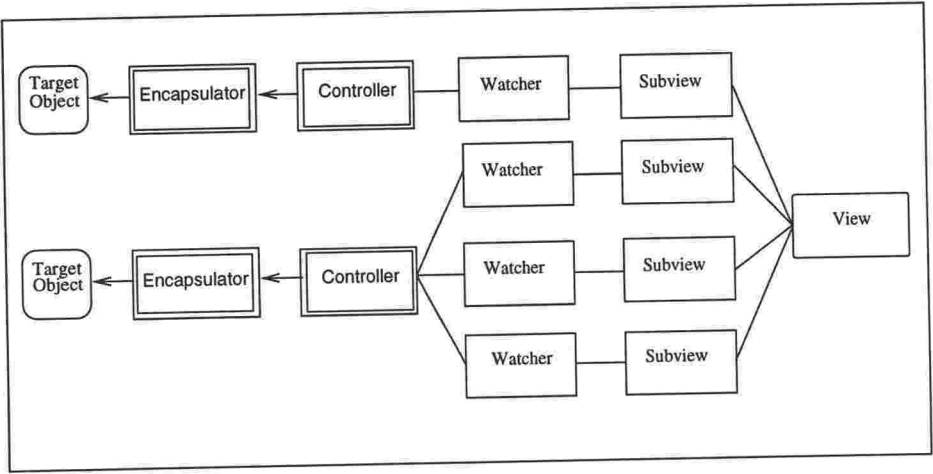
Figure 5.4: Views and Subviews

Figure 5.5 illustrates the overall scheme. The displays are produced by a forest of views, each of which can use a tree of watchers. The leaves of the watcher trees are attached to controllers and encapsulators associated with the objects they are visualising.
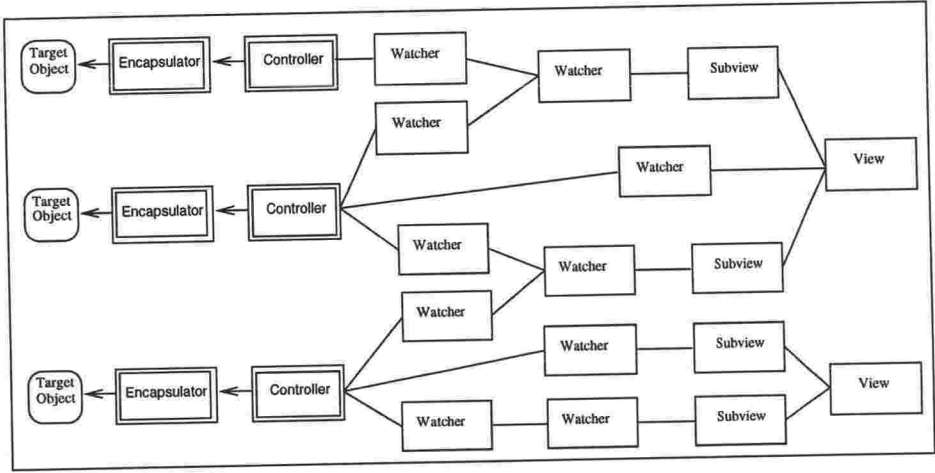
Figure 5.5: Multiple Views and Multiple Watchers

5.3.4 Inheritance Hierarchy

Frameworks are organised by inheritance. Abstract objects provide common or default behaviour which is inherited by more specialised objects. For example, Tarraingím's view object is abstract: it provides behaviour for initialisation and finalisation, but does not draw any graphics. A concrete view such as a bargraphView inherits from the abstract view, reusing the common behaviour and adding code to draw a bar graph.

Part of Tarraingím's inheritance hierarchy is illustrated in Figure 5.6[1]. Most of Tarraingím's objects inherit from tgimObject. Objects which handle events also inherit from eventClient (§8.6). The objects making up the framework (view, watcher, controller, and event) inherit from tgimObject or eventClient as appropriate, and specialised concrete objects inherit from them. Figure 6.1 illustrates the hierarchy of views, Figure 7.1 watchers, and Figure 8.5 events.

---

[1]Figure 5.6 is actually a reflexive view: it is an illustration of Tarraingím generated using Tarraingím itself.
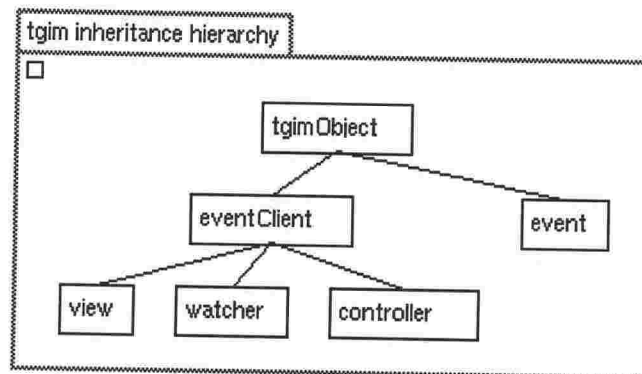
Figure 5.6: Tarraingím Inheritance Hierarchy

5.3.5 Summary

Tarraingím is implemented in SELF as a framework of cooperating objects grouped into three subsystems. The monitoring subsystem corresponds to the program component of the APMV model, and uses encapsulator and controller objects to monitor the target program. The mapping subsystem uses watcher objects to embody strategies linking the monitoring and display subsystems; it corresponds to the mapping component of the model. The display subsystem uses view objects to produce graphical displays and handle user input, and corresponds to the visualisation component of the model. Data about the program's actions and Tarraingím's changes and callbacks are passed around the framework using event objects.

5.4   Self

This section introduces the SELF programming language [215, 218] and is intended to provide enough background to allow a reader to understand the SELF examples presented in the rest of this thesis. It begins with a description of objects and expressions, describes the organisation of a SELF program, and then revisits the object oriented stack example from Section 3.6.1.

SELF has evolved in various ways through the duration of this project. This section presents the basics of the SELF language; more information about the various versions of SELFand SELF's implementation is contained in Chapter 9.

5.4.1   Objects

SELF objects are collections of named slots, and are written as lists of slots enclosed by "(|" and "|)". Each slot is associated with a message selector and may hold either a constant, a variable, or a method. For example, the example below defines an object named "fred" which defines the constant slot "pi", the one-argument keyword message "circ:" and the data (or variable) slot "size".

```
   fred = (|
        ^ pi = 3.14159265.        "constant"
        _ circ: r = (2 * pi * r).   "method"
          size ← 3.               "variable"
   |)
```

   "comments are enclosed in double quotes"

Privacy

Slots may be marked explicitly as being *private* or *public* by prefixing the declaration with "_" (for private) or "^" (for public). An object's private slot is accessible only from within that object, and that object's close relatives by inheritance. A slot's privacy may also be left unspecified: in such cases it acts as a public slot. In the above example, "pi" is public, "circ:" is private, and "size" is unspecified, so it is in practice public.

Object Creation

Within an executing program, new objects are created as *clones* (slot-by-slot copies) of other objects. Each object has a unique *identity* (§3.6). Two objects may have the same slots, and the same values contained within those slots, but if the objects were created separately, they can be distinguished on the basis of their identities. In particular, if the value of an object's variable slot is changed, no other objects will be affected [11].

Objects intended to be used as patterns for cloning are known as *prototypes*. Like LISP or SMALLTALK, a garbage collector is used so that objects do not have to be disposed of explicitly.

Some types of objects can be created using simple literals. These include numbers, strings, and blocks.

5.4.2 Expressions

SELF's expression syntax is directly derived from SMALLTALK. Since SELF is a pure object oriented language, almost all expressions are either literals or describe message sends to a particular *receiver* object. There are three types of messages: unary, binary, and keyword; a message's type depends upon its selector and the number of arguments it takes.

Unary messages simply name an operation and provide no arguments other than the message receiver object. For example, "top", "isEmpty" and "isFull" are unary messages.

Binary messages provide one argument as well as the receiver. Their selectors must be composed of nonalphabetic characters. "+", "−" and "*" are binary messages for addition, subtraction, and multiplication. Similarly, "@" creates a point from two numbers, and "##" creates a rectangle from two points.

Keyword messages are the oddest part of SMALLTALK and SELF syntax. They provide messages with one or more arguments. A keyword message has a particular arity, and the message selector is divided into that many parts. A keyword message is written as a sequence of keywords with argument values interspersed. For example, "at:Put:" is a two-argument keyword message for array assignment. The PASCAL code

```
a[i] := j;
```

is written in SELF as

```
a at: i Put: j.
```

Messages can be sent directly to the results returned by other messages. Parentheses can be used for grouping. For example:

```
draw = (style drawLineFrom: start negated To: finish negated).
c = ( (a squared + b squared) squareRoot ).
```

Implicit Self

The only expression that is not a literal object or message send is the keyword "self". This denotes the current object, that is, the receiver of the currently executing method. A message sent to "self" is known as a *self-send*. Because "self" is used pervasively in SELF (especially as messages are used to access variables, as described below), it is elided from the syntax wherever possible. The language is named "SELF" in honour of this omnipresent but mostly invisible keyword.

Accessing Variables

A variable or constant slot is read by sending a unary message corresponding to its name, and a variable slot is written by a one-argument keyword message, again corresponding to the variable slot's name. The SELF code

```
foo: 43.
bar: foo.
```

is roughly equivalent to the PASCAL

```
foo :=  43;
bar := foo;
```

if "foo" and "bar" are variables, but equivalent to

```
foo(43);
bar(foo);
```

if "foo" and "bar" are methods.

The use of messages to access variables is one of the main differences between SELF and SMALLTALK.

5.4.3 Blocks

Blocks represent lambda expressions. For example, the expression $\lambda xy.x + y$ when written in SELF is:

```
[| :x. :y. | x + y].
```

A block optionally may have arguments and temporary variables: these are written at the start of the block surrounded by "|" symbols. Arguments are prefixed by colons: temporary variables are not. Blocks are used to implement control structures, in concert with keyword messages. For example:

```
n isEven ifTrue: ['n is even!' printLine]
       False: ['n is odd!' printLine].
```

There is nothing special about the "ifTrue:False:" keyword message: unlike LISP it is not a special form or macro. Its arguments must be enclosed in blocks to avoid premature evaluation: "ifTrue:False:" will evaluate the appropriate argument block.

Iterators

Blocks can be used as mapping functions and iterators. For example, collections provide a "do:" message which applies a one-argument block to each element of the collection in turn. The total of the items in a collection can be computed by passing "do:" a block which accumulates each element:

```
total: 0.
collection do: [| :item. | total: total + item].
```

Returns

The "^" prefix operator is used to return prematurely from a method. Its semantics are essentially the same as the C `return` statement. A linear search of a collection for the string "'foo'" can be performed by combining an iterator and a return operator.

```
collection do: [| :item. | item = 'foo' ifTrue: [^true]].
^false.
```

5.4.4 Inheritance

Since SELF does not have classes, objects can inherit directly from other objects by using parent slots. A parent slot's declaration is suffixed by an asterisk "*". When a message is sent to an object, a *message lookup* algorithm is used to find a method to execute or variable to access. SELF's message lookup algorithm searches the message's receiver, then recursively searches any of the receiver's parents. If an implementation of the message cannot be found, an *undefined selector* exception is raised (§9.1.2).

```
foo = (|
        fred = ('Implemented in foo' printLine).
    |).

bar = (|
        parent* = foo.
        nigel = ('Implemented in bar' printLine).
    |).
```

For example, in the two objects above, sending "fred" or "nigel" messages to the "bar" object will execute successfully, but sending "nigel" to "foo" or "thomas" to either will result in an undefined selector exception.

Traits and Prototypes

Inheritance is often used to divide objects into two parts — a *prototype* and a *trait*. Typically the trait object contains method slots shared by all clones of the prototype, while the prototype contains the "per-instance" variables of each clone and a parent slot referring to the trait object. Prototypes roughly correspond to SMALLTALK's instances, and traits to SMALLTALK's classes [217, 43].

Multiple and Dynamic Inheritance

An object may contain more than one parent slot to provide multiple inheritance: if the method lookup algorithm finds more than one matching method, an *ambiguous lookup* exception is signalled. Parent slots may be variable slots as well as constant slots: this provides dynamic inheritance, which allows the inheritance hierarchy to change at runtime. For example, a binary tree node can be implemented using one variable parent slot but two alternative trait objects. One parent is used by empty tree nodes, and the other by tree nodes containing values. A node is created empty, and uses the empty node trait object. When a node receives a value, it alters its parent slot so that it inherits from the non-empty node trait object.

Resends

A method can use the resend operator (prefixed to a message selector) to call an inherited version of the method. In the example below, both "Implemented in bar" and "Implemented in foo" will be printed if "fred" is sent to "bar".

```
foo = (|
        fred = ('Implemented in foo' printLine).
    |).

bar = (|
        parent* = foo.
        fred = ('Implemented in bar' printLine.
                resend.fred).
    |).
```

5.4.5   The SELF Library

SELF includes a library containing over two hundred prototype objects. These are divided into various categories:

Control Structure Objects such as blocks, true and false provide messages like "ifTrue:False:" which implement basic control structures.

Numbers SELF includes both integers and floating point numbers.

Collections The largest category of SELF objects, collections are containers that hold other objects. SELF's containers include vectors and byteVectors which are fixed size arrays; sequences and orderedCollections which are like arrays but can grow or shrink to accommodate a variable number of arguments; strings which are special collections of characters; sets and dictionarys which are implemented either as hash tables or trees; doubly-linked lists; and sharedQueues which can be used to synchronise multiple processes.

Geometry Objects such as points, extents and rectangles provide basic two-dimensional geometry.

Mirrors SELF is structurally reflexive. This is provided by mirror objects, which reflect upon other objects. Each mirror is associated with one other object, the mirror's *reflectee*. Mirrors understand messages such as names, localDataSlots, and localAssignmentSlots, which respectively return the names of all slots, all data slots, and all assignment slots in the mirror's *reflectee*.

Foreign Proxies Various proxy objects provide access to functions and objects written in C or C++. Tarraingím uses proxies to provide graphical output using the X window system.

5.4.6 Example SELF Programs

Figures 5.7 and 5.8 contain a SELF version of the stack example from Section 3.6.1. This is in essence a direct translation: the data structure, algorithms, and encapsulation are unchanged.

```
"definition of stack"
traits stack = (|
    _ parent* = traits collection.

    ^ push: c = (contents at: index Put: c. index: index + 1).
    ^ pop = (index: index − 1. contents at: index).
    ^ isEmpty = (index = 0).

    ^ clone = (resend.clone contents: contents clone).
|)

stack = (|
    _ parent* = traits stack.

    _ contents ← vector copySize: 80.
    _ index ← 0.
|)
```

Figure 5.7: SELF Definition of a Stack Object

Figure 5.7 contains the definition of the stack object. This is split into two objects, traits stack containing method declarations, and the stack prototype, which inherits from traits stack. The push and pop methods are publicly exported from traits stack, while in the stack prototype, all the slots are private. Because the prototype inherits from the traits object, the methods defined in traits stack are able to access the data slots defined in the prototype. The traits object similarly inherits extra behaviour from traits collection.

```
main = (|
    _ lines ← 0.
    _ s ← stack.

    _ handleLine = (
        [eoln] whileFalse: [s push: read].
        [s isEmpty] whileFalse: [s pop write].
        lines: lines + 1).

    _ initialise = (s: stack clone).

    ^ reverse = (
        initialise.
        [eof] whileFalse: [handleLine].
        ('Reversed: ',lines,' lines\n') printLine).
|)
```

Figure 5.8: SELF Program using a Stack Object

The traits stack object provides a definition of clone to create new stacks. This uses the resend operation to call the clone operation defined in traits collection, and then clones the contents vector, which should

not be shared between different stacks. Figure 5.8 illustrates a SELF version of the stack main program. Blocks are used widely to implement the looping control structures.

Figure 5.9 shows a SELF implementation of Quicksort. This was used to generate the quicksort trace view in Figure 4.4.

```
quick = (|
        ^ quickSort =  (quickSortFrom: 0 To: size predecessor).
        _ quickSortFrom: l To: r = ( | i. j. x. |
            i: l.
            j: r.
            x: at: (l + r) / 2.
              [
                [(at: i) < x] whileTrue: [i: i successor].
                [x < (at: j)] whileTrue: [j: j predecessor].
                i ≤ j ifTrue: [
                  i = j ifFalse: [swap: i And: j].
                  i: i successor.
                  j: j predecessor.
                ].
              ] untilFalse: [ i ≤ j ].

            l < j ifTrue: [quickSortFrom: l To: j].
            i < r ifTrue: [quickSortFrom: i To: r].
            self).
|)
```

Figure 5.9: Quicksort in SELF

19. A language that doesn't affect the way you think about programming,
is not worth knowing.

Alan Perlis, *Epigrams On Programming* [168]

6

Display Subsystem

The display subsystem centres around view objects. As the main component of the subsystem, views are responsible for implementing all the subsystem's requirements: displaying graphics, handling the user interface, and accepting input to particular visualisations.

To produce a visualisation with Tarraingím, the visualiser must either write a view from scratch, or obtain (and modify if necessary) a view from Tarraingím's view library. This chapter describes the implementation of views from the visualiser's perspective. Section 6.1 presents an overview of the view objects, and describes their position within the wider system. Sections 6.2 to 6.6 show the construction of several example views. Section 6.7 concludes the chapter with an outline of the contents of Tarraingím's view library.

6.1 Views

Figure 6.1 shows the structure of the view inheritance hierarchy. The view object is the abstract view in Tarraingím's pipeline (see Figure 5.2) and inherits from eventClient and tgimObject (§5.3.4). Because they inherit from eventClient, all views can use the client event protocol to handle events they receive from watchers (§8.3.6). Concrete view objects inherit from view to implement displays of particular objects in the program. Tarraingím provides a library of concrete view objects (§6.7) and new concrete views can be written by visualisers.

The basic view object supplies a *public* protocol which provides an external interface to views, and a *private* protocol which is used to structure the view's implementation. Table 6.1 contains the essential messages of these protocols. Private messages are placeholders for view-specific tasks, and are called by public messages when necessary. Each concrete view object implements these messages in a manner appropriate to its type. For example, a bar chart view would handle the private drawModel method by drawing a bar chart, while a scatter plot view would draw a scatter plot. The visualiser creates a new type of view by implementing the private messages to perform the behaviour required by the new view.

6.1.1 Views in Context

A view must communicate with other objects. Figure 6.2 shows a view in the context of the objects with which it collaborates.
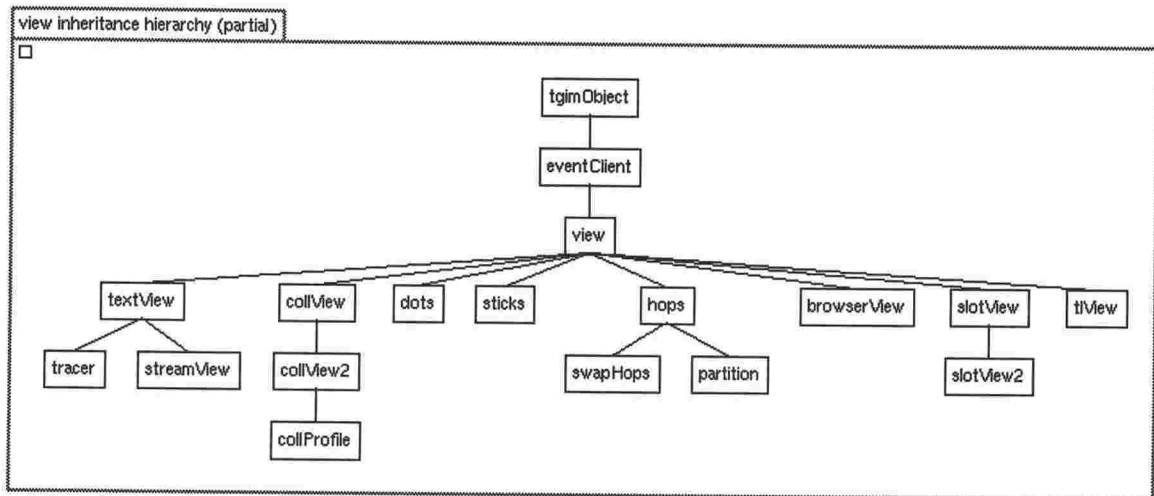
Figure 6.1: View Inheritance Hierarchy

view protocol

Public
> watch: object — Instructs the view to display object. This sets the view's aim to that object.
>
> copyWatch: object — Creates a new copy of the view which is then sent watch: object.
>
> aim — A view's aim is the object it is requested to visualise.
>
> model — A view's model is the object it is displaying.

Private
> initialise — This is sent when the view is initialised.
>
> drawModel — This is sent when the view needs to be redrawn. Particular views should implement this by redrawing their display. This is sent when the view is first displayed, and whenever the view receives a change event from its watcher which the view does not otherwise handle.
>
> callback — Sends a query callback to the view's model.
>
> command — Sends a command callback to the view's model.

View Hierarchy
> subViews — This returns a collection of the view's subviews.
>
> addSubView: v — This adds a new subview v to the view.
>
> removeSubView: v — This removes subview v from the view.
>
> *... and events dispatched using the client event protocol*

Table 6.1: Basic view protocol.

View Tree

A view can belong to a tree of views, with a single superView and several subviews. The abstract view object includes behaviour to manage this tree dynamically. A view can add, delete, or rearrange its subviews in response to changes received from the target program or user interaction.

View Parameters

The display produced by a view can be customised by variables within the view object. These variables are the view's *parameters*. They allow the user to tailor the view's appearance as the system is running.
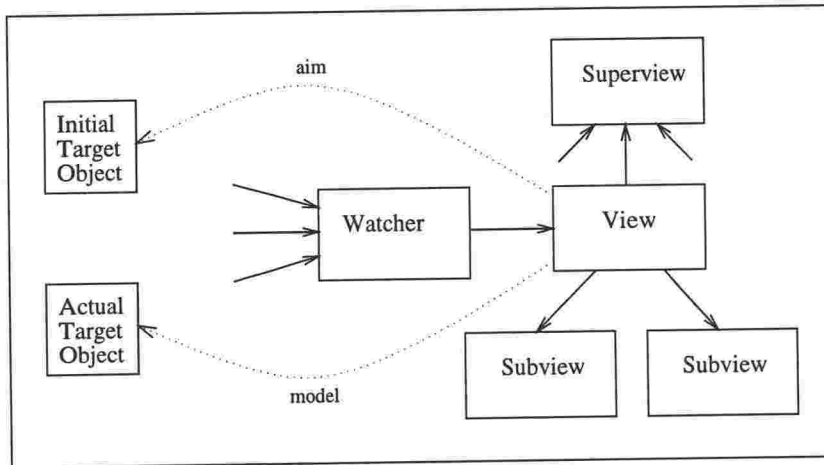
Figure 6.2: A View in Context

For example, a view's colour scheme, font, graphics scale, or the type of its subviews may be set by its parameters.

6.1.2 Watchers

A view is responsible for drawing the visualisation of an object, but does not itself monitor that object. This is handled by its associated **watcher**, which links the view to the object it is displaying.

Target Objects

A watcher links a view to two objects in the target program, the aim and the model. A view's aim is the object the user has requested be displayed. Views send their aim to their watcher. The watcher computes the model, which it then sends to the view. A view's model is the object the view will actually display, and is usually the same as the view's aim.

The distinction between aim and model is particularly useful for views displaying aggregate abstractions (§7.4.1). For example, a bar graph view can be used to display an invocation profile of a particular object. The view's aim is the object being profiled, and the model is the object holding the actual profile database. Informally, a view's or watcher's *target objects* are all the objects in the program upon which the view depends.

Events

Watchers send views event notifications describing changes of interest to the view. By default, views simply clear and redraw their display upon receiving an event. Specific events or event categories can be handled using the client event protocol, inherited from traits eventClient (§8.3.6).

Views themselves use events to send callbacks to their models. They generate these events by sending messages to callback for *query* callbacks which do not alter the state of the target object, or command for *command* callbacks which may change the target object. These events are directed through the view's watcher.

Section 8.3 discusses events in more detail.

6.1.3   Navel

The mechanics of input and output are handled by the NAVEL[1] graphics library, which we have written to provide a layer over the **X** window system [189]. NAVEL creates and manages an **X** window hierarchy which parallels Tarraingím's view tree. It provides messages to views for calling **X** graphics functions and handling **X** events (see Table 6.2).

---

view protocol

Accessing
    area   Returns the area of the view's window as a rectangle.

Graphical Output
    drawAt: p String: s   Draw the string s at position p, in the foreground colour.
    drawRect: r   Draw the outline of rectangle r, in the foreground colour.
    fillRect: r   Paint rectangle r solidly, in the foreground colour.
    fillRect: r Colour: c   Paint rectangle r solidly, in colour c.
    clear   Clear the window to the background colour.

Input Events
    leftUp: pt   The left mouse button has been released at pt.
    middleUp: pt   The middle mouse button has been released at pt.
    rightUp: pt   The right mouse button has been released at pt.
    key: k   Key k was pressed.
    expose   The window has been exposed, and needs to be redrawn.

Table 6.2: NAVEL graphics protocol used in examples.

A NAVEL view receives two kinds of events — those from the **X** window system describing user actions and requests to refresh the view, and those from its watcher. These two event streams are unrelated and are handled separately by the view. Receiving an **X** event, for example a notification of a mouse press, may cause the view to send another, perhaps a callback to the target program, but this is under the control of the view.

In the remainder of this thesis, *event* refers to the events exchanged between views, watchers, and controllers, rather than the **X** window system events associated with input/output.

6.2   A Simple View

Figure 6.3 shows a display produced by a simple view — a cubist picture of a trafficLight object. A trafficLight object is implemented by a single variable slot holding the trafficLight's colour, and a few simple messages (see Figure 6.4). The view is implemented by the tlView object (see Figure 6.5), which inherits from the abstract view object. Since this view is so simple, the only behaviour required in tlView is to draw the picture. To do this, the view implements the private drawModel message, which is received by a view when the view is created, needs to be repainted, or is notified of changes within its model.

The trafficLight object is very simple: it is essentially a single variable. The tlView object depends upon the local state of a trafficLight, so it needs to be informed whenever the local state changes (§4.3.4). This monitoring strategy is implemented by a localWatcher (§7.2.4), installed as the tlView's watcher. Any assignment actions to the target trafficLight object's local state (i.e. its colour variable slot) will be reported as changes to the view.

The graphics for the view consist of three rectangles, each representing one aspect of the trafficLight, and are drawn by the drawModel method. First, various local variables (w, h, top etc.) are used to

---

[1]NAVEL — a window system for looking at your SELF.
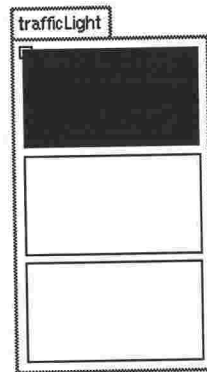
Figure 6.3: trafficLight View

```
traits trafficLight = (| "shared behaviour for trafficLights"
   _ parent* = traits clonable.

   ^ printString = ('a ', colour, ' traffic light').
   ^ copy = (resend.copy red).  "new trafficLights are always red"

   ^ red = (colour: 'red').
   ^ amber = (colour: 'amber').
   ^ green = (colour: 'green').

   ^ cycle = (red. amber. green. red).

   ^ isRed = (colour = 'red').
   ^ isAmber = (colour = 'amber').
   ^ isGreen = (colour = 'green').
|)

trafficLight = (|
   ^ parent* = traits applications trafficLight.
   _ thisObjectPrints = true.
   ^ colour ← 'red'. "local state"
|)
```

Figure 6.4: trafficLight Object

calculate the sizes and positions of the rectangles used in the display. The drawModel method uses the NAVEL message area, which returns the size of the X window displaying the view, so that the view will always fill the X window.

The rectangles representing the trafficLight's aspects are then drawn. The rectangle representing the illuminated aspect is drawn filled, while the other two are outlined. The tlView must determine which aspect is illuminated. To get this information, it sends callbacks (the isRed, isAmber, and isGreen messages) to its model. The callbacks are sent as messages to callback, which will create the appropriate callbackEvent and forward it to the view's watcher. This will eventually send the message to the target trafficLight in such a way that it will not be detected by any other views upon the same object (§8.2.3). The callback for the illuminated aspect will return true, and thus the rectangle corresponding to the active aspect will be filled.

```
traits tlView = (|
  _ parent* = traits view.
  _ drawModel = (|w. h. top. mid. bot. m = 5|

    "calculate sizes"
    w: area width − m double.
    h: (area height − (m * 4)) / 3.
    top: ((m@m)##(w@@h)).
    mid: ((m@(m double + h))##(w@@h)).
    bot: ((m@((m * 3) + h double))##(w@@h)).

    "draw rectangles"
    callback isRed ifTrue: [fillRect: top] False: [drawRect: top].
    callback isAmber ifTrue:  [fillRect: mid] False: [drawRect: mid].
    callback isGreen ifTrue: [fillRect: bot] False: [drawRect: bot].
    self).
|)

tlView = (|
  _ parent* = traits tlView.

  "install watcher"
  _ watcher ← localWatcher.
|)
```

Figure 6.5: trafficLight View Implementation

6.3 Generic Views

The trafficLight view above is specific to one type of object — a trafficLight. It is possible to design views which can visualise many different kinds of objects. These *generic* views interact with their target objects using only an abstract interface (§3.6.2). For example, a generic collection view can display a list, hash table, array, tree, string, or priority queue, since all these objects conform to the collection protocol — that is, they are different ways of implementing a collection.

Two instances of a simple generic view (the dots scatter plot sequence view from Figure 3.1) are illustrated in Figure 6.6, and the dots view implementation is given in Figure 6.7. Each dot corresponds to an element in a sequence of integers. The vertical position of a dot represents the value of the element, and its horizontal position corresponds to the element's position in the sequence.

The dots view's display is drawn in the drawModel method, which iterates over the view's model and draws a dot for each element. The view accesses its model's elements by sending a query callback (the "do:" message) to its model, passing a block which is then called once for each element in the model collection.

This view is generic because the interface it uses to communicate with its model is abstract. It sends only one callback, the message do:, and can visualise any object which implements this message with the required semantics — mapping a block over the object's elements. This message is part of the collection protocol, and is implemented by all collection objects. A dots view can display any type of collection object containing numbers (the two views in Figure 6.6 illustrate a vector and a list respectively).

The scale and size of dots in the display are governed by view parameters. The variable dotSize controls the size of the dots, while dotWidth and dotHeight control x-axis and y-axis scales respectively. By changing the parameters, different visualisations can be produced from one view definition. The two views in Figure 6.6 have different settings for their parameters.
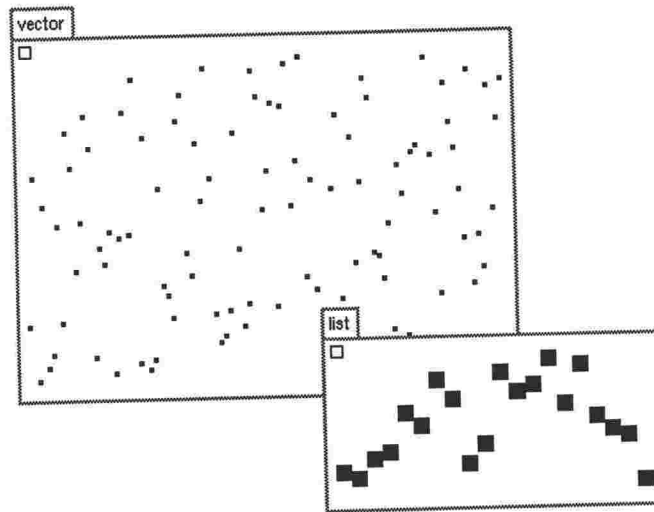
Figure 6.6: Two dots Views

```
traits dots = (|
  _ parent* = traits view.
  _ drawModel = (
    x: leftMargin.
    callback do: [ |:val. y. |
      y: (baseline − (dotHeight * val)).
      fillRect: ((x@y)##(dotSize@@dotSize)).
      x: x + dotWidth].
    self).

|)

dots = (|
  _ parent* = traits dots.

  "view parameters"
  _ dotSize ← 5.
  _ dotHeight ← 2.
  _ dotWidth ← 4.
|)
```

Figure 6.7: dots View Implementation

6.4 Dynamic Updating

The dots and trafficLight views presented above are batch views, as they update their entire display whenever they are notified that their target object has changed (§3.7.3). Any current output is cleared and their drawModel method is called to refresh the display.

This is expensive. The view's entire output must be redrawn even if only a small portion has changed. It is impossible to introduce any animation to draw smooth transitions between states without first determining in which way the target object has changed. In order to animate their output, or update their displays incrementally, views can inspect the received event and take specialised action, rather than recreating the entire picture from scratch.

Figure 6.8 shows an event handling method which provides incremental refresh for the dots view of Figure 6.7. Whenever an element of the visualised collection[2] is changed, this method directly updates the dot in the view representing that element.

The changeEvent: e message is sent to a view when the view receives a changeEvent from its watcher (§8.3.6). These events are sent to the view whenever an action occurs in the view's model which the view's watcher considers a change. When an event arrives, it is checked to see if is an event that the view can handle incrementally — in this case, any event caused by an at:Put message. If not, the event is resent to be handled by the view's default behaviour.

```
_ changeEvent: e = (| elem. old. new. x. y. |

    "determine if this message is suitable for dynamic update"
    ('at:Put:' ≠ e name)
      ifTrue: [^resend.changeEvent: e].

    elem: e at: 0.    "get event parameters"
    new: e at: 1.

    old: callback at: elem. "and old value"

  "erase old dot"
    x: leftMargin + (a * dotWidth).
    y: (baseline − (dotHeight * old)).
    fillRect: ((x@y)##(dotSize@@dotSize))
      Colour: background.

  "and draw new"
    y: (baseline − (dotHeight * new)).
    fillRect: ((x@y)##(dotSize@@dotSize))

    self).
```

Figure 6.8: Dynamic Updating of a dots View

The parameters of the event are obtained from the event object. The position of the element that has changed (the first argument of the at:Put message received by the view's model) is assigned to the elem variable, and the new value (the second argument of the message) is assigned to new. The variable old is set to the old value of the element at that position, retrieved by the at callback. The current dot (representing the old element's value) is then erased, and a new dot drawn. The view is thus updated to reflect the model, with a minimum of effort. Note that because the target object is queried to determine the previous value to erase, the change event must be received *before* the target object actually executes the at:Put message (§4.3.3). Such a strategy is provided by a preLocalWatcher, as compared to the localWatcher, which sends changes to its view after the target object has changed (§7.2.4).

The same technique is used to produce an animated view, although rather than simply erasing the old parts of the display and drawing the new, the visualiser must draw as many frames as required to produce an illusion of smooth motion.

---

[2]To simplify the presentation, the method in Figure 6.8 is only applicable to views of indexable collections, such as vectors, where each element is accessed by an integer index.

6.5   Hierarchical Views

Tarraingím views can be arranged into a tree, allowing a view to include one or more subviews (§5.3.3). This feature is common in window and user interface systems, but rare in program visualisation — only a few graphical debuggers, such as CERNO-II and GELO (§2.2.1), provide flexible hierarchical views. We have found hierarchical views useful for the following reasons:

- Complex views can be factored into component parts, and then each part can be written and tested independently.

- The user can build composite views of particular objects by grouping several preexisting views.

- A hierarchical view can be used with different types of subviews.

- A view can be reused as a subview of several different types of hierarchical views.

Views can be used as subviews without any extra effort on the part of the visualiser. The decision to use subviews (rather than simply one larger view) must be made when a view is being designed. One or more of a hierarchical view's parameters will be view prototypes. When the view is displayed, the prototypes are cloned to produce the required subviews.

6.5.1   Browser Views

As an example, Figure 6.9 displays two versions of Tarraingím's browserView. Each element in the browser is displayed by a subview. The two browsers pictured are the same type, but each is parameterised by different subviews. The implementation of a browserView is outlined in Figure 6.10.
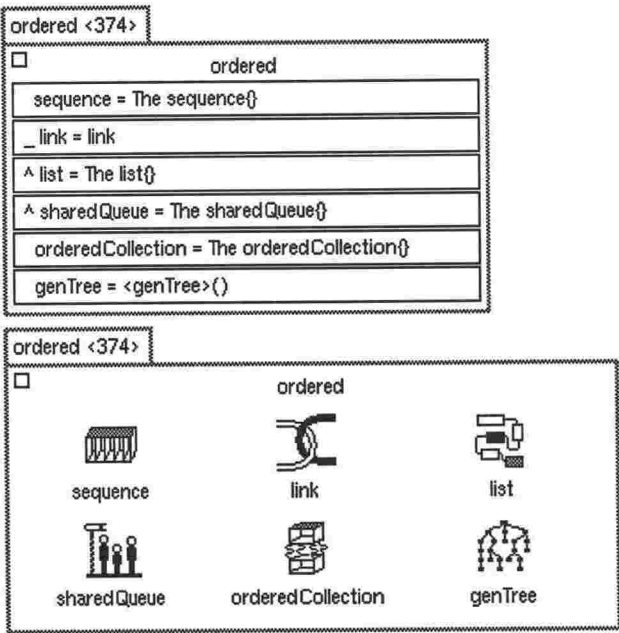
Figure 6.9: Two Browser Views

A browser view is first drawn when it receives a **drawModel** message. This creates the required subviews using the **copyWatch** message to clone the prototype subview (the **browserView**'s **elemProto** parameter) and configure each subview to display one of the **model**'s elements. Once the subviews have been created, they are laid out within the main view (by the **layoutSubViews** message, not shown in Figure 6.10), and a title specified by another parameter is drawn.

```
traits browserView = (|
  _ parent* = traits view.

  _ drawModel = (
    ((subViews isEmpty) && [callback isEmpty not])
      ifTrue: [makeSubViews].

    "draw title"
    drawAt: 10@10 String: title.
    self).

  _ makeSubViews = ( |tmp|
    callback do: [| :val |
      addSubView: (elemProto copyWatch: val)].
    layoutSubViews).
|)

tgim browserView = (|
  _ parent* = traits browserView.

  "view parameters"
  _ elemProto ← slotView.
  _ title ← 'Browser'.

  _ watcher ← localWatcher.
|)
```

Figure 6.10: browserView Implementation

Note that if the browserView is redrawn (i.e., drawModel is called again) the title will be redrawn, but the subviews are not recreated, as they exist in their own right. They are not redrawn by the browserView because they are fully independent — they are redrawn by their own definitions. Each subview has its own watcher, and so receives and handles changes regarding its model independently of both the superview and any other subviews (see Figure 5.4).

6.5.2 Structural Constraints

If the superview's model changes, the arrangement of subviews may need to change also. This is a structural constraint: the structure of the view tree must parallel the structure of the target objects. When the model adds or deletes elements, the superview must add or delete the subviews visualising those elements.

Figure 6.11 shows how Tarraingím handles structural constraints. The browserView can include a changeEvent method which is received when a change is detected in the view's model. When the browser receives this message, it reassesses its subviews, comparing their structure to the changed structure of its model, and reorganises itself appropriately.

Tarraingím uses auxiliary adjuster objects to manage structural reorganisation. An adjuster compares two lists and executes a series of editing actions to transform one into a parallel of the other. Similar facilities are used in UNIDRAW [221] and GINA [19]. The adjuster compares the current subview's aims to the new requirements, and adds or removes subviews so every element of the view's model is displayed by a subview. Once the structure has been altered, the subviews' layout within the main view is recalculated.

The changeEvent method sends an "adjust:Keys" message to an adjuster to actually make the necessary changes. This method's arguments are: the current list of subviews; a block for determining a views's

```
_ changeEvent: e = (

    "ensure subViews correspond to the model's contents"
    adjuster adjust: subViews
        Keys: [|:sv| sv aim]
        To: callback
        Create: [|:aim| addSubView: (elemProto copyWatch: aim)]
        Keep: []
        Destroy: [|:view| removeSubView: view].

    "recompute layout"
    layoutSubViews).
```

Figure 6.11: Structural Constraint for a browserView

aim; a list of the model's current contents; and three action blocks which are called as necessary to create a new subview, keep a subview intact, or destroy a subview. The "adjust:Keys" method checks each subview on the list in turn, using the "Keys" argument block to determine the subview's aim. It sends a "callback" to retrieve the browser's model, and compares the subviews's aims with this callback. Finally, it uses the "Create", "Keep" and "Destroy" argument blocks to update the subviews as necessary.

6.6 User Interaction

A program exploratorium must be a dynamic and interactive environment. Users must be able to control what is visualised and how it is displayed, and interact with the visualised program. The display subsystem provides support for user interaction. Views are able to receive information about a user's commands and either handle them internally or call upon the rest of the system. This section describes how Tarraingím supports the three main user interface tasks — selecting the objects to be visualised, sending commands to those objects, and customising the visualisation. Section 10.1 presents Tarraingím's user interface from the user's perspective.

6.6.1 View Navigation

Tarraingím's users choose objects to be displayed by navigating around the target program. Tarraingím starts by displaying the lobby (the root of the SELF name space [217]) with a browser like those illustrated in Figure 6.9. When one of the slots in a browser is selected with the mouse, a new browser is created for the object contained in that slot. All objects in the program are reachable from the lobby, so any object in the program can be located in this way — the effect is similar to the Macintosh Finder [224]. If the title of a browser view is selected, a view palette appears, inviting the user to open another view.

```
    "create a new browser for this object"
            ^ leftUp: pt = (browserView copyWatch: model).

    "pop up a menu"
            ^ rightUp: pt = (viewMenu copyWatch: model).
```

Figure 6.12: View Navigation

As views can be created dynamically, navigation can be implemented easily in Tarraingím, as shown in Figure 6.12. The leftUp message (sent by the left mouse button) creates a new browser for the object; the rightUp message pops up a menu of different views of the target object.

These messages are defined by the abstract view object, and so are inherited by other views by default.

6.6.2 User Input

Event handling and command callbacks can be used to allow the user to interact with the underlying objects in the program. For example, the dots view of Figure 6.7 can be extended to allow the user to change values in the view's model, by adding the method shown in Figure 6.13. When the user clicks on the view with the middle mouse button, the model is updated so the element under the mouse is altered.

```
"change a collection element"
   ^ middleUp: pt = (|min. max. x. y|

      "scale co-ordinates of input point"
      x: (pt x − leftMargin) /− dotWidth.
      y: (((baseline − pt y) / dotHeight)) asInteger.

      "if within bounds, perform assignment"
      (x isBetween: 0 And: callback size)
                      ifTrue: [command at: x Put: y].
      self).
```

Figure 6.13: User Input

The middleUp message is received whenever the user clicks the middle button in the view — its argument is a point representing the location of the mouse click. The point at which the mouse click occurred is scaled to match the view, and, if the point refers to a legitimate collection element, the model is updated by sending a command callback. The callback message at:Put changes the element positioned

Textual Collection Views Collections are the largest category of objects in the SELF library (§5.4.5). We have therefore built several textual and graphical views of collections. These views can generally be applied to any particular collection implementation. For example, the vector view in Figure 3.1 is a basic collView, showing the collection's elements. The views of dictionarys in Section 10.2.3 are displayed by coll2View views, which show both the keys and values of the collection's elements. Coupled with suitable watchers, textual and graphical collection views can be used to display aggregate abstractions, such as the profiles in Figure 3.4.

Graphical Collection Views The library also contains graphical views of collections. These are generally restricted to collections indexed by integers and containing numbers, for example, the dots and sticks views from Figure 3.1. Other graphical collection views include the lpView, which displays linked-list as a sequence of boxes and arrows (see Figure 10.39). A horizPointer view provides a pointer to an element in another collection view. This view is used to display the index component in the stack abstraction view in Figure 3.3.

Tree Views The nTreeView provides a generic view of an n-ary tree. This view can produce many different displays, depending upon the watcher and subview prototypes it is configured with. For example, the call tree view of Figure 3.2, the structure tree view of Figure 3.4, the parse tree view of Figure 10.3, and the view structure views described in Section 10.2.1, are all different versions of the basic nTreeView.

User Interface Views Tarraingím's user interface consists of various browser and inspector views that display an object as a list of slots (see Figure 6.9) . These are implemented by browserViews, which can be parameterised with different slotView prototypes to produce textual and iconic browsers. A browserView variant is also used to produce Tarraingím's menus and property sheet views.

Miscellaneous Views We have also built several custom views of various objects. Figure 10.35, for example, shows an fsmView of a finite state machine, which uses fsmStateViews to display individual states. Other custom views are illustrated in Figure 3.2, which includes a partition view of quicksort; Figure 3.3, which includes the stack implementation view combining several independent subviews, Figure 6.3, which shows a cubist tlView of a traffic light; and Figure 10.14, which shows a view of a controller's dispatch database (§8.1.3).

39. Re graphics: A picture is worth 10K words - but only those to describe the picture. Hardly any sets of 10K words can be adequately described with pictures.

Alan Perlis, *Epigrams On Programming* [168]

7

Strategy Subsystem

The strategy subsystem implements the mapping component of the APMV model (§3.7). This subsystem links the target program to the display, and determines how the target program is to be monitored. Each individual display (implemented by a view) is linked by a strategy (implemented by one or more watchers) to its target objects. The particular strategy chosen will depend upon the view, the object to be visualised, and the user's preference.

The remainder of this chapter describes the strategy subsystem. Section 7.1 reviews the subsystem's place in the framework as a whole, describes the main watcher protocol, and outlines the main categories of watchers. Sections 7.2 to 7.5 describe each category of watcher in turn, and Section 7.6 concludes the chapter with a discussion of Tarraingím's watcher library.

7.1 Watchers

A watcher embodies a strategy for connecting a view to an object within the program. Tarraingím includes a library of general purpose watchers, and specialised watchers can be written by the visualiser in SELF. Like a view, aspects of a watcher's operation can be altered by changing its parameters.

A tree of watchers can be used to implement a complex strategy, in the same way a hierarchical view can implement a complex display (see §5.3.3 and Figure 5.5). A *superwatcher* can use several *subwatchers* to assist it in monitoring its target object. Subwatchers are specified as parameters to their superwatchers, in the same way as the parameters of hierarchical views (§6.5). The precise use to which a subwatcher is put will depend upon the type of watcher that is parameterised. The root of a watcher tree is a view that ultimately receives changes from the watchers in the tree.

Tarraingím's users are not aware of the distinction between watchers and views. Instead, users think in terms of combinations of watchers and views presenting particular visualisations. This is because a watcher can completely change the information displayed by a view. For example, a collection view can display either the contents of a collection object in the target program, or a profile of the collection object's execution, depending upon the type of watcher used with the collection view.

7.1.1 Types of Watchers

The structure of Tarraingím's library of watchers is organised by inheritance, as illustrated in Figure 7.1. All watchers are Tarraingím objects and can handle events, so they inherit from tgimObject and eventClient

(§5.3.4). All watchers inherit from the abstract **watcher** traits, and are grouped into categories depending upon the number of subwatchers they use: *leaf* watchers use no subwatchers; *filter* watchers use one subwatcher; *indirect* watchers use two subwatchers; and *multiple* watchers use any number of subwatchers. Each category has an abstract watcher from which its concrete watchers inherit. This allows code common to all watchers to be inherited from the abstract **watcher** traits, and commonalities between watchers in each category can be inherited from the abstract category watchers.
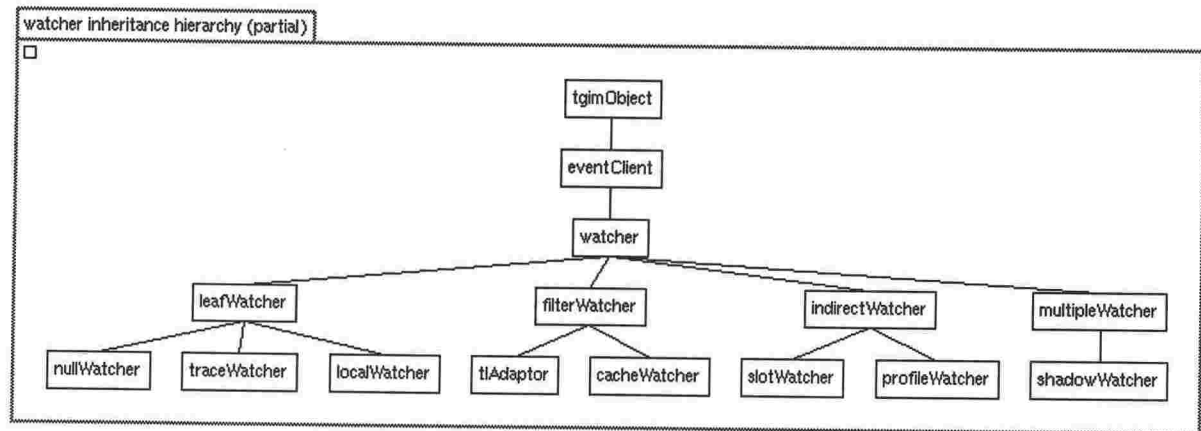
Figure 7.1: Watcher Inheritance Hierarchy

Leaf Watchers

Leaf watchers form the leaves of the watcher tree, and do not use any subwatchers. If a view uses a simple strategy implemented by only one watcher, that watcher must be a leaf watcher. Most leaf watchers use controllers to interface with the monitoring component to gather execution information about the target program — for example, traceWatchers, which monitor all the actions of the target object (§4.3), or localWatchers, which detect changes in an object's local state (§4.3.4). Leaf watchers are the only type of watchers which communicate with controllers. If another type of watcher needs to use a controller, it uses a leaf watcher as a subwatcher. Leaf watchers are described further in Section 7.2.

Filter Watchers

Filter watchers use one subwatcher. A filter watcher acts as a wrapper around its subwatcher, intercepting and modifying its event traffic. For example, a tlAdaptor watcher alters the callbacks and changes passing through it so that a tlView (§6.2) can be linked to a vector. A cacheWatcher filters changes to remove duplicates (§4.3.4). Filter watchers are discussed further in Section 7.3.

Indirect Watchers

Indirect watchers redirect their model, so that it is not the same as their aim. They use two subwatchers, one to monitor their aim (the object the user requests the view displays) , and one to monitor their model (the object the view actually displays). Indirect watchers can be used to specify a view's model as an indirect reference from its aim, which is useful in implementation views displaying an object's components (such as the stack implementation view in Figure 3.3). They can also be used to build aggregate abstractions, when the model is a new object which is updated by the watcher. For example, the profile views in Figure 3.4 are implemented by indirect watchers which create and maintain profile database objects. Indirect watchers are described further in Section 7.4.

Multiple Watchers

Multiple watchers use any number of subwatchers. They take a single watcher parameter which is cloned as many times as needed to create the subwatchers. Multiple watchers can be used for purposes such as managing the effects of aliasing (§4.4.2) which need low-grade information about a large number of objects. Multiple watchers are discussed further in Section 7.5.

7.1.2 Watcher Attachment

A watcher is always in one of three states — it is either *attached*, *provisionally attached*, or *detached*. A watcher is attached if it is part of a watcher tree which is currently monitoring the program. It will handle events passed up from controls or subwatchers, and passed down from its view or superwatchers. If a watcher is attached or provisionally attached, all watchers between it and the root of the watcher tree must also be either attached or provisionally attached, since they will receive events passed up the watcher tree.

A detached watcher is not monitoring the target program. It may be part of a watcher tree, although all subwatchers of a detached watcher must be detached. Note that an attached watcher may have several subwatchers which are not attached, and in a watcher tree there may be several detached subtrees.

A watcher must be attached explicitly. A view's watcher tree is attached when the view is initialised, and subwatchers are generally attached by their superwatchers. Attaching a watcher is not necessarily successful. If a watcher is not compatible with its target object, or the monitoring system is unable to monitor it, the watcher will not become attached. If it may be able to become fully attached in the future it will become provisionally attached, otherwise it will remain detached. A provisionally attached watcher receives events, and operates in the watcher tree like an attached watcher — in particular, it may have fully attached subwatchers. If a watcher is provisionally attached, it presumably cannot gather all the information required by its associated view, so its view will not present a display. Because a watcher's target objects may change as the result of the events the watcher receives, an attached or provisionally attached watcher's state may change at any time.

Behaviour to maintain these constraints upon watcher attachment is inherited by all watchers from the abstract watcher object.

7.1.3 Watcher Interface

The watcher protocol is divided into four categories: *accessing*, *down*, *up*, and *private*. We discuss each category in turn.

Accessing Protocol

Accessing methods (see Table 7.1) retrieve information about a watcher's state: its aim, model, and whether the watcher is attached. These messages are implemented in the abstract watcher object.

watcher protocol
_____

Accessing
aim	Returns the watcher's aim.
model	Returns the watcher's model.
attached	Returns true if the watcher is attached.

Table 7.1: Watcher accessing protocol

Down Protocol

The *down* protocol messages (see Table 7.2) are sent down the watcher tree — from a view to its watcher, or a watcher to its subwatchers. They are used to attach and detach watchers, to establish the topology of the watcher tree, and to control event routeing (§7.1.4). These messages are implemented in the abstract watcher object, and call private messages to perform type specific tasks (see Table 7.4). Watchers also use messages in the *client event protocol* inherited from eventClient (see Table 8.7) to dispatch *down events* down the watcher tree (§8.3.6).

watcher protocol
_____

Down

attach	Requests the watcher to attach itself. This implies that the watcher's view or superwatcher is ready to receive events.
detach	Detaches the watcher and any subwatchers.
watch: aim	Sets the watcher's aim, and attaches the watcher.
up: up	Sets the watcher's superwatcher and upEvents pointer to up.
upEvents: upEvents	Sets the watcher's upEvents pointer.

   ... *and down events dispatched using the client event protocol*

Table 7.2: Watcher down protocol

Up Protocol

The watcher *up* protocol consists of messages sent up the watcher tree from watchers to views, or subwatchers to superwatchers (see Table 7.3). This protocol comprises the sub:Model, sub:Warning and sub:Error messages describing the current state of a particular subwatcher, and messages from the client event protocol used to dispatch events up to views. The sub...messages are sent by a subwatcher to its immediate parent whenever the subwatcher's state changes. The first argument of these messages is the subwatcher that is sending the message. Default versions of these messages are implemented in the abstract watcher object.

watcher protocol
_____

Up

sub: sw Model: m	Subwatcher sw has acquired m as its model. It is now fully attached.
sub: sw Object: obj Warning: string	
	Subwatcher sw has found an anomalous situation (described in string) when monitoring the object obj. This object is not necessarily sw's model. sw is now provisionally attached.
sub: sw Object: obj Error: string	
	Subwatcher sw is unable to watch its model. The error relates to the object obj and is described in string. As a result, sw is no longer attached.

   ... *and up events dispatched using the client event protocol*

Table 7.3: Watcher up protocol

A watcher is typically attached using the down protocol watch message. This is sent by a view to its associated watcher when the view is being initialised, and gives the watcher its aim. When the watcher is successfully attached, it sends the sub:Model message back to its superwatcher (or view). The m argument of the sub:Model message is used to set the superwatcher's model. In this way, the user chooses a view's aim, but the view's watcher tree determines the view's model. The aim is propagated down the watcher tree, while the model is propagated up the tree. This mechanism is used by indirect watchers to visualise aggregate abstractions and implementation component views (§7.4).

Private Protocol

The abstract **watcher** object defines private messages which can be overridden in a concrete watcher to implement that watcher's type-specific behaviour. These messages are sent appropriately by the methods implementing the up and down protocol messages. The private messages are listed in Table 7.4.

watcher protocol

Private
  localAttach   This is sent when a watcher is being attached to allow it to initialise itself.
  localDetach   This is sent when a watcher is being detached to allow it to release resources.

Table 7.4: Watcher private protocol

To implement a new kind of watcher, the visualiser typically defines localAttach to customise the watcher's initialisation. A default definition of localDetach is provided in the abstract **watcher** object which simply detaches all subwatchers, so localDetach does not have to be defined unless localAttach allocates other permanent resources. To adjust the new watcher's handling of callbacks and changes, one or more event handling messages from the client event protocol can be implemented. The visualiser can also implement watcher up protocol messages, to handle state messages from subwatchers.

7.1.4 Message and Event Routeing

Messages and events are generally sent only one step in the appropriate direction in the watcher tree. A watcher or view sends messages down to its direct subwatchers, and subwatchers send messages up to their immediate parent. The one exception to this rule concerns up events travelling up the tree. Watchers include an **upEvents** pointer which is used to direct events to one of the watcher's ancestors, rather than to its immediate parent.

The **upEvents** pointer is provided for two reasons. First, up events comprise most of the traffic within the watcher tree, and many simple watchers forward all the up events they receive to their superwatcher. Routeing these events directly avoids the cost of unnecessary event dispatches. Second, events are received by a watcher using the client event protocol (§8.3.6). Unlike the **sub**... messages, the client event protocol messages are not tagged by the subwatcher from which they have arrived[1]. A watcher with more than one subwatcher cannot determine from which subwatcher events have arrived, so events received from all subwatchers must be treated similarly. In some situations, events from different subwatchers need to be processed differently. The division is often between two sets of subwatchers: some generating events to be processed by the watcher, others generating events to be handled further up the watcher tree. This can be arranged by setting each subwatcher's upEvent appropriately. The upEvent pointer of the top watcher in a watcher tree is set to point to that subwatcher's view, and the other watchers' upEvent pointers are set to the same as their parent watcher's pointer, unless they need to intercept up events.

Figure 7.2 illustrates routeing within a watcher tree. Each watcher is directly connected to its neighbouring subwatchers and superwatchers. The **attach**, **watch** and **sub**... messages are sent along these links. These messages transmit the view's aim to the watchers, and return the model to the view. The watchers' upEvents pointers are used to route events up from subwatchers to superwatchers, while the view sends callback events down the watcher tree.

7.2 Leaf Watchers

This section describes several example leaf watchers. Most leaf watchers monitor their target object, so the abstract leaf category object contains default behaviour for attaching and detaching monitoring using controllers (§8.1.2). Leaf watchers typically implement the localAttach message by computing a

---

[1]The **watcher** up protocol was designed some time after the client event protocol.
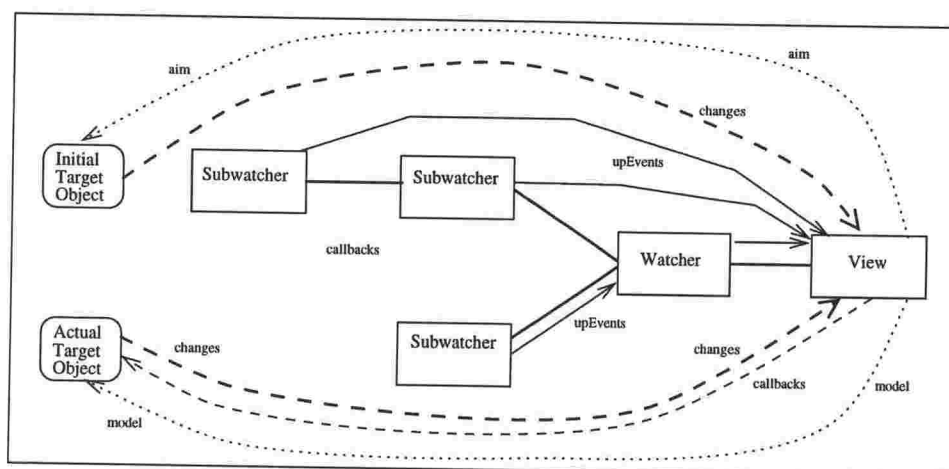
Figure 7.2: Watcher event and Message Routeing

monitoring plan and sending the plan to their model's controller. Leaf watchers are associated with only one object in the target program, so their model is the same as their aim.

Section 10.2.1 presents several illustrated examples of the use of leaf watchers.

7.2.1 Null Strategies

The simplest monitoring strategy is the null strategy, which ignores its target objects (§4.3.4). Tarraingím's nullWatcher implements this strategy. A nullWatcher is effectively a placeholder, as it makes no attempt to monitor its model. A nullWatcher is passed as a parameter for other watchers and views when no actual monitoring is required.

7.2.2 Monitoring an Object's Actions

The most basic local strategy is to monitor all events occurring within a particular object generated by the target program (§4.3). This strategy is implemented by a traceWatcher, shown in Figure 7.3.

The traceWatcher defines the localAttach message to register the traceWatcher with its model's controller. The control variable used in Figure 7.3 is initialised by leaf watchers' default behaviour to refer to their model's controller. The add message sent to control is part of the controller registering protocol, described in full in Section 8.1.2.

```
traits traceWatcher = (|
    _ parent* = traits leafWatcher.
    _ localAttach = (control add: self).
|)
```

Figure 7.3: Trace Watcher Implementation

When the traceWatcher is attached, the localAttach method is executed, and this causes the controller to record the traceWatcher in its dispatch database (§8.1.3). The controller then monitors the traceWatcher's model, and sends events representing the model's actions to the traceWatcher. The traceWatcher has no special definitions of event handling methods, so its inherited behaviour will forward all up events it receives up the watcher tree to its superwatcher or view. Similarly, the traceWatcher will forward any down events it receives to its controller, and thus to its target object.

When the traceWatcher is detached, the leaf category watcher's default behaviour will deregister the traceWatcher from the controller, and so the traceWatcher will no longer receive events.

7.2.3 Selective Monitoring

Leaf watchers can embody more selective strategies than monitoring all their target objects' actions. Tarraingím provides extra leaf watchers which supply more selective monitoring plans to their controllers, so that actions which are not required by their views do not need to be monitored (§4.3). These more selective watchers are equivalent to using a filter watcher attached to a more basic leaf watcher (§7.3.1).

For example, the singleMsgWatcher, illustrated in Figure 7.4, monitors only actions caused by the single specific message named by its message parameter. The singleMsgWatcher uses the add:For message to register itself with its controller. This message is also part of the controller registering protocol, (§8.1.2).

```
traits singleMsgWatcher = (|
    _ parent* = traits leafWatcher.
    _ localAttach = (control add: self For: message).
|)

singleMsgWatcher = (|
    _ parent* = traits singleMsgWatcher.
  "parameter"
    ^ message ← 'foo'.
|)
```

Figure 7.4: Single Message Watcher Implementation

Another selective local strategy is implemented by the topWatcher, shown in Figure 7.5. The top-Watcher forwards a change to its view whenever the last thread of control leaves its model (§8.2.1), that is, whenever the model has completed processing a large granularity operation (§4.3.2). Since the model is now quiescent, a view can send synchronous callbacks in response to these changes (§4.2.1).

```
traits topWatcher = (|
    _ parent* = traits leafWatcher.
    _ localAttach = (control addTopLevelReturn: self).
|)
```

Figure 7.5: Top Level Return Watcher Implementation

By using information about the target program's behaviour, the program can be monitored more efficiently (§4.3.1). For example, accessor methods, such as printString or size, do not change an object's abstract state, and thus do not need to be forwarded to the view.

The changeWatcher, shown in Figure 7.6, is an example of a watcher implementing a strategy which takes information about the program into account. The changeWatcher is a development of the top-Watcher, but whereas the topWatcher generates events for all messages executed by its model, the change-Watcher sends changes to its view for only those messages the visualiser considers significant. In Figure 7.6 the messages monitored are at:Put and removeAll, which are the messages sent by a profileWatcher to its tally collection (§7.4.1).

7.2.4 Monitoring Local State Changes

A view can be updated by monitoring its target object's state changes, rather than the messages it receives (§4.3.4). In SELF, changes to an object's local state are made by message sends to assignment

```
traits changeWatcher = (|
      _ parent* = traits leafWatcher.

      _ localAttach = (
             control addTopLevelReturn: self For: 'at:Put:'.
             control addTopLevelReturn: self For: 'removeAll:'.
             self).
|)
```

Figure 7.6: Top Level Change Watcher Implementation

slots (§5.4.2). Assignments can be detected by monitoring these assignment messages, and so all changes in a single object can be detected by monitoring all that object's assignment messages.

This strategy is implemented by the localWatcher, shown in Figure 7.7, which simply determines the names of its model's assignment slots, and requests monitoring of these messages. The localWatcher's localAttach method first assigns a mirror object (§5.4.5), reflecting on the localWatcher's model, to the modelMirror temporary variable. The watcher's controller is then requested to monitor all the model's assignment slots.

```
traits localWatcher = (|
      _ parent* = traits leafWatcher.

      _ localAttach = ( | modelMirror. |
      "get a mirror on the model"
             modelMirror: (reflect: model).
      "monitor all assignment slots"
             control addReturn: self
                    ForAll: modelMirror localAssignmentSlots.
      "and if vector, monitor wrapper method too"
             modelMirror isReflecteeVector
                    ifTrue: [control addReturn: self For: 'at:Put:IfAbsent']).
|)
```

Figure 7.7: Local Change Watcher Implementation

Some SELF objects, such as vectors, are *primitive* — they are implemented within the SELF virtual machine (VM). Section 9.3.2 describes how Tarraingím monitors these object's actions by monitoring the SELF-level wrapper messages which are used to access them. The localWatcher checks whether its model is a vector: if so, its model's at:Put:IfAbsent wrapper method is also monitored.

The localWatcher sends changes to its superwatcher (or view) after the action causing the change is complete — for example, when a message to an assignment slot has returned. As described in Section 4.3.3, some views need to receive notifications before (or both before and after) the change has taken place. Tarraingím's library therefore includes preLocalWatcher and allLocalWatcher prototypes. These are variants of the localWatcher which forward message receipt events (or both receipt and completion events) as changes to their view. The implementations of the preLocalWatcher and allLocalWatcher are basically the same as the localWatcher shown in Figure 7.7, but they request different events from their controllers.

Local change strategies are generally quite efficient, since most objects only have a few data slots which must be monitored. However, local change strategies can only be used when an abstraction is implemented by a single, self-contained target object (§4.3.4). Any changes generated by a local change watcher are not synchronous, that is, they do not correspond to atomic operations of the target object's

abstraction, unless the object has a very simple representation (§4.2.1). A small (but important) set of SELF objects do meet these conditions: those objects, such as vectors, which are used as if they were structured data types. These basic objects are commonly used in the implementation of more abstract objects, and can be monitored efficiently by local state change strategies, implemented by variants of localWatchers.

7.2.5 Alternative Strategies

Tarraingím's framework is sufficiently flexible that it can work with events from a variety of sources, as well as the events generated by the monitoring subsystem's encapsulators. Leaf watchers can be used to inject these events into the rest of the framework. For example, the localTimerWatcher shown in Figure 7.8 implements a strategy based upon polling (§4.3.4). The localTimerWatcher provides much the same information as a localWatcher, but it gathers this information by periodically inspecting its model, rather than monitoring the program directly.

```
traits localTimerWatcher = (|
      _ parent* = traits leafWatcher.

      _ localAttach = ( | modelMirror. |
       "get a mirror on the model"
            modelMirror: (reflect: model).
       "initialise and start the ticker"
            ticker message: (message copy receiver: self Selector: 'tick').
            ticker interval: interval.
            ticker start).

      "ticker will send the watcher this message every interval seconds"
      ^ tick = (modelMirror localDataSlots do: [ | :slot |
                  (timerEvent copyFor: model
                              Name: (slot,':')
                              With: (modelMirror at: slot) contents)
                     sendTo: upEvents.
                 ]).

      _ localDetach = (ticker stop).
|)

localTimerWatcher = (|
      _ parent* = traits localTimerWatcher.
    "parameters"
       ^ interval ← 60.
    "variables"
        _ ticker ← ping.
        _ modelMirror.
|)
```

Figure 7.8: Timer Watcher Implementation

The localTimerWatcher's localAttach method starts a process (managed by a ticker object) that repeatedly sends the watcher a tick message. When the localTimerWatcher receives this message, it uses a mirror to inspect its model, and generates timerEvents describing the current contents of its model's slots. These timerEvents are dispatched up the watcher tree (along the upEvents pointer §7.1.4) using the sendTo message from the event dispatch protocol (see Table 8.6).

The localTimerWatcher implementation shown in Figure 7.8 is quite naïve, as the tick method always sends a change (i.e., a timerEvent) for every data slot in the model. If a particular data slot has not actually changed in the interval between ticks, the timerEvent will be sent unnecessarily. To eliminate these duplicate changes, the naïve localTimerWatcher can be composed with a cacheWatcher (§7.3.2).

7.3   Filter Watchers

Filter watchers use one subwatcher. They can be inserted above another watcher in the watcher tree (which becomes the filter watcher's subwatcher) to modify the effects of that watcher without having to modify its implementation.

Filter watchers generally do not change their aim or model. They pass the aim they receive from their superwatcher down the watcher tree to their subwatcher, and return the model they receive from their subwatcher up to their superwatcher.

This section presents examples of three kinds of filter watchers. Basic filters (§7.3.1) simply delete some of the events they receive from their subwatcher; caches (§7.3.2) remove duplicate events; and adaptors (§7.3.3) translate callbacks and changes so that a view can be used to display an object for which it was not designed.

7.3.1   Filters

Filter watchers can be used to filter events generated by their subwatchers. They are used to restrict the changes received by a view, so that the view receives only those changes to which it should respond (§4.3).

When a filter watcher receives an event (via the changeEvent message), it checks the event against its filter condition. If the event meets the conditions it is forwarded by the filter; if not, it is discarded. Figure 7.9 shows a simple filter watcher, a prefixWatcher, which passes only those events whose message name begins with foo.

```
traits prefixWatcher = (|
        _ parent* = traits filterWatcher.
        _ changeEvent: e = (
              ('foo' isPrefixOf: e name)
                    ifTrue: [e sendTo: upEvents]).
|)
```

Figure 7.9: Filter Watcher Implementation

7.3.2   Caches

A cache watcher is a development of the basic filter watcher. Whereas a filter watcher discards events on the basis of a simple predicate, a cache watcher removes duplicate events. For example, a cacheWatcher placed between a view and a localTimerWatcher will remove the duplicate events generated by the local-TimerWatcher. Thus the view will then receive changes only when its model has actually changed.

Figure 7.10 shows a simple implementation of a cacheWatcher, which caches only one-argument events, such as those produced by a localWatcher or localTimerWatcher. The cache is a SELF dictionary object, held in the cacheWatcher's variable named cache. Events are handled by the changeEvent message. When an event arrives, it is checked to see that it has only one argument: if not, it is routed up the watcher tree (§7.1.4). The names of one argument events are then looked up in the cache, and the cached value compared with the message's argument. If the cached value is different from the new event's argument value, or the message name was not found in the cache, then the event is passed up the watcher tree, and the cache updated with the message's argument.

```
traits cacheWatcher = (|
      _ parent* = traits filterWatcher.

      _ changeEvent: e = (|old|
      "forward unsuitable events"
            e arguments size ≠ 1
                  ifTrue: [^e sendTo: upEvents].
      "get old value"
            old: cache at: e name IfAbsent: sentinel.
      "if the value is different, update cache and forward event"
            (old = (e at: 0))
                  ifFalse: [cache at: e name Put: e at: 0.
                        e sendTo: upEvents]).

      "sentinel — not equal to anything, including itself"
      sentinel = (| = x = (0 false) |).
|)

cacheWatcher = filterWatcher _Add: (|
      _ parent* = traits cacheWatcher.
   "variables"
      cache ← dictionary copy.
|)
```

Figure 7.10: Cache Watcher Implementation

7.3.3 Adaptors

Filter watchers can implement adaptors [163, 79]. An adaptor allows a view to visualise an object with which it would not normally be compatible. Adaptors translate events flowing through them, altering the event's parameters — the name of the message causing the event, and the values and types of the event's arguments (§8.3.2).

Consider an alternative representation of a traffic light (an altLight). Unlike the abstract object presented in Section 6.2, an altLight represents a traffic light as a vector containing three integers. An element value of 1 represents an aspect that is illuminated; any other value represents an aspect that is not illuminated. An altLight has no protection or interface — objects wishing to manipulate it do so by sending messages directly to the vector.

> "create an altLight showing an amber aspect"
>   altLight: vector copySize: 3 FillingWith: 0.
>   altLight at: 1 Put: 1. "self vectors are indexed from 0"

The tlView view illustrated in Figure 6.5 cannot visualise an altLight. In order to redraw itself, the view sends the isRed, isAmber and isGreen messages to its model, but an altLight does not implement these messages. A trafficLight view expects to receive changes about a colour assignment slot, and red, amber and green mutator messages. An altLight will simply generate two array accesses whenever it is changed — one as the current aspect is turned off and another as the new aspect is turned on.

> "change the amber light to green"
>   altLight at: 1 Put: 0.
>   altLight at: 2 Put: 1.

```
traits tlAdaptor = (|
    _ parent* = traits filterWatcher.

  "handle upEvents"
      ^ changeEvent: e = (
            | names = ('red' & 'amber' & 'green') asVector |
            ((e name) = 'at:Put:IfAbsent:') && [(e at: 1) = 1]
              ifTrue: [
                    (e copyName: names at: (e at: 0))
                          sendTo: upEvents]).
  "handle downEvents"
      ^ callbackEvent: e = (
            | names = ('isRed' & 'isAmber' & 'isGreen') asVector |
            (e copyName 'at:' With: (names keyAt: e name))
                  sendTo: down).
|)
```

Figure 7.11: Adaptor Watcher Implementation

To enable a trafficLight view to visualise an altLight, the callbacks sent by the view must be translated into vector assignments, and the vector actions of the altLight must be translated into trafficLight messages. This translation is implemented by the tlAdaptor watcher, illustrated in Figure 7.11.

When the tlAdaptor receives a changeEvent travelling up the watcher tree, it first checks that this event describes a vector assignment (i.e., that it is an at:Put:IfAbsent: message) and that the vector element is being set to 1 (i.e., that an aspect is being illuminated). If so, a new event which can be understood by the tlView is created and forwarded up the watcher tree. When a callbackEvent is received from the tlView (via the callbackEvent message in the tlAdaptor) a new callbackEvent interrogating the altLight vector is created and forwarded down the watcher tree. Each method uses a literal vector called names to translate between the names used by the trafficLight (and the tlView) and the vector positions used by the altLight.

7.4   Indirect Watchers

Indirect watchers separate their aim and their model. We call these watchers *indirect* because the model — the object actually displayed by the view — is not specified directly by the user. The user specifies the aim, and the indirect watcher computes the model and sends it to the view. For this reason, indirect watchers use two subwatchers — a main subwatcher to monitor the model, and an aux subwatcher to monitor the aim.

Indirect watchers have two main purposes. First, aggregate abstractions (§3.1.3) can be visualised by monitoring the aim, and using the information produced by the monitoring to maintain a database which is assigned to the watcher's model (§7.4.1). Second, implementation views displaying an object's components (such as the stack implementation view from Figure 3.3) can use indirect watchers to specify the components by reference (§7.4.2).

Section 10.2.3 presents two illustrated examples of the use of indirect watchers.

7.4.1   Aggregate Abstractions

Aggregate abstractions can be displayed by a watcher interposing a new model between a view and its aim. For example, an execution profile (such as the operation profile view from Figure 3.4) can be created by monitoring an object, and building a database of event frequencies into a table. This table is then monitored and displayed by a suitable view.

A simple operation profile is provided by the profileWatcher illustrated in Figure 7.12. The profile-Watcher uses two subwatchers, held in the profileWatcher's main and aux variables. The profileWatcher's aim is monitored by its aux subwatcher. The aux subwatcher is a recvWatcher, which is similar to a traceWatcher (§7.2.2) but sends changes for all message receipt events. Events received from the aim are used to update a dictionary (held in the profileWatcher's profileTally variable) which maps message names into the number of times particular messages have been called. The view's model is set to refer to the profileTally, which is monitored by the main subwatcher. Events detected from the profileTally by the main subwatcher are forwarded up the watcher tree. The view thus receives events regarding the profileTally, rather than the original object, and any callbacks it generates will also be directed to the tally. This routeing is illustrated in Figure 7.13.

```
traits profileWatcher = (|
        parent* = traits indirectWatcher.

    "attach subwatchers"
        ^ localAttach = (
                main upEvent: upEvent.    "events from main bypass us"
                main watch: profileTally.  "attach main watcher"
                attached ifTrue: [aux watch: aim]).  "and aux watcher"

    "handle events from the aux watcher"
        ^ changeEvent: e = (
                command at: e name Put: (callback at: e name IfAbsent: 0) succ).

    "route callbacks to main watcher and model"
        ^ callbackEvent: e = (e sendTo: main).

    "handle configuration messages from subwatchers"
        ^ sub: sw Model: mod = (
                main = sw ifTrue: [model: mod. up sub: self Model: mod]).
        ^ sub: sw Warning: msg = (up sub: self Warning: msg).
        ^ sub: sw Error: msg = (detach. up sub: self Error: msg).
|)

profile = (|
        _ parent* = traits profile.

        ^ profileTally ← dictionary.
        ^ aux ← recvWatcher.
        ^ main ← changeWatcher.
|)
```

Figure 7.12: Profile Watcher Implementation

The profileWatcher's localAttach method is called to attach the profileWatcher's subwatchers. The main watcher (monitoring the tally) is attached first, and its upEvents pointer is set to forward any events it generates to bypass the profileWatcher. If the main subwatcher is successfully attached, the aux watcher is attached. Events from the aux watcher are handled by the changeEvent method, which increments the entry in the tally for the message name of the event, by sending a command callback. Configuration messages (sub:Warning and sub:Error) are generally passed up the watcher tree, but any sub:Model messages from the aux watcher are ignored, as the main watcher determines the model for the whole profileWatcher (§7.1.3).
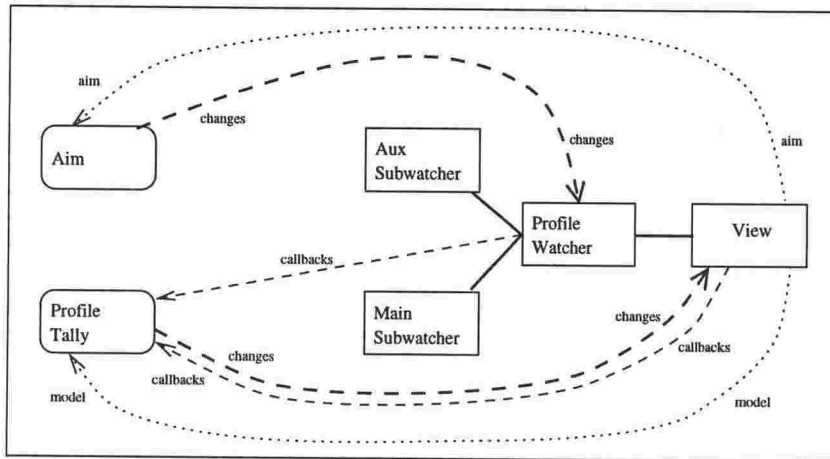
Figure 7.13: Event Routeing in a profileWatcher

7.4.2   Passing Models by Reference

Consider the stack implementation view presented in Figure 3.3. This view illustrates the implementation of the stack abstraction by displaying the components of the stack object. In particular, it includes a view of the contents component of the stack. The contents component is a vector which is assigned to the contents variable of the stack object. Since SELF vectors have a fixed size, if a stack needs to hold more elements than will fit into its current contents vector, it must allocate a larger vector, fill it with the elements of the old contents vector, and then replace the old contents vector with the new vector.

If a view of the contents component of a stack simply displays the original contents vector object, when that object is replaced within the stack object, the view will not notice this change and will continue to display the original vector object which is no longer part of the stack implementation. The model for a view of the stacks contents vector must be specified *by reference* (i.e., as the value of the contents slot of the stack object), rather than directly (i.e., as a particular vector object).

An indirect reference watcher can be used to dynamically redirect a watcher's model. By using an indirect reference watcher, a view's model can be specified by reference, and kept correct as the target program evolves. By using several indirect reference watchers, a view's model can be specified by a path of arbitrary length from its aim. This is similar to the use of pointer variables to denote objects in constraint systems [219].

Slot Watcher

A slotWatcher (see Figure 7.14) is a simple example of an indirect reference watcher. A slotWatcher monitors a particular slot of its aim and sets its model to that slot's contents. The aim is monitored by a cmplMsgWatcher: a version of a singleMsgWatcher (§7.2.3) which monitors only completion events.

When a slotWatcher is attached, its localAttach method is executed, which attempts to attach the aux subwatcher. When this subwatcher successfully acquires a model, it will send a sub:Model message up to the slotWatcher. The slotWatcher responds to this message by attempting to attach its main subwatcher to its model using the attachMain method. Whenever an event is received from the aux watcher, it is handled by the changeEvent method, which attaches the main watcher to the new model.

A slotWatcher will become provisionally attached if its main subwatcher cannot be attached, provided that its aux subwatcher is attached. This is because the slotWatcher's model can change (such a change being detected by the aux subwatcher), and so the main subwatcher may be able to attach to a different model in the future. Any sub:Error messages indicating attachment errors from the main subwatcher are therefore translated into sub:Warning messages, to indicate that the slotWatcher is now provisionally attached.

```
traits slotWatcher = (|
        _ parent* = traits indirectWatcher.

   "attach the aux watcher"
        _ localAttach = (
                aux message: (slot,':') canonicalize.   "select slot"
                aux watch: aim).   "attach aux watcher"

   "attach the main watcher whenever the local watcher changes"
        _ attachMain: inter = (| mir |
                mir: reflect: inter.
                (mir names includes: slot)
                        ifFalse: [^up sub: self Warning: 'No such slot: ',slot].
                attached ifTrue: [
                        main upEvent: upEvent.
                        main watch: (mir at: slot) contents reflectee]).

   "route callbacks to model"
        ^ callbackEvent: e = (e sendTo: main).

   "handle events from the aux watcher"
        ^ changeEvent: e = (main watch: e at: 0).

   "handle configuration messages from subwatchers"
        ^ sub: sw Model: m = (
                aux = sw
                  ifTrue: [attachMain: m]
                  False: [model: m. up sub: self Model: mod]).
        ^ sub: sw Error: msg = (
                aux = sw
                  ifTrue: [detach. up sub: self Error: msg]
                  False: [up sub: self Warning: msg]).
|)

slotWatcher = (|
        _ parent* = traits slotWatcher.
   "parameters"
        ^ slot ← 'foo'.
        ^ main ← localWatcher.
        ^ aux ← cmplMsgWatcher.
|)
```

Figure 7.14: Indirect Slot Watcher Implementation

If the aux subwatcher is unable to attach itself to the slotWatcher's aim (and thus sends a sub:Error message to the slotWatcher), the indirect watcher as a whole can never become attached, as it will never be able to determine its model. If the slotWatcher receives a sub:Error message from its aux subwatcher, it therefore detaches itself and relays the sub:Error message up the watcher tree.

Figure 7.15 illustrates the routeing within a slotWatcher. The view's aim is monitored by the aux subwatcher. The slotWatcher inspects this object to retrieve the value of the slot containing the model whenever the aux subwatcher indicates that it has changed. The model is itself monitored by the main subwatcher, which has its upEvents pointer set to route events directly to the slotWatcher's superwatcher

or view.
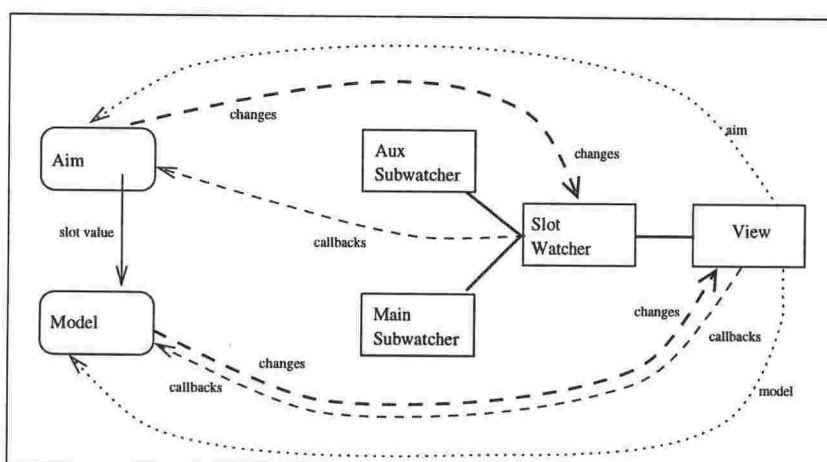
Figure 7.15: Event Routeing in a slotWatcher

Message Watcher

A slotWatcher allows an object to be specified relative to the contents of another object's slot. Other kinds of indirect reference watchers can also be constructed. For example, a messageWatcher specifies a view's model by reference to the *result* of a message sent to its aim, that is, according to the result of an abstract operation.

Consider again the stack example. A slotWatcher can be used when constructing a view of its implementation, looking inside the stack object and visualising each of its component objects separately. To produce a view of the object on top of the stack, the top object must be retrieved from the stack's implementation. A messageWatcher solves this problem, by calculating its model by sending a top message to the stack object, rather than directly inspecting the stack object's implementation. The stack must be monitored, and the model recomputed when it changes.

A messageWatcher is implemented by making two changes to the basic slotWatcher. First, the main subwatcher's model must be retrieved by sending a callback to the aim, rather than simply accessing a slot. Second, the whole abstraction represented by the stack must be monitored, rather than just one of stack object's slots, so the cmplMsgWatcher acting as the aux subwatcher must be replaced by a watcher implementing a more powerful strategy, such as a changeWatcher or topWatcher (§7.2.3).

7.5 Multiple Watchers

A multiple watcher employs many subwatchers. In this way, a watcher can monitor any number of objects. As with binary watchers, multiple watchers do not monitor these objects directly, rather a multiple watcher combines information gathered by a number of subwatchers, one for each object being monitored.

Multiple watchers are similar to hierarchical views (§6.5) in many ways. Hierarchical views and multiple watchers both allow information about multiple objects to be displayed in a single window. But whereas a hierarchical view uses a separate subview (and associated watcher) to display each object, a multiple watcher (and several subwatchers) can be used by a unitary view to display a number of objects directly, without intervening subviews.

A multiple watcher obviously requires multiple target objects to watch. Tarraingím's framework is designed so that watchers and views accept only a single object as their target — for example, the watch messages in the view public protocol (see Table 6.1) and watcher down protocol (see Table 7.2) have

only one argument. Multiple watchers therefore take a single object as their model, which by convention contains the multiple target objects to be monitored. Depending on the particular multiple watcher, this object may or may not be part of the program, and may or may not be monitored by the watcher.

Event routeing within multiple watchers is simpler than routeing within binary watchers. Generally, up events received from subwatchers are passed directly up to the multiple watcher's view or parent watcher, and down events are sent directly to the object which is the multiple watcher's model.

7.5.1   Aggregate Algorithmic Strategies

Multiple watchers can be used to produce aggregate views of algorithmic abstractions. For example, a standard trace or profile view, combined with a multiple watcher, can display the messages processed by a group of objects, whereas the same views display only a single object if a leaf or filter watcher is used. Other aggregate visualisations, such as timing diagrams [124], object call clusters [165], and road maps [104], could be produced using multiple watchers given suitable views.

The simplest multiple watcher is the multipleObjectWatcher, illustrated in Figure 7.16. A multipleObjectWatcher simply deploys subwatchers to monitor all the elements of the collection object it receives as its model, and forwards events it receives from these subwatchers up the watcher hierarchy.

```
traits multipleObjectWatcher = (|
        parent* = traits multipleWatcher.

   "attach subwatchers"
        ^ localAttach = (
              main watch: model.
              attached ifTrue: [callback do: [|:target|
                    addSubWatcher: (subProto copyWatch: target)]].
              attached ifTrue: [up sub: self Model: model]).

   "handle upEvents"
        ^ changeEvent: e = (e sendTo: upEvents).
   "handle downEvents"
        ^ callbackEvent: e = (e sendTo: main).
|)

multipleObjectWatcher = (|
        _ parent* = traits multipleObjectWatcher.

   "variables"
        ^ subWatchers ← list copy.
   "parameters"
        ^ subProto ← traceWatcher.
        ^ main ← nullWatcher.
|)
```

Figure 7.16: Multiple Object Watcher Implementation

When a multipleObjectWatcher is attached, it must attach its subwatchers — a main subwatcher which refers to its model (the collection object containing the actual target objects), and a series of other subwatchers to monitor the actual target objects. As with any other watcher, when a multipleObjectWatcher is attached, its localAttach method is executed, and this method begins by attaching the main subwatcher to its model. The multipleObjectWatcher does not require any dynamic information about the collection, so a nullWatcher is used (§7.2.1). Once the main subwatcher is successfully attached, the multipleObjectWatcher must monitor the actual target objects the model contains. To do this, it first sends a callback

to its model (using the do message) to enumerate the actual target objects, then creates a series of sub-watchers to monitor each of them. The subwatchers are created by cloning the multipleObjectWatcher's subProto watcher parameter, in much the same way that hierarchical views clone their view prototype parameters (§6.5). The resulting subwatchers monitor the actual target objects and send change events describing those objects' actions back up to the multipleObjectWatcher.

When the target program is running, the multipleObjectWatcher simply forwards any change events it receives from its subwatchers up the watcher hierarchy (via the changeEvent method), and forwards any callbacks it receives from its superwatcher down to its model via the main subwatcher (via the callbackEvent message.)

7.5.2 Anti-Aliasing Strategies

A multipleObjectWatcher simply monitors a group of objects. A shadowWatcher is a more complex multiple watcher which implements a strategy to mitigate the effects of aliasing, by detecting whenever an object is modified by a message sent via an alias rather than via the object's interface (§4.4). A shadowWatcher can be used to produce views of program abstractions where their implementation is susceptible to aliasing problems.

A shadowWatcher inspects its model's structure, and attempts to determine the members of the object complex of which its model is the head (§4.4.1). The complex's members are then monitored in addition to the model. A view using a shadowWatcher is thus informed when an aliased subcomponent of its model may have changed. We call this watcher a shadowWatcher because it depends upon a heuristic approximation of the model's object complex, since the precise object complex cannot be determined easily. We call this approximation the model's *shadow set*, and it is computed as a transitive closure over objects' variable slots (references to other objects), starting at the model. The shadow set is thus a subgraph of the object graph (§3.6), rooted at the model. The shadow set is an approximation because it can both underestimate and overestimate the object complex. The shadow set can underestimate the complex by not enumerating all references upon which the complex depends — for example, Tarraingím's definition of the shadow set ignores references through inherited objects. The shadow set can overestimate by enumerating objects which the head object does not in fact depend upon — for example, by including objects which are reachable from the head object, but which are encapsulated parts of another object's implementation. The shadowWatcher enumerates the shadow set, and monitors the local state of each object in the set. When an object's state changes, the shadow set is recomputed, and the monitoring is adjusted to correspond to the new shadow set.

An implementation of a shadowWatcher is outlined in Figure 7.17. When it is attached, its localAttach method is invoked. This attaches a main subwatcher, which, as in the multipleObjectWatcher, is a nullWatcher used mainly for handling callbacks. The localAttach method sends the recalcShadow message, which then sends the computeShadowSet message to a mirror on the model. The computeShadowSet message computes the shadow set using a depth-first search, and returns a set of mirrors reflecting the objects contained in the shadow set. Once the shadow set has been calculated, an adjuster (§6.5.2) is used to create a subwatcher to monitor each object in the set. These subwatchers are created by cloning the subProto parameter.

When an event is received by a shadowWatcher from a subwatcher, it is handled by the changeEvent message. This passes the event up the watcher tree, then sends recalcShadow to recompute the shadow set and adjust the subwatchers. Any new objects in the shadow set will have a subwatcher cloned and attached to them, and any objects no longer in the set will have their subwatcher removed.

```
traits shadowWatcher = (|
      parent* = traits multipleWatcher.

  "attach subwatchers"
       ^ localAttach = (
              main watch: model.
              recalcShadow.
              attached ifTrue: [up sub: self Model: model]).

  "recalculate extent of object complex"
       _ recalcShadow = (| shadow |
              shadow: modelMirror computeShadowSet.
              adjuster adjust: subWatchers
                    Keys: [|:sw| (reflect: sw model)]
                    To: shadow
                    Create: [|:mir|
                          addSubWatcher:
                                (subProto copyWatch: mir reflectee]
                    Keep: []
                    Destroy: [|:old| removeSubWatcher: old]).

  "handle up events"
       ^ changeEvent: e = (
              e sendTo: upEvents.
              recalcShadow).
|)

shadowWatcher =  (|
       _ parent* = traits shadowWatcher.
  "variables"
        ^ subWatchers ← list copy.
  "parameters"
        ^ subProto ← localWatcher.
        ^ main ← nullWatcher.
|)
```

Figure 7.17: Shadow Watcher Implementation

7.6   Tarraingím's Watcher Library

Tarraingím's watcher library, currently containing approximately fifteen different watchers, is about a third of the size of Tarraingím's view library. The watchers presented in this chapter have therefore described a large proportion of the content of the watcher library. We conclude our discussion of watchers with a description of our use of watchers to support the views in the view library.

7.6.1   Leaf Watchers

The majority of watchers in Tarraingím's watcher library are leaf watchers, and these are also the type of watchers most often used by views. The basic leaf watchers described in Section 7.2 (especially the nullWatcher, traceWatcher, topWatcher and localWatcher) are the watchers we have used most often, followed by customised variants of these watchers — for example, the Quicksort hops view in Figure 3.2 uses a variant of the cmplMsgWatcher.

Leaf watchers ultimately provide views and other watchers with access to the facilities provided by the controllers and encapsulators of the monitoring subsystem. Controllers and encapsulators are discussed in the following two chapters.

7.6.2   Filter Watchers

Most of the filter watchers we have used have been cacheWatchers, used in conjunction with localWatchers, as described in Section 7.3.2. Views which use a localWatcher can generally replace it with a cachingLocalWatcher (a combination of a localWatcher and a cacheWatcher) to eliminate spurious updates with no other effects.

The library views do not make much use of the basic filter watchers. The event dispatch mechanisms provided by Tarraingím's controllers (§8.1.3) effectively provide filtering on event parameters to leaf watchers, without the use of filter watchers. If the monitoring component is not able to provide such precise monitoring, filter watchers can be combined with leaf watchers to provide the same effect. Dynamic filtering, especially if based upon the current properties of the target program, is easier to implement in filter watchers than in controllers.

The library similarly does not make much use of adaptors, since we have had the luxury of constructing views to match their target objects. This may be in part because, in practice, target objects fall into one of two categories: either they are part of the SELF library, have a well defined interface, and can be displayed by generic views without adaptation; or they are custom objects, part of a particular program, and so require the construction of a custom view tailored to be compatible with them. Adaptors could prove more useful if Tarraingím was used to visualise a new program in a domain for which views had already been written, however, we have not yet carried out such experiments.

7.6.3   Indirect Watchers

Tarraingím's library contains several indirect watchers which provide aggregate abstractions of a program's behaviour or structure. The basic profileWatcher can be adapted to generate a variety of different profiles by changing the aux watcher used to monitor its aim. For example, a profile of assignments to an object's variable slots can be produced by replacing the recvWatcher by a localWatcher, or a profile of top level operations produced by using a topWatcher. We have also implemented several other aggregate abstractions of an object's behaviour. For example, an objectStackWatcher keeps track of the active message sends in an object and displays them as a stack, and the objectTreeWatcher calculates a tree of the history of an object's message invocations.

Several other watchers also produce aggregate abstractions which can be displayed by tree views. For example, the childTreeWatcher and parentTreeWatcher can be used to display inheritance hierarchies (such as Figure 7.1), and the structureTreeWatcher can display abstraction structure diagrams (such as the stack abstraction view in Figure 3.4).

Indirect watchers also provide an alternative to adaptors. An indirect watcher can be used to construct an aggregate abstraction based upon the original target object, that is directly compatible with the view. This changes the view's model, in contrast to an adaptor, which translates the view's callbacks and changes but does not change the model. Using an aggregate abstraction can be easier for the visualiser where complex objects are involved, as the target object can be translated in one batch pass when the watcher is initialised, rather than writing an adaptor which must translate every change and callback. Using an aggregate abstraction has some disadvantages if incremental updating or user input is required, as incremental changes from the original target object (the view's aim) must be translated into incremental changes in the aggregate abstraction model, and the user's commands must be translated from the model to the aim.

Indirect reference watchers (the slotWatcher and messageWatcher §7.4.2) are unique, since they can be used with any view and any other watcher. Although these watchers redirect their model, callbacks, and changes, they do not alter the event traffic in any way. We have found indirect reference watchers indispensable in constructing implementation views.

7.6.4   Multiple Watchers

The main application of multiple watchers has been for managing aliasing, as in the shadowWatcher. Very few of the views in the view library have had problems with aliasing in the target program, so, as with filter watchers, we have not used multiple views very much. There are two main reasons for this. Firstly, although aliasing is endemic in object oriented programs, the mere presence of aliasing does not necessarily affect a visualisation. A visualisation is only affected when an object is changed via an alias crossing the boundary of an object complex (§4.4.1). Many SELF objects (including most of the collections) only function correctly under the assumption that they will not suffer from the effects of unintended aliasing.

Secondly, we have avoided many potential aliasing problems by using multiple views instead of multiple watchers. For example, the elements of a collection can often be modified without reference to the collection containing them. A monolithic view displaying the collection is therefore affected by aliasing problems. A hierarchical view of the collection uses separate subviews to display each element: these views monitor the elements they are displaying, and are notified of any changes in their target element. The hierarchical view only needs to receive changes about the structure of the collection itself, and for this, monitoring only the collection object is usually sufficient.

Finally, it is interesting to note that there is one view which is very susceptible to aliasing. This view is the printStringView, a very simple view which displays an object's printed representation (known as the object's printString). The printString of a SELF object can include the printStrings of a large number of other objects, many of which are not normally considered part of the original object being printed. For example, a sequence responds to printString by recursively sending printString to each of its elements. If any of these elements are also sequences, they will print their elements, and so on. Although few of the objects printed will be part of the original sequence, a change in any of these objects can change the original sequence's printString. A watcher which handles aliasing must therefore be used with a printStringView.

22. A good system can't have a weak command language.

Alan Perlis, *Epigrams On Programming* [168]

8

Monitoring Subsystem

This chapter describes the design of the monitoring subsystem that implements the program component of the APMV model (§3.7). The first section (§8.1) describes the controller objects which provide an interface between this subsystem and the rest of Tarraingím. The next section (§8.2) describes how controllers manage the flow of control between the target program and Tarraingím. Section 8.3 describes the event objects which carry information around Tarraingím's framework. Although events do not belong to any particular subsystem, by far the majority of events are created by controllers in response to the monitored actions of the target program — therefore they are discussed here. Section 8.4 concludes the chapter.

Tarraingím's encapsulator objects, which are used by controllers to perform the actual monitoring of the target program, are described in Chapter 9.

8.1 Controllers

Controllers have two main responsibilities within the monitoring subsystem. First, they provide the interface used to create and manage encapsulators. Second, they package data from encapsulators into events and distribute the events to the rest of the system. Controllers act as dynamic meta-objects within SELF, in much the same way that mirrors are static meta-objects. Both controllers and mirrors provide information about other objects in the program: a controller dispatches execution events, while a mirror describes object's slots. The interface provided by controllers is not specific to encapsulators. Encapsulators could be replaced by another monitoring technique organised around objects in the target program without changing the controller interface protocols.

This section discusses how controllers are created, then describes how they provide information about the target program. Note that for technical reasons, some behaviour properly local to encapsulators is implemented within controllers. This is described in Section 9.1.4.

8.1.1 Creating Controllers

From outside the subsystem, controllers are located via SELF's mirror objects (§5.4.5). Tarraingím adds the controller message to all mirror objects. Sending the controller message to a mirror returns the controller associated with the mirror's reflectee[1]. Unlike mirrors, controllers are canonical — there is at most one

---

[1] A mirror's *reflectee* is the object upon which that mirror reflects.

controller for any object in the target program. Once a controller has been created for a particular target object, it is shared by all other objects interested in that target object. The intended model is that every object has a unique *virtual* controller, yet only those controllers required in practice are created. Like all SELF objects, controllers are recovered by a garbage collector when they are no longer required.

8.1.2 Registering Clients

Once a controller has been created, it can be used to obtain information about the execution of its target object. A controller's *client* (typically a watcher) may register its interest in the target object by sending the target object's controller its monitoring plan using the controller registering protocol (see Table 8.1). A client object must be able to receive event notifications from a controller, that is, it must support the client event-handling protocol (§8.3.6). Both views and watchers support the client event protocol, but, in the current configuration of the framework, watchers are the only objects used as clients of controllers.

controller protocol

Registering	
add: client	Registers client with the controller. In the future, any (non-meta) events occurring within the target object will be dispatched to client.
remove: client	Deregisters client. No more events will be dispatched.

Table 8.1: Controller registering protocol

A controller continues to dispatch events to a client as long as that client remains registered. Once a client is no longer interested in the target object (perhaps a view has been closed by the user, or an indirect reference watcher has changed its model §7.4.2), the client sends a remove: message to deregister itself from the controller.

Although some watchers need to be notified about events occurring within the target object, many other watchers are not interested in most of the messages a controller could send to them (§7.2.3). For example, while a trace view needs to be notified about every action of its target object, a simple data structure view needs to be notified only when one of its target object's slots changes — in effect when one of a small set of messages has been completely executed by the object. The controller registering protocol includes optimised messages which allow a controller's clients to choose the events they wish to receive (see Table 8.2).

Creating Encapsulators

When a controller's client requests dynamic information about its target object, the controller will attach an encapsulator to its target. This encapsulator will remain attached while there is at least one client registered. When all clients are no longer interested in the target object, the encapsulator is removed. There are no explicit commands sent from a client to attach or remove an encapsulator. Instead, this is managed automatically, by the controller. For debugging purposes, controllers provide attach and detach messages, messages to determine whether an encapsulator is currently in use, and a message to return the controller's target object (see Table 8.3). Since removing an encapsulator is slow (§9.1.1) there is an option to defer removing an encapsulator if the user considers that the target object may have an encapsulator reattached at some future time.

8.1.3 Dispatching Events

A controller dispatches events to all interested clients. The events may originate from the controller's encapsulator or may have been sent from another part of Tarraingím. The encapsulator sends events to its controller using a private protocol, while other objects use the client event protocol (§8.3.6).

controller protocol
_____

Optimised Registering
    add: client For: message
                            Register client's interest in all actions caused by the message named message.
    add: client ForAll: collection
                            Register client's interest in all actions
                            caused by the messages named in the collection.
    addTopLevel: client   Register client's interest in all top level actions (§8.2.1).
    addReceipt: client   Register client's interest in all receipt actions.
    addReturn: client   Register client's interest in all return actions.
    addTopLevelReturn: client
                            Register client's interest in top level return actions (§8.2.1).
    add: client MetaDepth: md
                            Register client's interest in all meta-events
                            with a metaDepth less than or equal to md (§8.2.3).
        ... this is incomplete. There is one message in this protocol for each table in the dispatch
        database (§8.1.3), but this gives the flavour!

Table 8.2: Controller optimised registering protocol

controller protocol
_____

Debugging
    attach   Attempt to attach an encapsulator to the target object.
    detach   Remove any encapsulator from the target object.
    retain: flag   If flag is true, then never automatically detach an encapsulator.
    isMonitoringActive true if monitoring is in process, i.e., if an encapsulator is attached.
    target   Returns the controller's target object.

Table 8.3: Controller debugging protocol

When dealing with events from encapsulators, the controller receives the event type, name, arguments, return value and self from the encapsulator. The controller calculates the remainder of the event parameters, such as recursion depth, and meta-depth (§8.2), packages all the parameters into an event object, then dispatches the event to interested clients (§8.3.6).

These events must be dispatched so that Tarraingím, and in particular the event's client watchers and views, remain synchronised with the target program (§4.2.1). In practice, synchronisation can be ensured by using SELF standard message sends to dispatch events. Events are therefore sent to controllers from encapsulators using standard message sends. Controllers similarly use standard message sends to forward these events to their client watchers and views. Event clients therefore execute serially in the same process the controller uses to dispatch the events, that is, the target program process which originally caused the event to be generated. The target program is not restarted until all event processing has been completed and the dispatch message sends return.

Dispatch Database

A controller maintains a database of clients which are interested in that controller's target object. Since clients are able to select the parameters of events about which they wish to be notified (§8.1.2), the database is organised to allow the controller to notify its clients about only the actions in which they are interested. Note that clients (such as filterWatchers, §7.3.1) may further restrict the events they handle.

The dispatch database is illustrated in Figure 8.1. Program events are indexed by level, type and name, while other events are essentially not indexed. The indexed and nonindexed tables are implemented

by SELF's standard dictionaries and hash tables respectively.

Program Events	Receipt	Top Level	one table indexed by message name
			one table for all messages
		All Levels	one table indexed by message name
			one table for all messages
	Completed	Top Level	one table indexed by message name
			one table for all messages
		All Levels	one table indexed by message name
			one table for all messages
	Unwind	Top Level	one table indexed by message name
			one table for all messages
		All Levels	one table indexed by message name
			one table for all messages
Meta Events			one table for all messages

Figure 8.1: The controller's Dispatch Database

Optimised Event Creation

A controller may receive a message from an encapsulator when there are no clients interested in that particular type of event. In such cases, the controller will not create an event object. This happens surprisingly often, since Tarraingím's encapsulators intercept every message, and notify their controllers twice for each one (§9.1.2). Avoiding the creation of such unnecessary event objects increases Tarraingím's efficiency, especially with respect to the garbage collector.

8.2   Control Flow

Controllers manage the flow of control within Tarraingím. When an encapsulator detects an action within its target object, the encapsulator notifies its associated controller. The controller then dispatches events to its client watchers.

Controllers calculate two event parameters (§8.3.2) which describe the flow of control within the target program, and between the target program and Tarraingím. These parameters are the depth of method invocations within an object and the metaDepth of recursive monitoring within Tarraingím. In this section, we describe how these parameters are used to manage the flow of control within Tarraingím.

8.2.1   Event Depth and Top Level Events

Informally, the depth parameter of an event is the number of active method sends within the event's object when the event occurs. A common use of depth is to format a trace visualisation, as in Figure 8.2. This displays a trace of the behaviour of a trafficLight object (see Figure 6.4) which is sent the messages cycle and print. Each line representing a message receipt or return event is indented by the event's depth. The send of cycle in the first line of Figure 8.2 has a depth of one, the send of red in the second line of the figure has a depth of two, and so on.

Figure 8.2 includes a number in angle brackets before each message send. This is the identifier of the SELF process executing each message. Thus process <92> executes the cycle message (lines 1 to 18), while process <94> executes the printString message (lines 19 to 22). The effects of processes in SELF are discussed further in Section 8.2.2.

Depth

We define the depth of an event in terms of the depth of the event's object at the time the event occurs. A controller maintains a value for the depth of activations within its target object. An object's depth is

```
┌─trace view─┐
│□           │
│  1 <92>  cycle
│  2 <92>  red
│  3 <92>    colour: 'red'
│  4 <92>    colour: 'red' »» a red traffic light
│  5 <92>  red »» a red traffic light
│  6 <92>  amber
│  7 <92>    colour: 'amber'
│  8 <92>    colour: 'amber' »» a amber traffic light
│  9 <92>  amber »» a amber traffic light
│ 10 <92>  green
│ 11 <92>    colour: 'green'
│ 12 <92>    colour: 'green' »» a green traffic light
│ 13 <92>  green »» a green traffic light
│ 14 <92>  red
│ 15 <92>    colour: 'red'
│ 16 <92>    colour: 'red' »» a red traffic light
│ 17 <92>  red »» a red traffic light
│ 18 <92>  cycle »» a red traffic light
│ 19 <94>  printString
│ 20 <94>  colour
│ 21 <94>  colour »» 'red'
│ 22 <94>  printString »» 'a red traffic light'
└────────────┘
```
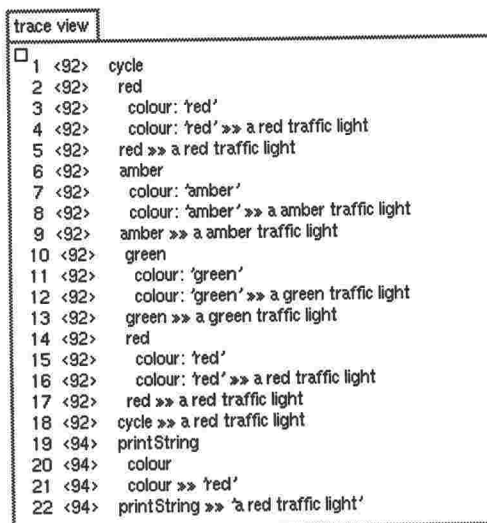
Figure 8.2: A Trace View

defined as follows:

1. When no messages are being evaluated within the object, the depth is zero.

2. When a message is received, the object's depth is increased by one *before* a receipt event is generated.

3. When a message returns (either locally or non-locally), the object's depth is decreased by one *after* a return event is generated.

An event's depth is simply the object's depth at the time the event is generated. The matching call and return events caused by a single message send normally have the same depth (§8.2.2). This is the case in Figure 8.2 where matching call and return events are indented by the same amount.

Note that messages sent from one object to another do not directly affect the sending object's depth (although the receiving object's depth is increased for the duration of the message). If a send results in the second object sending messages back to the first, these messages will affect the depth of the first object when they are received by that object. A self-send (such as the sends of red, amber and green from cycle in Figure 8.2, see also §5.4.2) alters the depth of the sending object because the sending object itself receives the self-message. Thus depth does not distinguish between inter-object and intra-object sends.

Top Level Events

An event is a *top level* event if it occurs due to a *top level* message send to an object, that is, when no other message sends are active within the object. Top level events appear at the very left of a message trace — for example, in Figure 8.2, the receipts and returns of the cycle and printString messages are top level events.

Using the definition of an event's depth, an event is a top level event if it has a depth of one. That is, if the event is a message receipt the object's depth was zero before receiving the event; if it is a return event the object's depth will be zero once the event has returned.

Top level events are important because they mark transitions in an object's activity status. When an object's depth is zero, it is quiescent, and, assuming its implementation is correct, the object is in a consistent state (§4.2.1). A top level receipt event indicates that a quiescent object is about to become active. A top level return event conversely indicates that an active object is about to become quiescent, that is, that an active object has just completed a large granularity operation (§4.3.2). Several watchers

therefore implement strategies using top level events (especially top level returnEvents) to synchronise the visualisation system and the target program (§7.2.3).

The event parameter topLevel is true if the event is a top level event. Because top level events are commonly distinguished within Tarraingím, controllers' dispatch databases are optimised according to this parameter (§8.1.3).

Initial Depth

The initial value of an object's depth is defined to be zero. When a controller is created for an object, its depth is set to zero. The controller's depth is updated once an encapsulator is attached to the target object and the controller begins to receive notifications of the target program's actions.

Initialising a controller's depth to zero can result in an incorrect value for an object's depth. An initial value of zero assumes no processes are executing inside the object when its controller is created. If messages are being executed within the object, the depth maintained by the controller will be too low. This can cause certain problems (§4.2.1). Any event depth parameters will also be too low, so trace views will be formatted incorrectly. Spurious topLevel events will be generated, so callbacks will be sent to objects unable to handle them.

To avoid these problems, the controller could determine an object's initial depth by inspecting all the running processes in the SELF system, and computing the actual depth of activations within the target object. We have not implemented this inspection for two reasons. First, carrying out such an inspection every time an encapsulator was attached to an object would slow Tarraingím's execution. Second, we have not found the lack of this precaution to be a problem in practice.

Tarraingím typically begins monitoring objects (either at the user's request or via indirect strategies) when the object is quiescent, or nearly so, and thus zero is a reasonable estimate for the object's depth. In cases where processes are active inside the object, Tarraingím ensures that depth cannot become negative, and thus the depth variable maintained by a controller eventually acquires the correct value.

8.2.2 Multiple Processes

SELF provides multiple processes (§5.2.6) so it is possible for more than one process to be active within a single object. A controller, however, maintains only one value for its target object's depth. By the definition of depth, an object's depth is the sum of the depths of all the processes inside the object. This is because the definition of depth depends only upon the type of actions (receipts or returns) received by controllers. The particular process in which the actions occur is ignored.

An effect of handling processes in this way is that an event will be considered top level only when no other processes are active within the object. A top level send is thus caused only by the first process entering the object and a top level return is caused only when the last process leaves. A second effect is that matching receipt and return events caused by a single message send may occur at different depths.

These effects are illustrated in Figure 8.3 (compare with Figure 8.2). The messages cycle and printString have again been sent to a trafficLight object, but in parallel using two different processes (processes <134> and <135>), and their execution has been interleaved. Only the receipt and return of cycle are now top level events. Although the print message is sent from, and returns to, the outside of the trafficLight object, the cycle message is active within the trafficLight during the entire execution of print. Message receipts and returns no longer occur at the same depth. For example, the printString message received on line 3 of Figure 8.3 now occurs at the same depth as the return of the colour message on line 8.

We have chosen this approach because SELF's object model provides no integrated support for parallelism. Although SELF provides some special objects (the library includes semaphore, process and sharedQueue prototypes), SELF objects in general are not concurrent. Most SELF programs contain only one process, and when multiple processes are used, the language does not permit any assumptions about their interaction within non-concurrent objects.

Our definition of depth is conservative: it allows views to function sensibly in the presence of multiple processes, but avoids the complexity of explicitly managing these processes. Trace views (such as

```
trace view
  1 <134>   cycle
  2 <134>     red
  3 <135>       printString
  4 <134>         colour: 'red'
  5 <134>         colour: 'red' »» a red trafficLight
  6 <135>         colour
  7 <134>         red »» a red trafficLight
  8 <135>         colour »» 'red'
  9 <134>       amber
 10 <134>         colour: 'amber'
 11 <135>         printString »» 'a red trafficLight'
 12 <134>         colour: 'amber' »» a amber trafficLight
 13 <134>       amber »» a amber trafficLight
 14 <134>       green
 15 <134>         colour: 'green'
 16 <134>         colour: 'green' »» a green trafficLight
 17 <134>       green »» a green trafficLight
 18 <134>       red
 19 <134>         colour: 'red'
 20 <134>         colour: 'red' »» a red trafficLight
 21 <134>       red »» a red trafficLight
 22 <134>   cycle »» a red trafficLight
```

Figure 8.3: A Trace View with Multiple Processes

Figure 8.3) continue to function in the presence of parallelism, even if the displays they present appear somewhat odd. More importantly, top level events will only be sent when an object activity status changes from quiescent to active, or vice versa, so views relying on top level events for synchronisation can operate without problems.

Most of Tarraingím's views do not present information about the structure of any concurrent processes executing the target program (§6.7). For example, the trace views elsewhere in this thesis do not include process identifiers. Given that most SELF programs are single-threaded, presenting process information would be an unnecessary complication. Information about an event's process is always available as the event's process parameter (§8.3.2), and views may use this to visualise the process structure of the application program. If necessary, views or watchers can request notification of all events from a controller, and then maintain their own value for depth on a per-process basis.

8.2.3 Meta-Depth

Tarraingím's views depend upon information about the actions (the message sends) within the target program. Views also send callbacks to objects in the target program — to retrieve information or execute user commands. These callbacks are simply message sends that are executed by the objects in the target program, so their actions will be detected in the same way as the actions of the actual target program.

Callbacks and monitoring can thus interact in two ways which have the potential to cause problems for Tarraingím. First, views are generally used to display the actions of the target program, not the incidental effects of the visualisation. For example, if a trace view includes events generated from callbacks in its display, it gives a misleading impression of the target program. Second, if a view sends callbacks in response to receiving events, the callbacks could cause events to be sent back to the view, which in turn could cause the view to send more callbacks. This can easily result in an infinite recursive loop (or, as described by *The New Hacker's Dictionary* [177], Tarraingím enters *sorcerer's apprentice mode*).

To avoid these problems, Tarraingím maintains a value for the *meta-depth* of a process or an event, which measures the amount of recursion within the system. Meta-depth can be used to determine whether an event is caused by the target program or is the result of some action within the monitoring system.

Processes and Callbacks

Tarraingím maintains a meta-depth value for each process in the SELF system. This is stored in the metaDepth slot of each SELF process object. When an event occurs, the event's metaDepth parameter is

set to the value of the current process's metaDepth slot. The metaDepth of the target program processes is set to zero, and thus events caused by the target program have a metaDepth parameter of zero.

When views and watchers send callbacks to query objects within the target program, the callback process's metaDepth (and thus the metaDepth of any events caused by the callback) is incremented, so that it is greater than zero. By default, encapsulators ignore events generated by processes with a metaDepth greater than zero, so views are not notified about events occurring as the result of these callbacks (§9.1.4).

Command callbacks are used to alter the target program, rather than passively retrieve information (§6.6.2). When the user edits information in a view, callbacks are sent back to the program to update the underlying objects. Tarraingím similarly uses command callbacks to update the intermediate models maintained by indirect watchers (§7.4.1).

Command callbacks can change the state of the target program, and as such, have to be treated as part of the program's execution. A view which ignores events generated by command callbacks will not correctly reflect the state of the target program. Command callbacks are therefore executed as if they were part of the target program, i.e. at a metaDepth of zero, so events generated by these callbacks will be monitored by the rest of the system.

Avoiding Recursion Within Tarraingím

A process's meta-depth is also used to control recursion within Tarraingím. Encapsulators manipulate meta-depth as follows:

- A process's metaDepth is *always* incremented when entering an encapsulator. Since controls, watchers, views, and callbacks generally execute in the current process, they will run at a metaDepth higher than that of the target program.

- Any other background processes supporting views (§5.2.6) execute at a metaDepth of one.

- Command callbacks generated in response to events, or any other process running at metaDepth of zero, should never send messages to the object generating the event or any other object that could send messages to that object.

In this manner, meta-depth is managed more or less automatically. The responsibility for handling recursion within Tarraingím is placed onto the visualiser who wishes to write a watcher or view which sends command callbacks as a result of receiving an event.

Query callbacks used to retrieve information operate without any special handling in user code. As encapsulators increase the process's metaDepth, such callbacks always execute at a metaDepth of at least one, and thus do not generate events. Similarly, the metaDepth of background processes must be set correctly when they are created, but will be maintained automatically afterwards.

If query callbacks send messages to the target program which change it significantly, dependent views will not detect the change because they will not receive any events. This is an error, but the Tarraingím system will continue to function. It is the responsibility of the programmer writing the callbacks to detect this situation, and to use command callbacks instead (§6.1.2).

Command callbacks are the only difficult cases. If they are run in response to a user action, such as editing a view (§6.6.2), there is no possibility for recursion, and they may be sent without problems. On the other hand, if they are sent from within a watcher or view to handle a target program event, updating an intermediate database such as a profile or call tree for example, then the programmer must take care to avoid any circularity. When writing views or watchers using command callbacks, the visualiser must check for these cases, and act accordingly.

Monitoring Tarraingím

As part of debugging watchers, views and callbacks, the visualiser may need to monitor objects belonging to Tarraingím itself. Similarly, the user may wish to monitor *all* the side effects monitoring has upon the

target program, including all query callbacks. We call this *reflexive* visualisation, because Tarraingím is being used to visualise its own execution.

Tarraingím supports reflexive visualisation by allowing a process's metaDepth to exceed one. A metaDepth of zero is used by the target program, a metaDepth of one is used by Tarraingím to monitor the target program, and metaDepths greater than one are used to monitor Tarraingím's monitoring. We call events with a non-zero metaDepth *meta-events*.

Meta-events are ignored by most watchers, but custom watchers can be written by the visualiser to accept events with an arbitrarily large metaDepth — a leaf watcher can request meta-events when it registers with its target object's controller (§8.1.2). The controller then configures its associated encapsulator to generate meta-events at the given meta-depth (§9.1.4). In this way, watchers, views and controllers can be displayed by Tarraingím.

8.2.4   Depth vs. Meta-Depth

Depth and meta-depth measure different things. Depth is maintained per object, and counts the number of active message sends (i.e. the depth of recursion) within an object. Meta-depth is maintained per process, and measures the depth of recursion within Tarraingím itself. Depth does not distinguish between messages sent from the target program, and those sent by Tarraingím, whereas this is the main purpose of meta-depth. Depth may be any natural number (depending upon the program), while meta-depth will typically be either zero (meaning the process is running the target program or command callbacks) or one (when the process is running within Tarraingím or performing query callbacks).

This is illustrated in Figure 8.4, which shows the changes in depth and metaDepth during the processing of the first few events of the trace in Figure 8.2. Note that depth increases throughout the example, whereas metaDepth alternates between zero and one. Whenever the target program is executing (lines 1, 8, 9, and 16 from Figure 8.4) the metaDepth is zero, and whenever the controller (lines 3, 4, 11 and 12) or event clients (lines 5 and 13) are executing, the metaDepth is one.

depth	meta depth		
0	0	1.	trafficLight receives the message cycle.
0	0	2.	encapsulator catches the message and increases metaDepth.
0	1	3.	encapsulator notifies the controller, which increases depth.
1	1	4.	receiptEvent for cycle is dispatched to watchers.
1	1	5.	watchers and views execute.
1	1	6.	watchers return to controller, and thus to encapsulator.
1	1	7.	encapsulator resets the process's metaDepth.
1	0	8.	target program continues execution of cycle message.
1	0	9.	cycle method sends red to self.
1	0	10.	encapsulator catches the message and increases metaDepth.
1	1	11.	encapsulator notifies the controller, which increases depth.
2	1	12.	receiptEvent for red is dispatched to watchers.
2	1	13.	watchers and views execute.
2	1	14.	watchers return to the controller, and thus to encapsulator.
2	1	15.	encapsulator resets the process's metaDepth.
2	0	16.	target program continues execution of red message.

Figure 8.4: Depth vs. Meta-Depth

8.3   Events

Events are packages of data about a particular occurrence within Tarraingím — generally an action of the target program. When events are created, their *parameters* are initialised to describe the occurrence that

caused them. Events are *dispatched* by controllers to any of their clients which have registered an interest in receiving them. A client *handles* events by inspecting their parameters and then taking appropriate action, such as updating a display or generating more events. Although events are an important part of Tarraingím's design (§5.3.2), the events themselves are quite simple objects. The purpose of events is simply to package information, and they have little behaviour of their own.

This section first outlines the various types of events (§8.3.1) and describes the parameters common to all events (§8.3.2). The uses and parameters of particular concrete types of events are then discussed (§8.3.3 – §8.3.5). Finally, the event dispatch protocol (used to dispatch events to their clients), and the client event protocol (used by clients to handle events) are described (§8.3.6).

8.3.1 Event Types

Tarraingím categorises events according to their type. An event's type reflects its origin, and describes the kind of occurrence the event records. Thus, encapsulators monitoring program actions generate programEvents, while the user interface responds to user commands by generating commandEvents. The various types of events are illustrated in Figure 8.5.

The leaves of the event hierarchy are concrete objects which are instantiated by the system, and the interior nodes are abstract objects which are used to organise the hierarchy but are never instantiated. Thus completedEvents and unwindEvents, which are generated by message return actions, inherit from abstract returnEvent object. Both these types of returnEvents and message receiptEvents inherit from the abstract programEvent object.
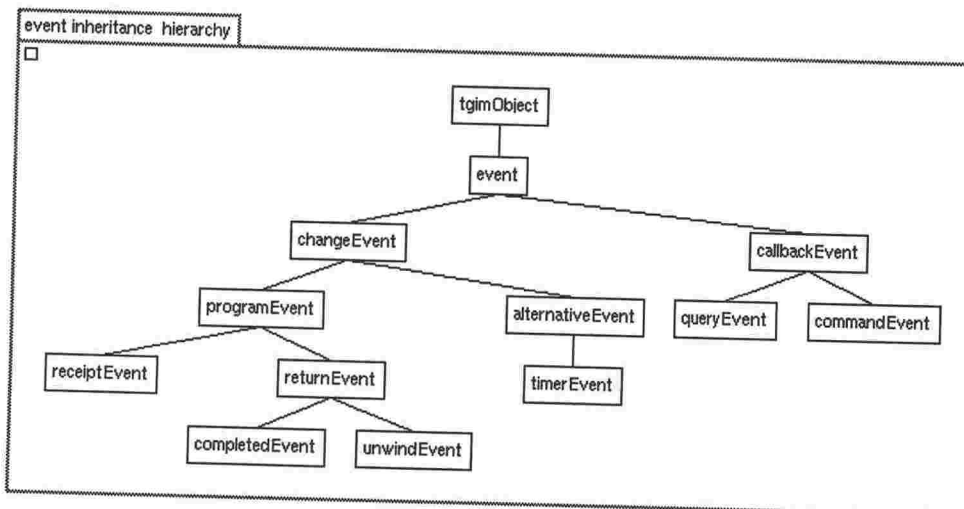
Figure 8.5: Event Types

There are two main categories of events — changeEvents and callbackEvents. Change events are routed up from the leaves of a watcher tree towards its associated view, while callback events are routed down the watcher tree from the view (§5.3.2).

8.3.2 Event Parameters

An event's parameters contain detailed information about the event. For example, when a receiptEvent is created to record a message receipt action, the event's parameters are initialised to describe the name of the message, the message's arguments, the object receiving the message, and so on. Some parameters are common to all events, whereas others depend on an event's type. Common parameters are defined in the abstract event object, and are thus inherited by other types of events.

The most significant event parameters are the type of the event, and the object where the event occurred. Events are further described by a name and some arguments, which provide more detail. The

precise meaning of an event's parameters depend on the event's type. For example, a receiptEvent's name parameter will contain the name of a message received by the object in the receiptEvent's object parameter, whereas the name of a callbackEvent is the name of a message to be sent to the object in the callbackEvent's object parameter. Table 8.4 lists the basic messages used to manipulate an event's parameters, and to create new events with particular parameter values.

event protocol

Accessing Parameters

type	The event's type.
object	The object where the event was observed.
name	The event's message selector.
arguments	An array of the arguments to the event.
at: n	Returns the event's n'th argument.
process	The SELF process that caused this event.

Creating Events

copyFor: o	Copy the event, changing its object parameter to o.
copyName: n	Copy the event, changing its name to n.
copyWith: a	Copy the event, changing its first argument to a.

Table 8.4: Event parameters protocol

8.3.3 Program Events

Program events are generated by encapsulators in response to the target program's actions. There are three concrete types of program events — receiptEvents, completedEvents, and unwindEvents. Both completedEvents and unwindEvents mark the return of a message (a completedEvent is a local return, while an unwindEvent is a non-local return); they are subtypes of the abstract returnEvent type. Program events are generated in pairs, with every receiptEvent eventually matched by a returnEvent (§9.1.2).

Program events inherit all the common parameters from the basic event object. The name and arguments parameters contain the name and arguments of the message causing the event, and the object parameter refers to the object where the event was detected.

Some parameters are unique to program events (see Table 8.5). In particular, the receiver parameter contains the value of self within the monitored message send. Note that in some circumstances (if the message is inherited) this may not be the same as the event's object parameter, as this specifies the object where the event was detected — that is, the encapsulator's target object (§9.1.3). The depth parameter measures the depth of recursion within the target object. The topLevel parameter is true whenever an event's depth is one (this is used for convenience in identifying top level events). The metaDepth parameter measures the amount of recursion within Tarraingím itself (§8.2).

Return events are program events generated whenever a message returns. They extend program events with two extra parameters (see Table 8.5). The returnValue parameter contains the value returned by the message. The local parameter specifies whether the event is a local return: local is always true for completedEvents and false for unwindEvents.

8.3.4 Alternative Events

Alternative events are change events generated by alternative strategies (§4.3.4). For example, timerEvents are generated by a timerWatcher when that watcher detects that a slot in its target object has changed (§7.2.5). The name and arguments of a timerEvent respectively contain the name and contents of the changed slot.

programEvent protocol	
receiver	The receiver of the message causing the event.
depth	The depth of the event. (§8.2.1)
topLevel	true if this event is a top level event.
metaDepth	The metaDepth of the event. (§8.2.3)

returnEvent protocol	
returnValue	The value returned by the message.
local	true if the event is a local return.

Table 8.5: Program Event parameters

Although Figure 8.5 shows only one concrete type of alternative events (timerEvents), more types are possible. New types of watchers are required if Tarraingím is extended with another strategy. If the events produced by these new watchers must be handled differently from the existing event types, new types of alternativeEvent will be required.

8.3.5   Callback Events

Callback events are sent from views and watchers down the watcher tree, and are used to implement callbacks sent to objects in the target program (§3.7). Callback events use the name and argument event parameters to describe the message to be sent to the target object.

There are two concrete types of callbackEvents: queryEvents, by far the most common, which are used to retrieve information from the target object using accessor messages (§4.2); and commandEvents, which are used to send mutator messages to change the target object in response to user commands (§6.6.2). The type of callback is chosen by the visualiser when designing views. A queryEvent callback is generated by a message sent to callback (as in the tlView implementation illustrated in Figure 6.5), while a commandEvent callback is generated by sending a message to command (as in the dots view updating method illustrated in Figure 6.13). The only practical difference between types of callbacks is the meta-depth at which they execute inside the target program (§8.2.3).

8.3.6   Dispatching and Handling

Once an event has been created, it must be dispatched to all interested clients. This is implemented by the dispatch message, which uses the event's object parameter's controller and dispatch database (§8.1.3). The controller calls sendTo to send the event to each interested client in its database. The sendTo message is also used to route events around the watcher tree.

event protocol	
dispatch	
dispatch	Dispatch this event to all interested clients via the event object's controller.
sendTo: client	Dispatch this event to client.

Table 8.6: Event dispatch protocol

A client needs to handle different types of events in different ways. For example, a trace view displays receipt events differently to return events (see the quicksort trace view in Figure 4.4). Event handling also differs between clients — a trace view will handle a return event by displaying it, while an indirect reference watcher may adjust the configuration of other views. In programming language terms this is a *multi-method* dispatch, as the required behaviour depends upon the types of both the client and the event [78].

8.3 EVENTS

SELF does not support multi-methods, but can simulate them using double-dispatching [106]. An event implements the sendTo: client message by sending itself to the client as the argument to another message, the name of which encodes the event's type. These messages form the client event protocol (see Table 8.7), and must be understood by all clients receiving events. Clients can implement behaviour which depends upon an event's type by implementing the appropriate message.

Client event protocol

event handling
tgimEvent: e	Handle the event e.
changeEvent: e	Handle the change event e.
programEvent: e	Handle the program event e.
receiptEvent: e	Handle the receipt event e.
returnEvent: e	Handle the return event e.
completedEvent: e	Handle the completed event e.
unwindEvent: e	Handle the unwind event e.
callbackEvent: e	Handle the callback event e.

...there is one method in this protocol for each event type illustrated in Figure 8.5.

Table 8.7: Client event protocol

Clients may need to handle several types of events in the same way. For example, views by default redraw themselves in response to all types of changeEvents (§6.1.1). Watchers by default forward changeEvents up and callbackEvents down the watcher tree (§7.1.3). This could be handled easily within a language with multi-methods because the multi-method lookup would consider the inheritance hierarchy of the event object as well as that of the view, whereas SELF's single-dispatch lookup considers only the receiver's type.

Tarraingím supports the grouping of events by providing a default implementation for the client event protocol which treats an event as if it belongs to its parent type. This is implemented by the eventClient object displayed in Figure 8.6. All watchers and views inherit from eventClient and often rely upon its behaviour. For example, the views described in Chapter 6 which explicitly handle events do so by defining a single changeEvent method (see Figures 6.8 and 6.11). All concrete changeEvents received by these views are eventually routed through this message.

```
traits eventClient = (|
    parent* = traits tgimObject.

    "top of hierarchy, do nothing"
        tgimEvent: e = (self).

    "handle events by dispatching to parent type"
        changeEvent: e = (tgimEvent: e).
        programEvent: e = (changeEvent: e).
        receiptEvent: e = (programEvent: e).
        returnEvent: e = (programEvent: e).
        completedEvent: e = (returnEvent: e).

    "... there is one method in this object for each event handling message"
|)
```

Figure 8.6: Client Protocol Mixin

8.4   Summary

This chapter has presented the controller and event objects, which are used to gather information about the target program, and route it around Tarraingím's framework. To summarise the chapter, Figure 8.7 lists the sequence of operations used to process a single receipt action.

1. The foo message is sent to the target object.

2. The encapsulator attached to the target object intercepts the message's receipt.

3. If the current process's metaDepth is less than or equal to the encapsulator's metaDepth (§8.2.3):

   (a) The encapsulator increases the current process's metaDepth by one (§8.2.3).

   (b) The encapsulator sends the message's name (foo), any arguments, and other parameters (the receiver, self), to its associated controller (§8.1.3).

      i. Because the event is a message receipt, the controller increases its depth by one (§8.2.1).
      ii. The controller searches the tables in its dispatch database for clients interested in the foo message (§8.1.3).

         • If the current process's metaDepth is greater than one, the database's meta events table is searched.
         • If the controller's depth is one, the database's top level events tables are searched.
         • Otherwise, the database's general tables are searched.

      iii. If any interested clients are found, a receiptEvent is created and its parameters initialised to describe the receipt action (§8.3.3).

         • The receiptEvent is dispatched to each interested client in turn (§8.3.6).
      iv. The controller returns control to the encapsulator.

   (c) The encapsulator decreases the current process's metaDepth by one (§8.2.3).

4. The encapsulator interception method returns.

5. The target object begins executing the foo method.

Figure 8.7: Handling a Message Receipt Action

29. For systems, the analogue of a face-lift is to add to the control graph an edge that creates a cycle, not just an additional node.

Alan Perlis, *Epigrams On Programming* [168]

9

Encapsulators

Tarraingím's encapsulator objects form the core of the monitoring subsystem. An encapsulator monitors the actions of a single object within the target program, and passes this information to its controller. Encapsulators' requirements were discussed in Section 5.1.1. The target program must be monitored on a per-object basis, and only those actions of monitored objects of interest to the controller's clients should be monitored. The monitoring should be efficient, and an object must be able to be monitored dynamically. The user should be unaware of the monitoring.

This chapter presents our experiments with using encapsulators to monitor SELF programs. The first section presents the basic encapsulator design, and describes how this design can be adapted to work within SELF. The second section describes how the basic encapsulator design suffers from the *self problem* [127], then presents several alternative encapsulator designs which avoid this problem. The third section describes how encapsulators handle primitive operations and messages implemented directly within the SELF compiler. The chapter concludes with a summary of our work with encapsulators.

9.1 The Design of Encapsulators

Encapsulators monitor the actions of a particular object in the target program by postprocessing the target program's structure (§2.5.2). The techniques behind encapsulators and their basic design were first developed by Pascoe in SMALLTALK [164]. Pascoe described several experimental applications of encapsulators, such as implementing monitors and atomic objects for concurrency control, and linking models to views in SMALLTALK's MVC interface framework [122]. The techniques behind encapsulators have been used to build proxy objects in several distributed SMALLTALK systems [16, 138] and object oriented databases [174]. Encapsulators have also been used for tracing and debugging SMALLTALK programs [121], and providing general reflexive facilities [75].

9.1.1 Attaching an Encapsulator to an Object

Figure 9.1 illustrates the basic design of an encapsulator. The target object is displaced from the program and replaced by an encapsulator. All the objects in the program which originally referred to the target object now refer to the encapsulator, and therefore any messages sent to the target object arrive instead at the encapsulator. When the encapsulator receives a message, it notifies its controller of the message receipt, then resumes the program by forwarding the intercepted message to the displaced target object. When the forwarded message returns, the controller is again informed and the program continued. Thus

the program executes as if the encapsulator was not present, and the controller is notified of all messages the target object receives.
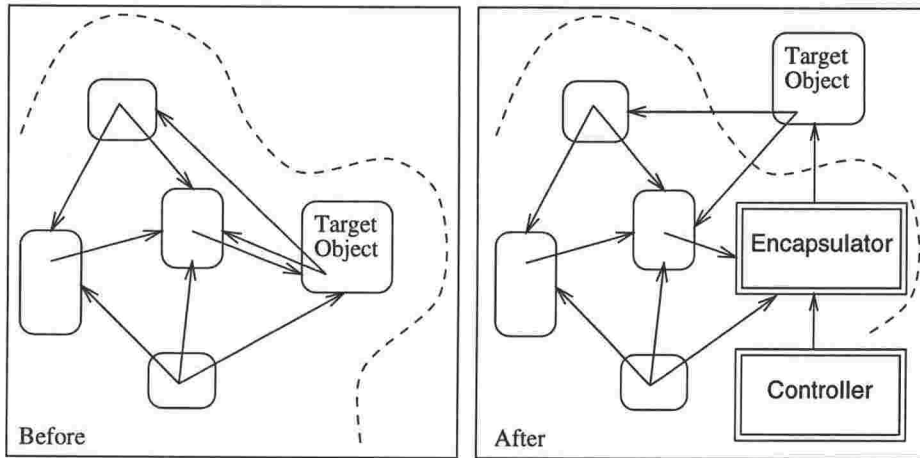

Figure 9.1: Monitoring an Object with an encapsulator

Displacing the Target Object

In order to monitor the actions of an object, an encapsulator must first displace the target object from the program, while privately retaining it in order to be able to continue the execution of the program. In other words, the encapsulator must assume the target object's *identity*, while the target object's *slots* must be preserved with a different identity (§5.4.1).

Pascoe's encapsulators are attached with SMALLTALK's become operation, which exchanges the identities of two objects. Using become, an encapsulator can be simply substituted for the target object. Traditional SMALLTALK implementations refer to objects indirectly via an *object table*, so become can be implemented very efficiently by simply swapping the object table entries referring to each object [85]. SELF [44], like more modern SMALLTALK implementations [60], refers to objects directly via pointers, and implements become by an exhaustive search-and-replace operation, replacing all references to one object with references to another.

The search-and-replace technique is used to attach Tarraingím's encapsulators to SELF objects. A slot by slot copy is made of the target object, then the original target object is replaced by the encapsulator — the encapsulator effectively assumes the target object's identity. The search-and-replace operation is much slower than a simple become, because the program's entire memory image must be examined.

Primitives

Some SELF objects are *primitive* objects, which are implemented directly by the SELF compiler. Section 9.3.1 describes how Tarraingím's encapsulators monitor primitive objects.

9.1.2   Intercepting Actions

Figure 9.2 shows a SELF[1] version of the original encapsulator object. This figure is adapted from Pascoe's SMALLTALK encapsulator [164]. The encapsulator contains two variable slots: controller contains the encapsulator's controller, and target refers to the displaced target object. The encapsulator also contains a single method, undefinedSelector: msg Arguments: args, which performs the actual catching and forwarding of messages.

---

[1] This figure is not quite "real" SELF — in particular, the undefinedSelector message should have several more parameters. These parameters are not used by encapsulators, and so have been omitted.

```
encapsulator = (|
"variables"
  _ target.  "holds the encapsulator's displaced target"
  _ controller.  "holds the encapsulator's controller"

"this will be called to handle all messages the encapsulator receives"
  ^ undefinedSelector: msg Arguments: args = (|rv|
       controller encapReceipt: msg Arguments: args.
       rv: msg sendTo: target WithArguments: args.
       controller encapReturn: msg Arguments: args ReturnValue: rv.
       rv).
|)
```

Figure 9.2: A Basic Encapsulator

The encapsulator in Figure 9.2 intercepts all messages it receives by using the *undefined selector* exception that is raised when a SELF object receives a message that it does not understand (§5.4.4). The encapsulator does not define any messages itself, so it does not understand any messages it receives, apart from messages accessing the target and control variables (§9.1.4). Whenever the encapsulator receives a message, SELF's message lookup algorithm sends the undefinedSelector message to the encapsulator to signal the error. The parameters msg and args of the undefinedSelector method give the name and arguments of the intercepted message. Defining only undefinedSelector allows the encapsulator to intercept all the messages it receives with a single generic method definition.

The encapsulator's undefinedSelector method first sends an encapReceipt message to notify its controller of a message interception, including the message's name (msg) and arguments (args). The intercepted message is then sent to the displaced target (using the sendTo:WithArguments: message, SELF's version of Smalltalk's perform), and its return value is stored in the encapsulator's local variable rv. The message's return is communicated to the controller with an encapReturn message, which also sends the value returned by the message. Finally, the undefinedSelector method itself returns rv, the value returned by the intercepted message.

Top Level Events Only

Pascoe's basic encapsulator of Figure 9.2 intercepts all messages sent to its (now displaced) target object, from other objects in the target program. Once a message has been intercepted, it is sent directly to its target object, which then processes the message without reference to the encapsulator. Pascoe's encapsulator therefore only intercepts *top level* events (§8.2.1). Many of Tarraingím's watchers and views require information about all of their target object's actions.

Efficiency

Pascoe's encapsulators are notably inefficient, for two reasons. First, they catch *all* the top level messages the target object receives. Many strategies (§4.3) and watchers (§7.2.3) only require information about a few particular messages. The controller's dispatch database (§8.1.3) ensures that watchers only receive events that the watchers have requested, but monitoring unnecessary events imposes an overhead on the target program. Second, the undefinedSelector exception mechanism is slower than a standard message receipt. The encapsulator must be unsuccessfully searched for a slot matching the original message name and the message's name and arguments must be specially packed into SELF objects, before the undefinedSelector method can be called.

Non-Local Returns

A single message send executed by the target program consists of two actions: a message receipt action and a message return action (§4.3.3). An encapsulator notifies its controller as soon as a message is intercepted, and again when the message has returned from the target object. These events therefore always occur in nested matching pairs. Every receipt event (marking the beginning of a message send) will eventually be matched by a return event (when a given message send terminates). Most returns are local returns, which occur whenever a method completely executes its body. SELF also includes a return operator which can return *non-locally* from arbitrarily many message sends, unwinding the stack as required (§5.4.3).

Non-local returns cause several problems for encapsulators. Should a forwarded message return non-locally from within the target object, it will return non-locally through an encapsulator, without notifying the encapsulator's controller. The controller, therefore, will not send events to watchers and views, and will not correctly update its depth value (§8.2.1). In Figure 9.2, a non-local return from the target object would cause the sendTo message to return directly. The rest of the undefinedSelector method (the controller's notification via encapReturn, and return of rv from the method) would be ignored.

Non-local returns can be intercepted in SELF. An expression may have an associated exception block: if the expression returns non-locally, the exception block is evaluated as the stack unwinds. An encapsulator can use this facility to trap non-local returns. Whenever a message is sent to the target object an exception block can be established. If the send performs a non-local return, the exception block is executed. This notifies the controller that the message is returning non-locally, and includes information about the value being returned. The program will then continue with the non-local return.

Inheritance

Objects in SELF can inherit from other objects (§5.4.4). For example, the stack object in Figure 5.7 inherits from the traits stack object. Similarly, the trafficLight object in Figure 6.4 inherits from traits trafficLight. Since SELF is a prototype-based language, traits stack and traits trafficLight are completely normal SELF objects (§9.2.1), and encapsulators can be attached to them. Attaching a Pascoe style encapsulator to an inherited object will sever the inheritance link, and this may adversely affect the target program.

The problem of monitoring, or otherwise manipulating, inherited objects in a prototype-based language is known as the *split object* problem [63]. SELF's traits objects correspond to classes in languages such as SMALLTALK. Like a class, any particular trait only defines part of an object — the complete definition of the object is split across the object itself, and all traits objects from which it inherits. The split object problem does not arise in class based languages, because the partial definitions exist only as classes. In a prototype language, partial objects (traits) are indistinguishable from any other objects, and can be manipulated without regard for the other objects which they help define.

The split object problem has not affected Tarraingím in practice. This is because inherited objects do not really represent abstractions in the target program. Like classes, inherited objects instead represent a partial definition. SELF programs do not manipulate traits objects directly, instead traits are inherited by the concrete objects actually making up the program. It is these concrete objects, rather than the partial inherited objects, which Tarraingím needs to monitor.

Primitives

Some SELF messages are *primitive* messages, which, like primitive objects, are implemented directly by the SELF compiler. Section 9.3.2 describes how Tarraingím's encapsulators monitor primitive messages.

9.1.3   Notifying the Controller

When an encapsulator intercepts an action of the target program, it transmits data about the action to its associated controller. The controller packages the data into an event and distributes the event to the

rest of the system. In the current version of Tarraingím, this notification is sent using an internal protocol between encapsulators and controllers. The encapReceipt and encapReturn messages from Figure 9.2 are part of this protocol.

The data sent are the name of the message, the message's arguments, and (for a return event) the message's return value. Tarraingím's encapsulators also collect some extra information, most notably self, the receiver of the intercepted message (§8.3.3). If the encapsulator monitors traits objects, self may not be the same as the target object. Other event parameters, such as the depth of message sends within the target object, are generated within the controller (§8.2.1).

An encapsulator's undefinedSelector method is executed by the target program process which sent the message intercepted by the encapsulator. The target program is effectively suspended until the encapsulator returns. This process is also used by the controller to dispatch the generated events to interested watchers or views. In this way, the target program and the monitoring system remain synchronised while events are being dispatched (§8.1.3).

9.1.4 Reflexive Monitoring

In order for an encapsulator to intercept all the messages it receives, it must not implement any messages it could receive from the target program. The names of an encapsulator's local variables and methods must therefore be chosen to avoid matching any messages used in the target program [164]. Tarraingím therefore adopts the convention that all encapsulator local names and messages are required to begin with the prefix "e3_" (the "3" is for our third implementation of encapsulators). For this reason, Tarraingím's encapsulator's slots named "e3_Target" and "e3_Controller" rather than "target" and "controller".

Slot names prefixed with "e3_" are legal SELF, but unusual style and therefore unlikely to appear in any other program. An encapsulator cannot itself monitor messages with these names, since if it receives a message with the same name as one of its local slots, the local operation contained in the slot will be carried out, rather than the message being monitored. An encapsulator cannot therefore be monitored by another encapsulator.

Care is needed if other objects are used in an encapsulator's implementation. Ideally, an encapsulator should be able to monitor any object in the SELF world. If an object used in the implementation of encapsulators is monitored, the monitoring of the object may itself be detected by the monitoring system, resulting in infinite recursion. To avoid this problem, we minimise the use of other objects in encapsulators' implementation, and where this is unavoidable, arrange that encapsulators will not detect their use.

For example, an encapsulator compares objects using the primitive message _Eq, rather than the SELF language message ==, and integers are manipulated with primitives _IntAdd and _IntEq rather than + or = (§9.3.2). Since primitives are handled directly and no SELF code is invoked, an encapsulator will not detect such operations. Where SELF code is unavoidable (for example, all SELF control structures are implemented within the language), it is duplicated and rewritten specially for encapsulators, following the naming convention for encapsulators.

These restrictions upon the programming style used within an encapsulator provide the major technical reason for separating controllers from encapsulators. Controllers, being in all respects normal SELF objects, do not have these limitations. Moving functions from encapsulators to controllers minimises the size of encapsulators, and reduces the possibility of errors in the encapsulators' implementation accidentally breaching these restrictions. Separating encapsulators and controllers also increases the amount of the Tarraingím system it is possible to visualise reflexively, as controllers can be monitored, while encapsulators cannot.

Meta-Depth

An encapsulator can monitor the actions of any standard SELF object. It should therefore be possible to use encapsulators to reflexively monitor Tarraingím, especially watchers, controllers, and views. As described in Section 8.2.3, reflexive monitoring is controlled by processes' and encapsulators' metaDepth

parameters. In particular, Tarraingím's encapsulators have an e3_metaDepth slot, which is also usually zero. Before an encapsulator dispatches any event to its associated controller, it compares the current process's metaDepth with its local e3_metaDepth slot. If the current process's value is less than or equal to the encapsulator's value, the process's metaDepth is increased and the event is dispatched to the encapsulator's controller (§8.4). If the process's metaDepth exceeds that of the encapsulator, the event is ignored, and the target program continued without dispatching through the controller.

Maintaining metaDepth in encapsulators allows controllers to be monitored, but not encapsulators. This is partly a necessary consequence of our implementation of encapsulators, in that some object has to maintain metaDepth, and that object cannot itself be protected by the metaDepth mechanism. This is also a side-effect of the subsystem's overall design goals. Encapsulators are supposed to be hidden from the user, as well as from SELF's own reflexive facilities. The interface to the monitoring subsystem is through controllers, not encapsulators. Tarraingím's model of program monitoring is that the controller, when requested, generates events relating to the target object, but the target object itself is not changed. Encapsulators are simply the implementation mechanism used by controllers. Tarraingím's reflexive facilities may be monitored by the usual meta-recursive "tower" [134] — an encapsulator can be attached to a controller, then a second encapsulator can be attached to the first encapsulator's controller, a third encapsulator to the second encapsulator's controller, and so on *ad infinitum*.

9.2   The Self Problem

Pascoe's encapsulator, described in Section 9.1.1, has two practical shortcomings. First, the displaced target object can easily "escape" from the encapsulator, that is, references to the target object can be passed outside the encapsulator. Obviously, if the target object can escape, the encapsulator cannot monitor its operation. Second, Pascoe's encapsulator only intercepts top level message sends (§8.2.1). Throughout this thesis, we have assumed that the monitoring system is able to gather information about all the target object's actions, not just those occurring at the top level — for example, the quicksort trace example from Section 4.3.5 displays all the message sends involved in quicksort. Many of the strategies discussed in Chapter 4 and the watchers from Chapter 7 likewise require information about the programs actions at arbitrary depths.

These shortcomings are symptoms of the *self problem* [127], a fundamental problem in the design of object oriented languages. This section describes the self problem in detail, and presents three alternative designs for encapsulators which avoid the self problem.

9.2.1   Delegation and the Self Problem

The self problem describes a difficulty many object oriented systems have in implementing delegation [127]. The problem arises in the binding of self, when messages are sent between objects.

In a normal method invocation, self is bound to the current object, that is, the object that received the message. Methods can refer directly to self, for example, to pass the current object as an argument to another message, or to store it in a global variable. Self-sends send messages to self (§5.4.2) to implement recursive functions or procedural decomposition, and to access variables. If a message is sent to another object, self is rebound to the new message's receiver, as that object responds to the message.

Forwarding

Message forwarding is illustrated (in the context of encapsulators) in the first frame of Figure 9.3. When a message m arrives at an encapsulator, self within the encapsulator will be bound to the encapsulator itself. As m is forwarded to the displaced target object, self is rebound so it refers to that object. At this point, if the target object executes a self-send n, the message is sent to the object referred to by self, i.e., the target object, and the encapsulator will not intercept the message. If the object passes self to another object, or stores it in a global variable, other objects could directly access the target object [75]. Pascoe's encapsulators use forwarding, and can therefore be bypassed, even though all existing references to the target object are changed to refer to the encapsulator when the encapsulator is attached (§9.1.1).
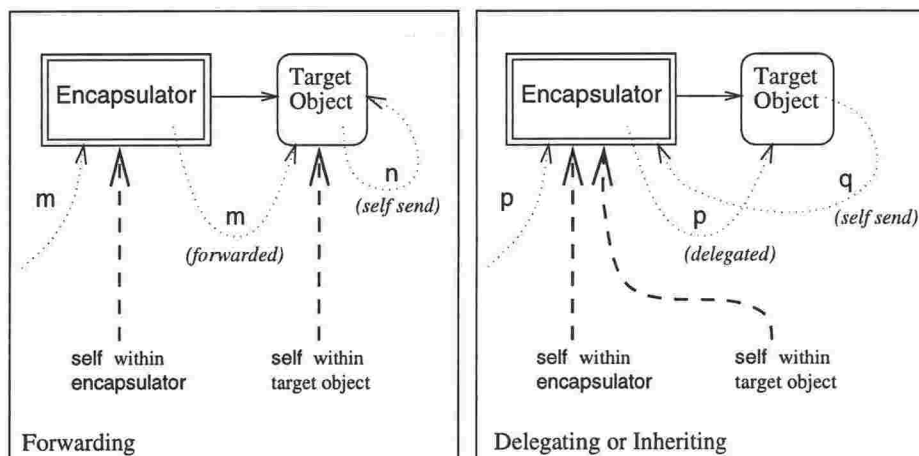
Figure 9.3: The self Problem

Delegation

The second frame of Figure 9.3 shows how the binding of self does not change when a message is delegated to another object, rather than being forwarded with a normal message send. The message p is intercepted by the encapsulator, and as it arrives as a normal message send, self is bound to the encapsulator. The encapsulator then delegates the message to its target object, rather than forwarding it. Thus self is not rebound, and continues to refer to the encapsulator. If a self-send q is executed by the target object, the message is received by the encapsulator, since this is the current value of self. Similarly, if the target object transmits or stores self, it will use a reference to the encapsulator, rather than a reference to the displaced target object.

The difference between forwarding and delegation is in the type of message send used to pass the message from the encapsulator to the target object. Forwarding uses a standard message send, while delegation uses a special type of send which does not rebind self [127]. The self problem occurs when the semantics of delegation are required, but only forwarding is available. Since most OO languages only support forwarding, the self problem is quite common.

Inheritance

Most class-based object oriented languages provide inheritance between objects and classes and between related classes. The message *resend* operations used in class inheritance are equivalent to a static form of delegation [209, 129]. When an object in a class-based language receives a message, a method in the object's class will be located, then invoked with self bound to the object that received the message. The message may be subsequently resent to another class, for example, using SMALLTALK's super send, or CLOS's call-next-method. A resend invokes a method in the new class, which is typically an ancestor of self's class, but the binding of self is not changed.

A message resent between classes is thus treated like a message delegated between objects. For example, the second frame of Figure 9.3 could also describe an encapsulator which inherits from its target object. The encapsulator receives a p message, and resends this to its target object: self remains unchanged. If the target object executes a self-send of q, the method lookup algorithm begins its lookup with self, i.e., the encapsulator.

Delegation and Inheritance in SELF

Classless languages such as SELF do not distinguish between classes and instances, rather, all objects may perform either rôle. Inheritance in SELF occurs between objects, but behaves like inheritance between

classes in typical object oriented languages, and therefore much like delegation. SELF's inheritance is implicit, that is, objects automatically resend any messages they do not implement to any objects they inherit from, while delegation is usually explicit.

SELF also provides some experimental support for explicit delegation. Our SELF version of Pascoe's encapsulator uses the sendTo operation (in Figure 9.2) to forward the intercepted message to the target object. Using sendTo performs a normal message send, and so alters the binding of self: this type of encapsulator suffers from the self problem. SELF also provides a sendTo:DelegatingTo message, which implements a *delegated perform*. The sendTo:DelegatingTo message takes an extra argument, to which self is bound for the duration of the send. Table 9.1 compares the sendTo and sendTo:DelegatingTo messages.

---

message protocol
_____

Message sending
    sendTo: obj   Forwards a message to obj. The message lookup algorithm begins at obj, and self is bound to obj while the resulting method is executed.

    sendTo: obj WithArguments: args

                                  Forwards a message to the object obj with args as arguments. The method lookup algorithm begins at obj, and self is bound to obj while the resulting method is executed.

    sendTo: obj DelegatingTo: del WithArguments: args

                                  Delegates a message to the object del with args as arguments. The message is actually sent to obj. That is, the method lookup algorithm begins at del, but self is bound to obj while the resulting method is executed.

Table 9.1: Message sending protocol

We have experimented with several alternative designs for encapsulators which use SELF's support for delegation and inheritance to avoid the self problem. Delegating encapsulators (§9.2.2) are very similar to Pascoe's forwarding encapsulators, but use the sendTo:DelegatingTo message to delegate messages to their target objects. Inheriting encapsulators (§9.2.3) use inheritance, and since they must inherit from their target object, they can no longer use the undefinedSelector exception to intercept messages. Custom encapsulators (§9.2.4) are an optimised development of inheriting encapsulators.

9.2.2 Delegating Encapsulators

Delegating encapsulators avoid the self problem by delegating messages to their target object. A SELF implementation of a delegating encapsulator is illustrated in Figure 9.4. Like the basic encapsulator of Figure 9.2, it intercepts all messages by defining only an undefinedSelector method. Once intercepted, messages are delegated to the displaced target object, using the sendTo:DelegatingTo message. This ensures that the delegated messages are invoked with self bound to the encapsulator. Any references to self within the target object refer to the encapsulator, and any self-sends are received by the encapsulator. The target object is prevented from escaping the encapsulator, and the encapsulator can monitor all its self-sends.

Privacy

Slots in SELF objects may be either *public* or *private* (§5.4.1). Public slots can be accessed from any other object, while private slots can only be accessed from "the inside" of an object.

An encapsulator can be attached to an object containing private slots. This poses no problem for Pascoe's forwarding encapsulators, since they only intercept top level messages, which by definition must originate from the outside of an object and so be sent to public slots. Delegating encapsulators also intercept the target object's self-sends, however, and self-sends can be used to access an object's private slots, for example, to read or write a private variable. A delegating encapsulator is always considered

```
delegating encapsulator = (|
  "variables"
  _ target.
  _ controller.

  "this handles messages the encapsulator receives"
  ^ undefinedSelector: msg Arguments: args = (|rv|
      controller encapReceipt: msg Arguments: args.
      rv: msg sendTo: self DelegatingTo: target WithArguments: args.
      controller encapReturn: msg Arguments: args ReturnValue: rv.
      rv).
|)
```

Figure 9.4: Delegating Encapsulator Design

outside its target object, because there is no inheritance relationship between the two, so the encapsulator cannot access the target's private slots. Whenever the target sends a self-message to access one of its private slots, the message will be successfully intercepted by the encapsulator, but cannot be delegated back to the target, because the encapsulator does not have access to the private slot. For this reason, delegating encapsulators cannot be used in versions of SELF which enforce privacy (§9.2.5).

Inheritance

Delegating encapsulators also fare better than forwarding encapsulators if other objects inherit from their target object. Although no explicit inheritance link is built from a delegating encapsulator to its target object, the delegated send can mimic inheritance in many respects. Some objects will still not function correctly if a delegating encapsulator is attached to an object from which they inherit. In particular, if an object inherits from two or more objects, the lack of an explicit inheritance link may cause SELF's message lookup algorithm to go awry, and if an object itself defines an undefinedSelector method, it may override the definition used by the encapsulator.

9.2.3   Inheriting Encapsulators

The design of inheriting encapsulators (Figure 9.5) is quite different from the design of forwarding (Figure 9.2) or delegating (Figure 9.4) encapsulators. As with the previous designs, an inheriting encapsulator contains a controller variable, which refers to its associated controller, and a target variable, which refers to its displaced target object. In an inheriting encapsulator, the target is a parent slot, and the encapsulator inherits from its target object via this slot.

The major difference between the designs is that an inheriting encapsulator does not use an undefinedSelector method to intercept all incoming messages. This is replaced by a large number of wrapper methods, each of which intercepts one particular message.

When a message is sent to a forwarding or delegating encapsulator, SELF's message lookup algorithm searches the encapsulator for a slot implementing the message. Of course, such an encapsulator does not define any messages (except undefinedSelector), so an error is signalled by sending the encapsulator the undefinedSelector message. An inheriting encapsulator inherits from the target object, so when it receives a message, the lookup algorithm will search the target object for any messages the encapsulator does not define. An inheriting encapsulator must therefore explicitly implement all the messages it needs to capture. If an inheriting encapsulator does not define any messages, that message will be passed directly to its target object.

All messages sent originally to the target are received by the encapsulator and the appropriate wrapper method run. The body of a wrapper method is similar to a delegating encapsulator's undefinedSelector

```
inheriting encapsulator = (|
   _ target*. "dynamic inheritance from target object"
   _ controller.

 "wrappers for messages"
    ^ cycle = ( | rv ← nil. |
       controller encapReceipt: 'cycle'.
       rv: resend.cycle.
       controller encapReturn: 'cycle' Returning: rv.
     rv).
    ^ colour: n = ( | rv ← nil. |
       controller encapReceipt: 'colour:' With: n.
       resend.colour: n.
       controller encapReturn: 'colour:' Returning: rv With: n.
     rv).
    ^ colour = ( | rv ← nil. |
       controller encapReceipt: 'colour'.
       rv: resend.colour.
       controller encapReturn: 'colour' Returning: rv
     rv).
     "..."
|)
```

Figure 9.5: Inheriting Encapsulator Design

method. The wrapper first notifies the controller of the message receipt. The intercepted message is then resent to the target object. This uses the SELF resend operator which resumes the message lookup while keeping self bound to the initial message receiver. As the encapsulator inherits from the (displaced) target object, this continues the target program. The return value from the resend is stored in the local variable rv. The controller is then notified of the message's return, and of the value returned by the message. The wrapper method then returns normally, passing the stored return value to the original sender of the message in the target program.

Wrapper Methods

An inheriting encapsulator needs to contain a wrapper method for each message it intercepts. To completely monitor the target object, it must be able to intercept all messages implemented by the target — either directly, or inherited. As SELF objects can change their inheritance structure dynamically (§5.4.4), the messages implemented by an object cannot be determined by a simple static inspection. Inheriting encapsulators therefore contain a wrapper for every message name defined in the SELF system. They are thus assured of intercepting any messages their target object actually implements. All inheriting encapsulators share the same set of wrapper methods.

There are over 7000 message names used in the SELF library, Tarraingím, and the examples from this thesis. Since writing 7000 wrapper methods would require a large effort, and the actual set of methods required depends on the particular target program, Tarraingím automatically constructs the wrapper methods. Every message name in the SELF system is stored as a canonicalString, SELF's version of LISP's atoms or Smalltalk's symbols. Tarraingím enumerates all the canonicalStrings known to the SELF system, and constructs wrapper methods for those which meet the syntactic criteria for SELF message names.

The wrapper methods must be regenerated whenever a new message name is added to the SELF system. We believe new wrapper methods could easily be created incrementally, however we have not yet implemented this optimisation. In practice, we rebuild the wrapper messages approximately once a month, or after any major change to Tarraingím or to the example programs.

The SELF compiler limits the number of slots an object can contain to approximately five hundred. Inheriting encapsulator's wrapper methods are therefore split across several objects, and shared by all inheriting encapsulators. SELF's prioritised multiple inheritance [215] is used to ensure the wrapper methods and the target object are searched in the correct order by the message lookup algorithm.

Evaluation of Inheriting Encapsulators

Inheriting encapsulators have several advantages over delegating encapsulators, although both types capture essentially the same information. The specialised wrapper methods used to intercept messages should be quicker than a generic undefinedSelector handler. The explicit inheritance link, maintained by inheriting encapsulators to the target object, ensures they are considered "inside" their target objects, and avoid delegating encapsulators' privacy problems. Inheriting encapsulators also avoid many of the problems delegating encapsulators have with inheritance — they do not disrupt the inheritance hierarchy, and they do not depend upon undefinedSelector.

An inheriting encapsulator's wrapper methods will almost certainly define some messages which are not defined by its target object. If other objects inherit from the encapsulator's target object, these extra message definitions may cause ambiguous message lookup errors.

9.2.4   Custom Encapsulators

Forwarding, delegating and inheriting encapsulators all monitor every message received by the target object. As many watchers only need a subset of these messages, this is quite inefficient (§8.1.2). Custom encapsulators are a development of inheriting encapsulators which contain wrapper methods only for those messages actually required by their controller, so that they do not intercept any unnecessary messages.

When a controller attaches a custom encapsulator to an object, the encapsulator does not contain any wrapper methods. As the controller is requested to monitor particular messages, the appropriate wrapper methods should be created and added to the encapsulator. A custom encapsulator's wrapper methods should be built automatically in the same way as an inheriting encapsulator's wrapper methods, but as this has not yet been implemented, for the purposes of our experiments we have written the wrapper methods manually.

Should the controller need to monitor all the messages the object receives (in practice, if more than a few messages must be monitored) the encapsulator can be converted into an inheriting encapsulator by simply including all the wrapper messages.

Custom encapsulators also function well in the presence of inheritance. Like inheriting encapsulators, they maintain an explicit inheritance link between themselves and their target object, but unlike inheriting encapsulators, they cannot introduce the possibility of a message lookup ambiguity, because they only include wrapper methods for messages actually understood by their target objects.

9.2.5   Summary

Figure 9.6 compares the four types of encapsulators described in this section. The columns list the types of encapsulator, the technique used to intercept messages, the technique used to send the intercepted message to the target object, and any problems to which encapsulators are susceptible.

Our choice of encapsulator has been determined in practice by the facilities of particular SELF versions. Tarraingím was initially developed with SELF versions 1.0 and 2.0, which supported prioritised inheritance and enforced slot privacy. Inheriting encapsulators, which avoid privacy problems, worked well within these versions. Tarraingím as described in this thesis uses SELF version 2.3, which adds support for catching non-local returns. An "encapsulator", as used in the rest of this chapter, and indeed throughout this thesis, therefore denotes an inheriting encapsulator, unless another type of encapsulator is specified.

SELF versions 3.0 and 4.0 do not support prioritised inheritance or privacy declarations. Inheriting encapsulators rely on prioritised inheritance, so they will not work with this version of SELF. The lack of

Type	Interception	Send to Target	Problems
Forwarding	undefinedSelector handler	message send	self, inheritance
Delegating	undefinedSelector handler	delegated send	privacy, inheritance
Inheriting	generic wrapper messages	resend	inheritance
Custom	specific wrapper messages	resend	

Figure 9.6: Types of Encapsulators

privacy declarations resolves the problems with delegating encapsulators described in Section 9.2.2, and we have verified that delegating encapsulators will perform satisfactorily in these versions of SELF.

The dependence of encapsulators upon the details of particular version of SELF highlights another advantage of the separation of controllers and encapsulators. Encapsulators are not visible outside the monitoring subsystem, as they are always accessed indirectly via controllers. Controllers therefore insulate the rest of Tarraingím from the details of encapsulators' implementations. The type of encapsulator used by Tarraingím can be changed without this change affecting any watchers or views.

9.3   Primitives

Some low level operations and data structures cannot be expressed efficiently in SELF, and are therefore supported directly by the SELF compiler and VM. Just as the SELF language provides objects and messages, so the VM provides *primitive objects* and *primitive methods*. Primitive objects represent data in the run-time system (such as integers, vectors or processes) or data provided by the operating system (for example TCP connections or X windows). Primitive messages implement fundamental operations for all types of objects (e.g., low-level copying or identity comparison) and specialised behaviour for primitive objects (e.g., adding two integers, scheduling a process, or opening a window).

To monitor a program, we need to be able to observe primitive objects and monitor primitive messages. This section begins by describing how encapsulators can be extended to handle primitive objects (§9.3.1) and messages (§9.3.2). Tarraingím's handling of two special cases is then described: cloning messages used to create objects (§9.3.3), and mirror objects used for structural reflexion (§9.3.4).

9.3.1   Primitive Objects

The VM implements SELF objects as if they were made up of three distinct parts — an object's *identity*, a *slots part*, and a *primitive part*. Section 5.4.1 has described an object's identity and an object's slots part. Most objects are *plain* objects, that is, they have no primitive part (equivalently, their primitive part is empty). Objects with non-empty primitive parts are known as *primitive* objects. A primitive object's primitive part contains data interpreted only by the SELF VM. This primitive data is accessed using primitive messages specific to the type of primitive object. For example, an integer object contains a fixed-precision integer value in its primitive part, and integer primitive messages are used to manipulate this value.

Figure 9.7 lists the various types of primitive objects in SELF. The figure groups the object types into two categories — mutable and immutable. Most types of primitive objects, including all plain objects, are *mutable*. A mutable object's data slots can be assigned to by the program; data in their primitive part can be changed by primitive messages; slots can be added to or removed from the slots part; and their primitive part can be removed or replaced by another mutable object. A plain object can be changed into a mutable primitive object, and vice versa.

Immutable objects (which are always primitive) cannot be changed. Primitive messages can retrieve data from their primitive parts, but cannot change them. Their slots part always consists of a single

static parent slot (named parent) which inherits behaviour from a particular traits object (§5.4.4). Each different type of immutable primitive object inherits from a different traits object. These traits objects are used by the SELF library to supply behaviour for the immutable primitive objects.

Mutable	Immutable
vector *(two subtypes)* byte vector mirror *(sixteen subtypes)* proxy *(two subtypes)* method *(two subtypes)* process	integer float canonical string block activation *(three subtypes)* assignment

Figure 9.7: Types of SELF Objects

Mutable Primitive Objects

Encapsulators can be attached directly to mutable primitive objects. Such encapsulators must monitor the operation of the object's primitive part, and ensure the object continues to operate correctly. Since the primitive part is manipulated by primitive messages, its execution is monitored in the same way as other primitive messages (§9.3.2).

Ensuring the object continues to operate correctly is more difficult. The primitive messages used to manipulate an object's primitive parts (*primitive object messages*) are sent to the object to be manipulated, typically as self-sends from normal methods inherited by the primitive object (§9.2.3). When an encapsulator is attached to an object, it displaces that object from the program, and arranges that self will refer to the encapsulator (§9.2). Any primitive object messages sent to the target object will then arrive at the encapsulator, but the encapsulator will not be able to handle them, since it does not include the object's primitive part.

To avoid this problem, we displace only the target object's slots part. The primitive part remains with the original object, that is, it becomes the encapsulator's primitive part. Any primitive object messages sent to the target object find the encapsulators primitive part, and so execute successfully. This is illustrated in Figure 9.8.
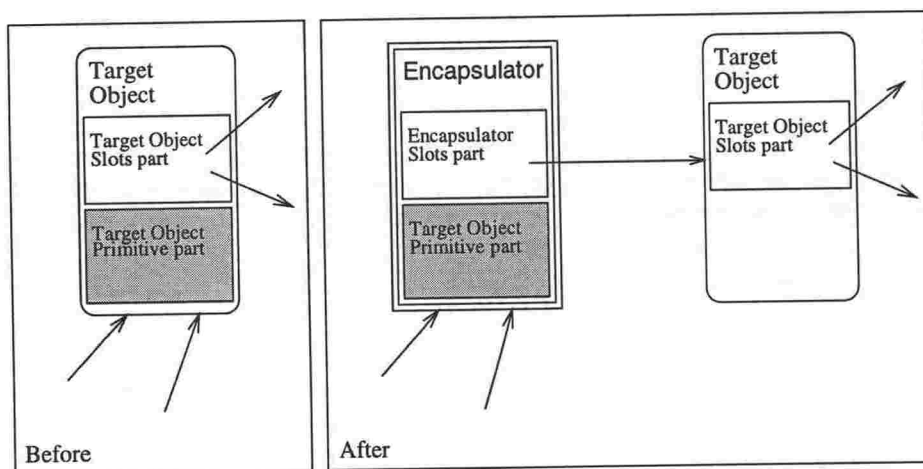

Figure 9.8: Attaching a split encapsulator to a Mutable Primitive Object

We call an encapsulator using this technique a *split* encapsulator, since the various parts of the target object are split when an encapsulator is attached. A split encapsulator may be any type of encapsulator which handles self correctly — delegating, inheriting, or custom (§9.2). Note that this problem did not occur in Pascoe's forwarding encapsulators in SMALLTALK. SMALLTALK's primitives can only be sent to self, and forwarding encapsulators do not change the target object's binding of self.

Immutable Primitive Objects

Immutable primitive objects cannot be altered. It follows that they cannot be displaced from the target program, and that encapsulators cannot be attached to them. Immutable primitive objects inherit from a traits object which is always a mutable plain object. An immutable object can therefore be monitored by attaching an encapsulator to the object's parent, rather than to the object itself (§9.2.4). As immutable objects only include a single parent slot, this encapsulator would not miss any messages handled by the primitive object before arriving at the parent. Unfortunately, it would monitor *all* the primitive objects of that type, not just the desired target object. While a filter watcher (§7.3.1) can be used so that views receive only the actions of the object of interest, overall this approach is quite inefficient.

In practice, we have found little need to monitor immutable primitive objects. Although these objects are used widely throughout SELF programs, they are used in very mundane ways, and their behaviour is usually implemented in the SELF library, rather than user's programs.

9.3.2   Primitive Messages

Primitive messages are used by SELF programs to perform fundamental operations for all object types (such as creating and comparing objects), and specialised operations for primitive objects. Primitive messages are essentially VM subroutines which are called from SELF programs. They are handled very differently from other SELF messages.

The names of all primitive messages begin with an underscore "_". When a primitive message is to be sent (for example "a _IntAdd: b"), its arguments and receiver are first evaluated, as for any other message. Unlike normal message sends, SELF's message lookup algorithm (§5.4.4) is not invoked. Instead, the VM recognises the primitive message syntax, and simply searches an internal table.

This special lookup poses a problem. Encapsulators monitor objects by intercepting the messages they receive: this depends critically upon messages being delivered by the standard message lookup algorithm. The VM effectively executes primitive messages as soon as they are sent, with no SELF-language level lookup. There is no way to intercept or change the behaviour of primitive messages apart from modifying the VM, so encapsulators cannot intercept these messages.

Primitive Message Wrappers

Primitive messages are in general a problem in SELF. Because these messages are type-specific (_IntAdd is distinguished from _FloatAdd), using them eliminates the possibility of polymorphism. Since primitive messages may be sent directly to any object in the system, and can directly manipulate an object's primitive part, they can bypass the encapsulation or data-hiding provided by the language. Code using primitive messages is very sensitive to small changes in the language implementation, because the primitive messages are implemented directly by the VM.

Writing primitive messages is therefore considered bad style [215]. The SELF library includes a normal method (found by the message lookup mechanism) corresponding to almost every primitive message. These *primitive wrapper* methods (not to be confused with an inheriting encapsulator's wrapper methods) simply invoke the corresponding primitive message (see Figure 9.9). Sending a wrapper message is the usual way to invoke most primitives in SELF. This allows the primitive facilities to be used while maintaining the benefits of SELF such as polymorphism, encapsulation, and abstraction. Changes to the VM can be isolated within the wrappers, rather than propagated throughout the whole program.

```
times = (|
    ^ user    = ( _TimeUser).  "user time in msec used by Self"
    ^ system  = ( _TimeSystem).  "system time in msec used by Self"
    ^ cpu     = ( _TimeCPU).  "cpu time (user + system) used by Self"
|)

traits mirrors = (|
    ^ reflecteeIfFail: fb = ( _MirrorReflecteeIfFail: fb).
    ^ namesIfFail: fb = ( _MirrorNamesIfFail: fb).
|)
```

Figure 9.9: Some Primitive Wrappers

An encapsulator cannot monitor primitive messages, however, it can monitor the wrapper methods since they are standard SELF methods reached by the normal method lookup algorithm. This is only a partial solution: the target program may be written in a style that sends primitive messages directly, and the user can always enter primitive sends interactively. In practice, monitoring only the primitive wrapper methods and ignoring the actual primitive messages has proved a satisfactory solution.

Note that SMALLTALK uses primitive *methods* rather than primitive *messages* and, since methods are executed *after* a message lookup, effectively avoids this problem [85]. Always sending SELF's primitive messages from within wrapper methods is effectively treating the primitive messages as if they were methods. For efficiency reasons, SMALLTALK includes some *special selectors* to perform common operations (such as identity comparison) which, like SELF's primitive messages, bypass the normal method lookup algorithm. SMALLTALK encapsulator implementations have dealt with special selectors by removing them from the target program's source code [16] or by changing the SMALLTALK compiler [138, 164] to generate normal sends for the same messages.

Receivers and Arguments

Some primitive messages (such as _Quit which ends the SELF session) take no arguments and ignore their receiver. Most primitive messages, however, do take arguments, and do rely upon the value or structure of their receiver. For example, the _Eq primitive message tests pointer equality between two objects (similar to LISP's eq operation), and the _IdentityHash message returns a hash value for an object such that two objects which are _Eq will have the same hash value. Section 9.3.1 described primitive messages which are used to manipulate an object's primitive part. These messages depend on the identity and the primitive parts of their receiver and arguments. Split encapsulators do not alter the identity or primitive parts of their target objects (§9.3.1), so these messages can be handled without any additional attention.

Some primitive messages depend upon the slots part of their receiver or arguments, in addition to their primitive part and identity. Split encapsulators displace their target object's slots part, so that if an encapsulator is attached to the receiver or an argument to such a message the message will not perform correctly. For example, the _Clone primitive message returns a low-level copy of its receiver, including both its primitive and slots parts. If _Clone is sent to an encapsulator, it will return a copy of the encapsulator, not the target object.

The next subsection describes how we have extended Tarraingím's encapsulators to handle messages (such as _Clone) which depend upon their receiver's slots part. Section 9.3.4 further extends these techniques to deal with the primitive messages associated with mirror objects, which provide structural reflexion in SELF.

9.3.3 Cloning Messages

The *cloning* primitive messages (_Clone, _Clone:Filler and _CloneBytes:Filler) provide SELF's basic support for creating new objects. As SELF is a prototype-based language, the programmer clones a prototype to

create a new object, rather than creating the object *ab initio* (§5.4.1). Sending _Clone to an object returns a new object with exactly the same structure, slots, and values as the receiver of the _Clone message. These messages depend upon an object's slots and primitive parts.

When an object with an attached encapsulator receives a cloning primitive message the result is a clone of the encapsulator. These two encapsulators will have the same target object and the same controller. This is a problem: we must instead return a clone of the displaced object, without an encapsulator attached. To do this, it is necessary to intercept these primitive messages, or in practice, their primitive wrappers. There are, however, no unique wrappers for these primitives. Although all the cloning primitive messages are wrapped by other methods, these messages are often redefined by the programmer to provide other behaviour.

For example, the message clone is defined in traits clonable as a wrapper for the _Clone primitive. All *clonable* objects (those which can be cloned, either in the SELF library or a user's programs) inherit from traits clonable to get access to the _Clone primitive. In traits oddball (a traits object similar to traits clonable, but for *oddball* objects which can not be cloned), clone is defined to return self, that is, to return the actual object receiving the message rather than a copy. This is because objects which inherit from traits oddball should be unique: only one copy of each of these objects should exist in the SELF world. This restriction is implemented by traits oddball's definition of the clone message.

We cannot simply consider the clone message a wrapper for the _Clone primitive, even though the _Clone primitive is only accessible via the clone message. This is because clone does not necessarily call the _Clone primitive message. We therefore modify the SELF library source code to introduce a unique wrapper message for each cloning primitive. After this modification, every use of _Clone is replaced with a call to a new primitiveClone wrapper method which does nothing but call _Clone. Similar substitutions are made for the _Clone:Filler and _CloneBytes:Filler messages.

Primitive Override Methods

Calls to the cloning primitives can now be easily identified, since they will send one of these new wrapper messages. To handle these primitives within an encapsulator, we add extra methods to the encapsulator which override the standard behaviour of the cloning primitives. These *primitive override* methods are inserted between the encapsulator proper and the target object, that is, after the encapsulator's undefined-Selector trap or wrapper methods.

For the cloning primitive messages, the primitive override methods return copies of the displaced object, not the encapsulator. If a split encapsulator has been used, the primitive override methods combine copies of the various parts into a single object. Primitive overrides are invoked after the encapsulator intercepts the cloning wrapper message receipt, so the wrapper messages can be monitored by the encapsulator. The actions of the primitives are then emulated by the overrides, rather than being handled by the actual wrappers and primitive messages.

To summarise, when an object receives a message (such as clone) which will eventually invoke a cloning primitive, the definition of that message in the target object will eventually send the new primitive wrapper message (such as primitiveClone). If an encapsulator is not attached to the target object, primitiveClone will invoke _Clone, which will copy the object. If an encapsulator is attached to the target object, the message will be handled by the appropriate primitive override method, which will return a copy of the target object, without an attached encapsulator.

Programming and Debugging Messages

A small number of primitive messages provide low-level support for programming and debugging. Like cloning messages, these messages also depend on all parts of their receiver and arguments. For example, _Print directly prints the structure of its receiver, _Define replaces its receiver's structure with that of its argument, and _RemoveSlot deletes one of its receiver's slots.

If _Print is applied to an object with an attached encapsulator it will display the structure of the encapsulator, not the target object. _Define may completely remove an encapsulator from its receiver, or

duplicate an encapsulator in the same way as _Clone. A well-chosen _RemoveSlot can completely disable an encapsulator.

SELF programming style eschews these messages, which are designed to support interactive programming and debugging. A much better interface for reflexive programming is provided by SELF's mirror objects. The programming primitives are used directly from the SELF command line, or in script files used to load SELF programs, so no wrappers exist for these messages.

Tarraingím does not handle these primitives. This has not proven to be a problem, as these messages do not appear in typical SELF programs. If they were more common, primitive wrapper methods could be written for these messages, so they could be handled with primitive override methods in encapsulators.

9.3.4 Mirrors

Mirrors are primitive objects which implement SELF's structural reflexive facilities. Both SELF's debugger and Tarraingím use information obtained via mirrors to manipulate or display the structure of objects. Each mirror object provides information about one particular object in the SELF world – that mirror's reflectee (see Figure 9.10). Like other types of primitive objects, mirrors are manipulated by primitive messages. Mirrors' primitive messages are sent by wrapper methods inherited by the actual mirror objects.
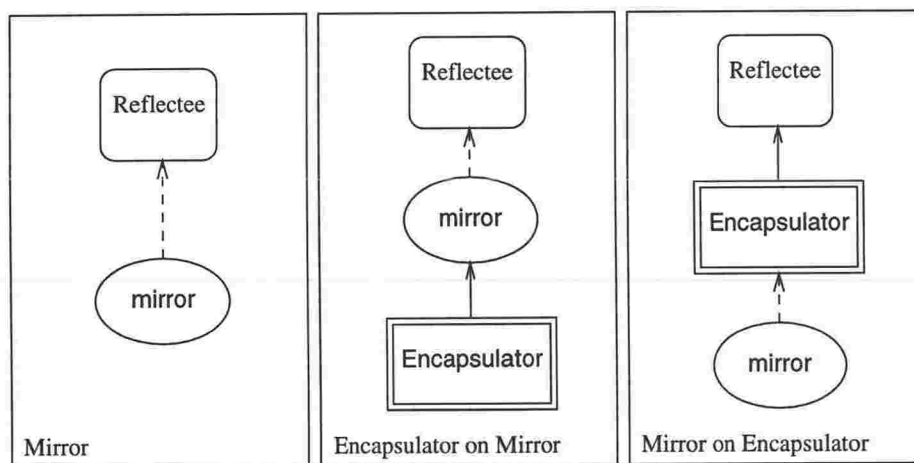

Figure 9.10: mirrors and encapsulators

mirror itself) has an encapsulator attached (see the third panel of Figure 9.10). If this is so, the mirror will provide information about the encapsulator, rather than about the original reflectee.

Note that only those mirror primitive messages which refer to the reflectee's slots part need to be redirected. A split encapsulator does not change its target object's identity or primitive part, so mirror primitives which refer to the mirror reflectee's identity or primitive part do not need to be redirected.

Tarraingím therefore modifies the standard implementation of mirrors to detect when their reflectee has an encapsulator attached, and if so, to ignore the encapsulator. This is done by altering the primitive wrappers in the traits mirror object for those messages which would otherwise detect the encapsulator, as shown in Figure 9.11. The new wrappers send mirror primitives to the result of a call to slotMirror, rather than (implicitly) to self.

The slotMirror method determines if the mirror's reflectee has an encapsulator attached, using the mirror's _MirrorNames primitive. If so, it returns a new mirror reflecting on the displaced slots part of the target object, so the mirror primitive message will be redirected, and will now apply to the displaced slots part of the target object. If not, slotMirror returns self, that is, the original mirror, which must be reflecting an object without an encapsulator attached. In either case, the existence of an encapsulator is hidden from the mirror's client.

```
traits mirror = (|

  "normal wrapper method"
   ^ namesIfFail: fb = ( _MirrorNamesIfFail: fb).

  "altered wrapper method"
   ^ namesIfFail: fb = (slotMirror _MirrorNamesIfFail: fb).

  "definition of slotMirror"
   _ slotMirror = (
       _MirrorNames includes: 'target'
        ifTrue: [_MirrorContentsAt: 'target']
                     "this primitive returns a mirror"
        False: self).
  "..."
|)
```

Figure 9.11: Primitive Method Wrappers within mirrors

These changes to mirror primitive wrappers comprise the largest single modification of the SELF library source code that is required for Tarraingím to operate. The source file containing the changes is about one hundred lines of SELF code. The changes required to insert cloning primitive methods wrappers (§9.3.3) amount to less than fifty lines. In order to implement Tarraingím, we have changed less than one hundred and fifty lines of the preexisting SELF library, comprising less than thirty methods.

Dynamic Reflexion

The SELF system includes debuggers and profilers which can display a program's call stack. Installing an encapsulator into the target program causes additional method sends, which place extra frames onto the stack, and are revealed by the SELF debugger. These message sends are ignored by Tarraingím, either directly, because of their selector names (§9.1.4), or indirectly, by the meta-depth mechanism (§8.2.3).

Like the structural reflexive utilities, the SELF debugger is implemented using mirrors to gain access to a reified control stack. The problem for these activationMirrors is not that the objects making up the stack have been replaced by encapsulators, but that extra frames will have been introduced into the stack by the messages sent within encapsulators. We believe that techniques similar to those used to hide

encapsulators from object mirrors would also effectively hide encapsulators from activation mirrors. We have not yet implemented these changes, as we have not found this to be a problem in practice.

9.3.5 Summary

SELF uses primitive objects and messages to provide access to data types and operations implemented by the SELF VM. Primitive objects can be monitored by using split encapsulators, which displace the target object's slots part while leaving the primitive part intact. Primitive messages cannot be monitored directly, but they are usually called from SELF-level wrapper methods which can be monitored by encapsulators. Some of these wrapper methods have to be modified so that objects with encapsulators can be cloned and to hide encapsulators from SELF's structural reflexive facilities.

9.4   Summary

This chapter has described four types of encapsulator. Forwarding encapsulators were originally described by Pascoe in SMALLTALK, and monitor the top level actions of their target objects. We have developed delegating, inheriting, and custom encapsulators, which avoid the self problem and can monitor all their target objects' actions. The Tarraingím prototype program exploratorium described in this thesis uses inheriting encapsulators, but we believe that both delegating and custom encapsulators should also be effective in appropriate environments.

Most types of encapsulators have some limitations upon their operation. Delegating encapsulators do not work in versions of SELF which enforce privacy; inheriting encapsulators' wrapper methods must be updated whenever the target program is changed; most types of encapsulators do not operate correctly in all situations when other objects inherit from their target object; and immutable primitive objects are difficult to monitor with any kind of encapsulator.

The target program's style must be constrained if it is to be monitored with encapsulators. In particular, the encapsulators' and target program's namespaces must be separated, and primitive messages must always be sent from wrapper methods. These constraints have not been a problem in practice.

Encapsulators of all types impose overheads on the execution of the target program. Delegating, inheriting and forwarding encapsulators are quite inefficient, because they capture many messages which are not required by the monitoring system. Custom encapsulators intercept only those actions that are needed, but, as their name suggests, they must be tailored specially to suit a particular situation, and this makes them more expensive to create than other types of encapsulators.

We found the construction of a monitoring subsystem for Tarraingím more challenging than we anticipated. Languages such as SMALLTALK and SELF generally have debuggers which are based upon an interpreter [84], or require specialised support from the compiler [103], and in general, debugging techniques for object oriented languages are currently the subject of research [52].

Two interactions between this subsystem's requirements posed the majority of problems. First, Tarraingím's monitoring has to be selective with respect to target objects, and the message names and types of the objects' actions. Encapsulators are an efficient way to monitor all the actions of a single object, while particular messages sent to any object can be monitored by wrapping message definitions [23]. For Tarraingím's purposes, selecting actions on a per-object basis is more important than the particular message name or event type. Thus monitoring the target program with encapsulators and then filtering the resulting actions with controllers and watchers is an effective solution.

Second, objects must be monitored dynamically — encapsulators may be attached to (or removed from) their target objects while the target program is running. Encapsulators with a fixed structure, such as delegating and inheriting encapsulators, are quicker and easier to attach to their target objects than more complex encapsulators, such as custom encapsulators. Once attached, though, custom encapsulators can monitor the target program more efficiently. Since inheriting encapsulators performed acceptably within Tarraingím, we have not pursued custom encapsulators.

The requirement that monitoring should be transparent to the user has been comparatively easy to achieve. Through SELF's use of mirror objects to implement almost all reflexive functions, encapsulators can be hidden by quite simple changes to mirrors. Once these changes are made, all of SELF's reflexive facilities can be used without detecting encapsulators.

116. You think you know when you learn, are more sure when you can write, even more when you can teach, but certain when you can program.

Alan Perlis, *Epigrams On Programming* [168]

10

Tarraingím in Action

25. One can only display complex information in the mind.
Like seeing, movement or flow or alteration of view
is more important than the static picture, no matter how lovely.

Alan Perlis, *Epigrams On Programming* [168]

This chapter presents several examples of the Tarraingím system in use. Section 10.1 describes Tarraingím from the user's point of view, and shows how a program, or any part of the SELF world, can be explored using Tarraingím. The second section (§10.2) describes how the component parts of Tarraingím are combined to create visualisations, illustrating the use of views, watchers, and controllers. Finally, the third section (§10.3) presents a larger visualisation, showing the operation of a parser for a small programming language. All the figures in this chapter, like all the screen dumps in this thesis, were created using Tarraingím.

As the epigram quoted above indicates, static figures do not give a good impression of dynamic views. This is doubly the case when presenting an interactive system such as Tarraingím, where the user's commands and choices are as important as the information presented. In this chapter, although we present sequences of views the user could request, or "before" and "after" versions of the same view, this does not give a good impression of the feel of an interactive system.

10.1 Exploring The Self World

All the objects in a SELF program, including the library, the user's program, and the Tarraingím system's implementation, are contained within SELF's object space, known as the SELF *world*. The SELF world is organised by structures built directly out of objects, rather than by using special packages (as in LISP) or dictionaries (as in SMALLTALK) [217]. The SELF world begins from an object known as the lobby, which is the root of the SELF namespace. The SELF library is organised as a tree rooted at the lobby. The slots of the lobby contain various *category* objects which are used to structure the namespace. The category objects contain further *subcategory* objects, and these subcategories eventually contain the objects making up SELF's library.

In this section, we describe how an end-user can use Tarraingím to browse SELF's namespace and library. Figure 10.1 illustrates an iconic browser view displaying the lobby. This view is displayed by default when Tarraingím is started from the SELF command line. Each icon in the view represents the contents of one of the lobby's slots.
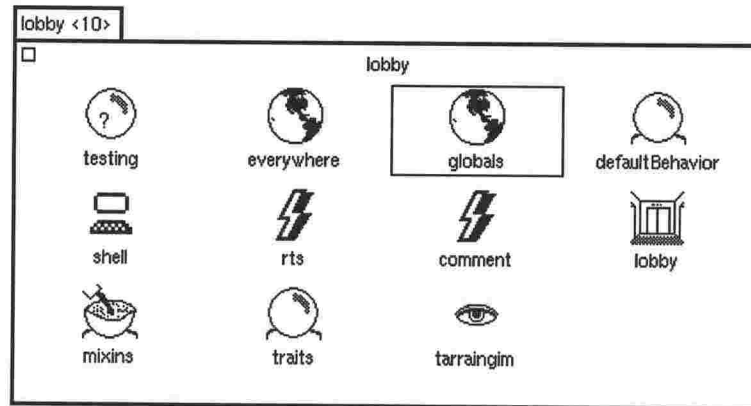
Figure 10.1: The lobby

The icon browser view is implemented by a hierarchical view (§6.5). Each slot in the target object is visualised by a separate subview, which draws the name of the slot, and an icon representing the slot's contents. The subviews choose the icon for a particular object from a table supplied by the visualiser. The icon chosen can depend upon the object's name (for important objects such as the lobby) or the kind of object. If no icon is found, the object is displayed using a *soap bubble* icon[1]. The structure of the icon browser view is discussed further in Section 10.2.2.

The lobby object contains the following slots:

- testing contains various objects related to testing the system.

- everywhere contains important system variables accessible from every object.

- globals contains various globally accessible objects: in particular, the prototypes of objects in the SELF library.

- defaultBehavior contains behaviour common to most objects (including support for printing objects, and comparing their identity).

- shell is the command line user interface.

- rts is a method used to install Tarraingím.

- comment is a method which provides a comment about the lobby.

- lobby is a link to the lobby itself. This is how the lobby gets its name.

- mixins contains objects which are inherited by other objects (like traits), but are not associated with a particular prototype.

- traits contains traits objects for the prototypes stored in globals.

- tarraingim contains system variables for Tarraingím.

All Tarraingím's windows can be manipulated (moved, resized, temporarily hidden and so on) using the X window manager. This is controlled from the window's title tab. The title tab for the browser view in Figure 10.1, lobby <10>, displays the name and *object ID number* for the object displayed in the view. A view's title tab display is implemented by NAVEL (§6.1.3), and shows a view's target object's name and object ID number by default. A view's title is one of its parameters and can be changed by the visualiser (§6.1.1).

---

[1] ⬤ see the quotations at the end of this chapter.

10.1.1 Navigation

The user can navigate around the SELF world by clicking the left mouse button on the icons in browser views (§6.6.1). Tarraingím then produces a browser displaying the indicated object. For example, selecting globals from the lobby browser shown in Figure 10.1 produces the view of the globals object displayed in Figure 10.2. Like the lobby, and the other category objects described in the chapter, the globals object is part of the standard SELF library.
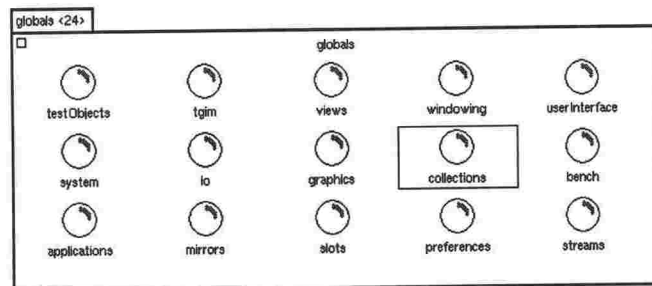
Figure 10.2: globals

The globals object contains fifteen slots, each containing a subcategory object. These objects (tgim, views, testObjects, and so on) contain useful objects belonging to that category, or further subcategory objects. For example, selecting collections from the globals browser produces a further browser, which displays the collections subcategory (see Figure 10.3), containing various subcategories of collections. Selecting tgim would display a browser on the tgim category object, which contains prototypes of the objects implementing Tarraingím.

Figure 10.3: collections category

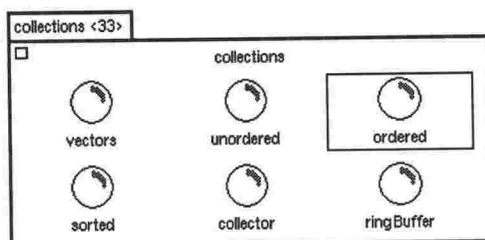Selecting ordered from within collections displays the ordered collection subcategory (see Figure 10.4), which contains actual object prototypes.
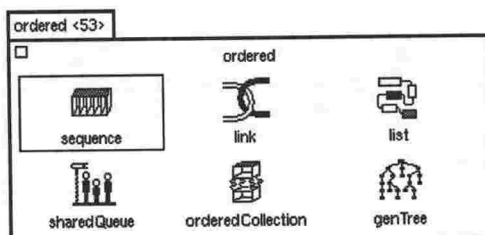
Figure 10.4: ordered collections

The ordered collection subcategory contains the following objects:

- sequence and orderedCollection are two alternative implementations of collections which are both indexable (like arrays) and extensible (like lists).

- link is a doubly linked list record.

- list is linked list implemented using links.

- sharedQueue uses a linked list and a couple of semaphores to implement a queue which can be shared between several concurrent processes.

- genTree is a generic n-ary tree.

Since the icon browser displays objects by showing their slots and contents, a prototype object can be selected and a new view will appear displaying its internal structure. Figure 10.5 displays the SELF library's sharedQueue and sequence prototypes.
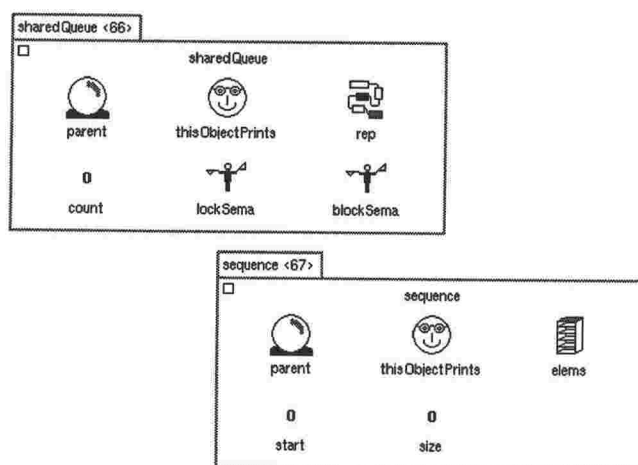

Figure 10.5: sequence and sharedQueue prototypes

Each of these objects has a parent slot containing its corresponding traits object, and a slot thisObjectPrints containing true, which is used by the SELF command line to determine that the object can be printed. The sharedQueue prototype has a rep slot containing a list, a count variable, and two semaphores. The sequence includes the elems array which holds the sequence elements, and also start and size pointers which indicate the portion of the array containing sequence elements.

Iconic views can be used to explore these objects further. For example, if the sequence's parent slot is selected, its traits object will be displayed. This contains mostly methods, which can in turn be selected and displayed, as illustrated in Figure 10.6.

10.1.2   View Selection

As well as navigating about the program, the user can request different views of any object (§6.6.1). Each view supplies a pop-up menu (on the right mouse button) from which alternative views can be selected (§6.6.1). This is illustrated for a sequence in Figure 10.7.

The sequence view menu contains some views which can display any kind of object, and others which are applicable only to collections (such as sequences). The inspector view displays an object's slots textually (it is a counterpart to the iconic browser, §6.5); the print view displays the default printed representation of an object; the trace, profile and profile (bar) views display operation traces and profiles respectively (some of these views are illustrated in Figures 3.2, 3.4, and 6.9). All these views can display any type of object. The dots, bigDots and sticks views (pictured in Figure 3.1), also the collection and keyed views, display only collections.

Note that although this menu is called the "view" menu, it really contains combinations of views and watchers. This is because end-users think in terms of complete visualisations, and are not aware the
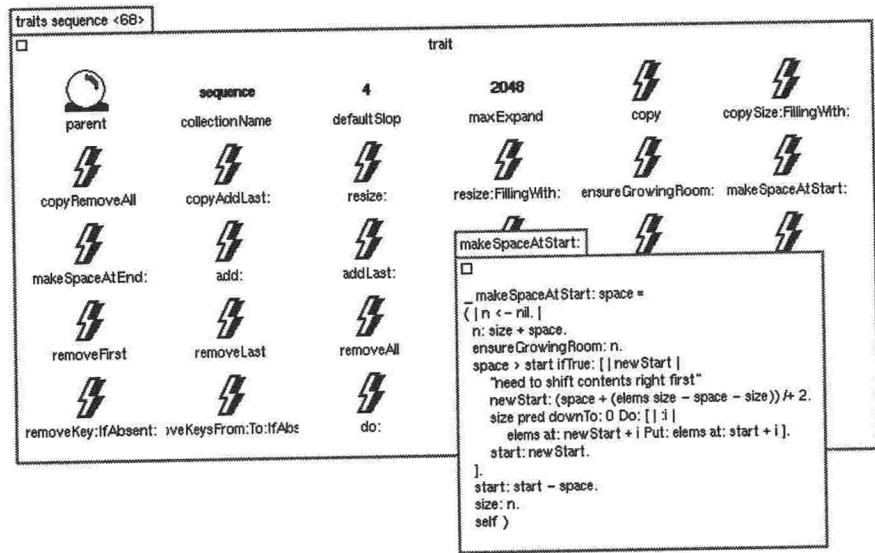
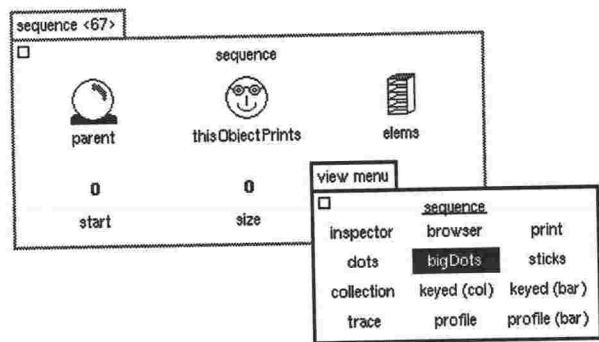Figure 10.6: **traits sequence** and a sequence Method

Figure 10.7: View Menu for a **sequence**

distinction between watchers and views (§7.1). For example, the keyed (col) and profile menu entries both create coll2View view objects (§6.7). The collection menu entry invokes a coll2View using a topWatcher (§7.2.3), and so directly displays the values in the sequence, while the profile menu entry invokes a coll2View which uses an indirect profileWatcher (§7.4.1), and so displays a profile of the collection's execution.

Selecting bigDots from the menu will produce a large scale scatterplot view of the sequence, similar to the view of a **vector** displayed in Figure 10.8.

10.1.3   User Commands

The user can send messages to the object displayed in a view. Each view supplies a command pop-up menu listing some messages which the view's target object understands. The visualiser must construct a suitable command menu for each view. The command menu for a sequence is illustrated in Figure 10.8, as requested from the big dots view. The menu contains commands to fill, shuffle and reverse the sequence, to run several sorting algorithms, and to print out the sequence's contents.

Some views allow their target objects to be manipulated directly (§6.6.2). For example, a dot in the view in Figure 10.8 can be dragged vertically to update the appropriate element of the sequence object.
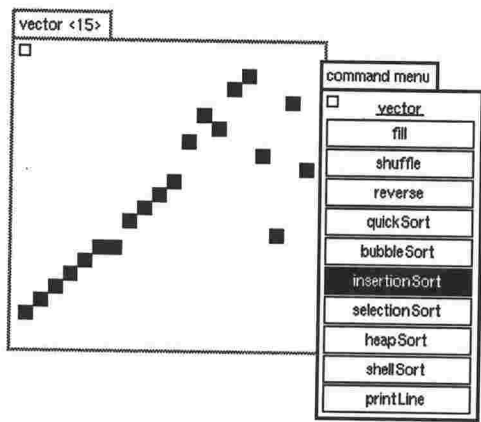
Figure 10.8: Command Menu for a sequence displayed in a bigDots view

10.1.4 View Commands

Although most view manipulation is carried out using the **X** window manager, some extra functions are available via a third pop-up menu, illustrated in Figure 10.9. This menu is known as the "meta" menu, since it relates to the view itself, rather than the object displayed by the view.

The left column of the meta-menu sends commands to the view: close closes the view; redraw refreshes the image (this can invoke user-driven update strategies, §4.3.4); name changes the title displayed in the view's title tab; and layout recalculates the view's layout. The entries in the right column request reflexive views of the view itself. Selecting props displays a property sheet for the view; browser displays an iconic browser of the view object; and meta requests a view of the view's structure, as described in Section 10.2.1.

Figure 10.9 also shows the property sheet for a dots view. This dialog box can be used to change the view's display parameters (§6.3). In Figure 10.9, the property sheet has been used to change the size and scale of the dots displayed by the view from Figure 10.8.
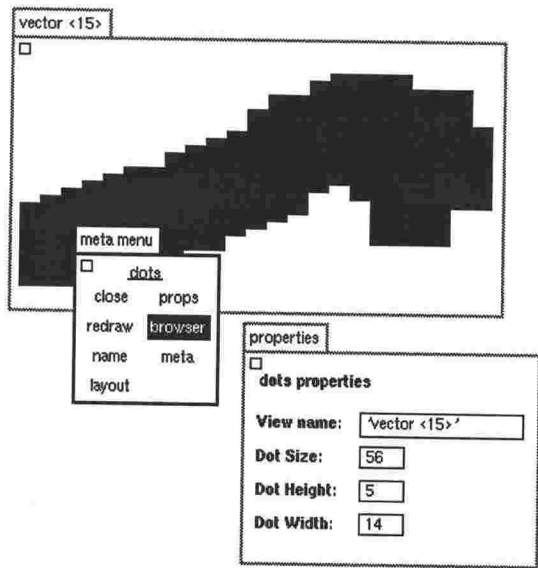

Figure 10.9: Meta Menu and Properties View for a bigDots View

Figure 10.10 shows an icon browser reflexively displaying a dots view. This browser displays the

internal implementation of the dots view, which relies heavily upon the internal graphics library. Of casual interest are the window, manager and display slots related to the **X** window system, and the iWatcher slot containing the view's watcher.
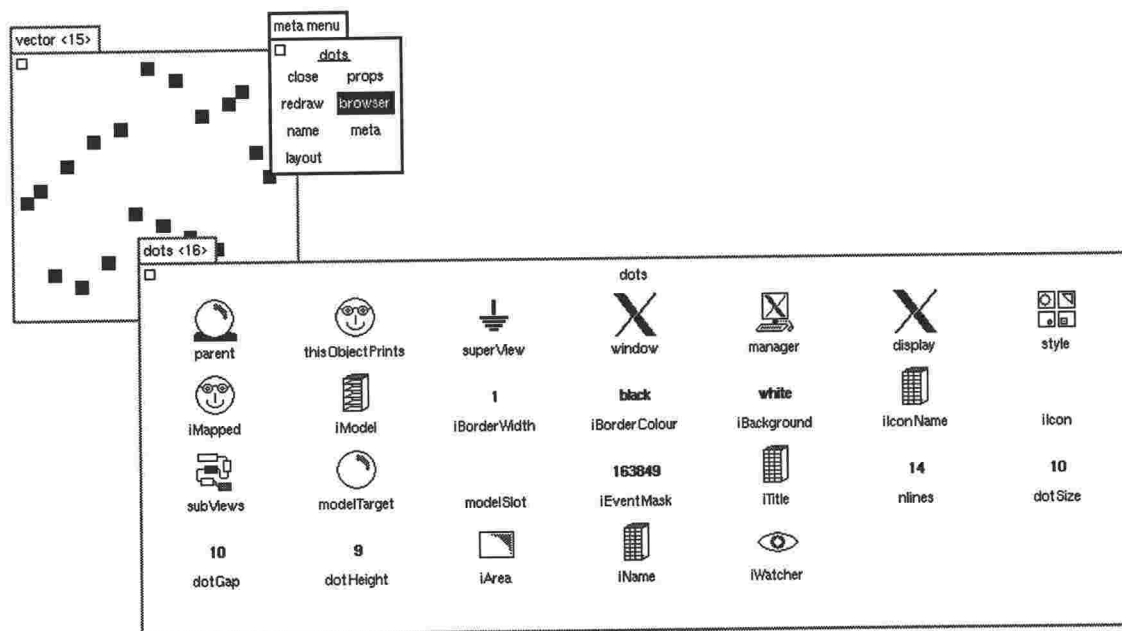

Figure 10.10: Reflexive Browser for a bigDots View

10.2 Tarraingím's Structure

Tarraingím's visualisations are constructed according to the APMV model by composing objects within the Tarraingím framework. As described in Chapter 5, a display is drawn by a view object, which may use several subviews. Every view is connected to a watcher, and each watcher may in turn use subwatchers. Watchers use controllers and encapsulators to monitor the target program.

This section illustrates how these objects are composed to produce visualisations, and begins by discussing simple displays employing only a single view and watcher (§10.2.1). Section 10.2.2 describes more complex displays incorporating multiple views, and Section 10.2.3 illustrates how compound strategies can be constructed from multiple watchers.

10.2.1 Simple Views

Many displays can be implemented with only one view and one watcher. This section describes the structure of three such views of collections: a scatterplot view, an operation trace view, and a simple textual list.

Dots View

Figure 10.11 shows a dots view (titled vector <15>) displaying a vector, and also a view structure view (titled dots <16>) displaying the implementation of a dots view. A view structure view reflexively displays the actual structure of controller, watcher, and view objects used in another view's implementation, in the same way the line diagrams used in Section 5.3.3 display the arrangements of these objects in Tarraingím's framework. A view structure view is requested by selecting the meta entry in a view's meta menu (see Figure 10.9).
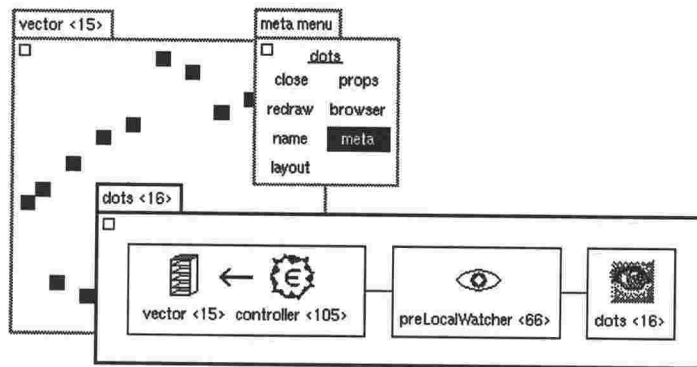
Figure 10.11: Dots View Structure

Reading the view structure view from left to right, it shows the three components of the APMV model: program, mapping, and visualisation. The program component is represented by the target object (the object vector <15>) and the controller (controller <105>). The mapping component is represented by the watcher (preLocalWatcher <66>), and the visualisation component is represented by the dots view itself (dots <16>).

The title tabs of all the views in this section have been configured to display the name and object ID of the view's target object. The view structure view in Figure 10.11 illustrates a dots view object with the object ID <16> — the view object maintaining the dots view shown in that figure. The dots view itself is displaying a vector (a primitive array of fixed size) with object ID <15>. This is the same vector that is shown to be the dots view's target object in the view structure view.

The dots view implementation is quite simple, utilising only a single view and watcher. The dots view object, described in detail in Section 6.3, draws and maintains the graphical display of the scatterplot. The preLocalWatcher notifies the dots view about changes in its target object's state before they are applied (§7.2.4).

The preLocalWatcher <66> uses the controller <105> to monitor the target object. If the target object needs to be monitored, the controller will automatically create and attach an encapsulator (§8.1.2). Note that view structure views do not display encapsulators, since encapsulators are hidden within the monitoring subsystem, and are always accessed via controllers[2] (§9.1).

Trace View

A trace view displays a textual list of the messages its target object has received recently. Figure 10.12 shows the structure of a trace view. Again, this consists of three parts: a controller monitoring the target object, a watcher, and a view.

The target object is vector <15>, the same object displayed by the dots view in the previous figure, so the controller (controller <105>) is also the same. The watcher (traceWatcher <145>) and view (tracer <24>) are different: the traceWatcher monitors all the actions of its target object (§7.2.2), and the view displays a textual list of the messages the target object receives.

Collection View

Figure 10.13 shows the structure of a simple collection list view (a collView). This is similar to the two previous cases, with the same target object and controller, but again, a different watcher and view. In this case, the watcher (topWatcher <156>) monitors top level message return actions of its target object (§7.2.3). When the view receives a change from the watcher, the view redisplays itself using a callback,

---

[2]The controller icon 🌀 was chosen to represent both encapsulators and controllers.
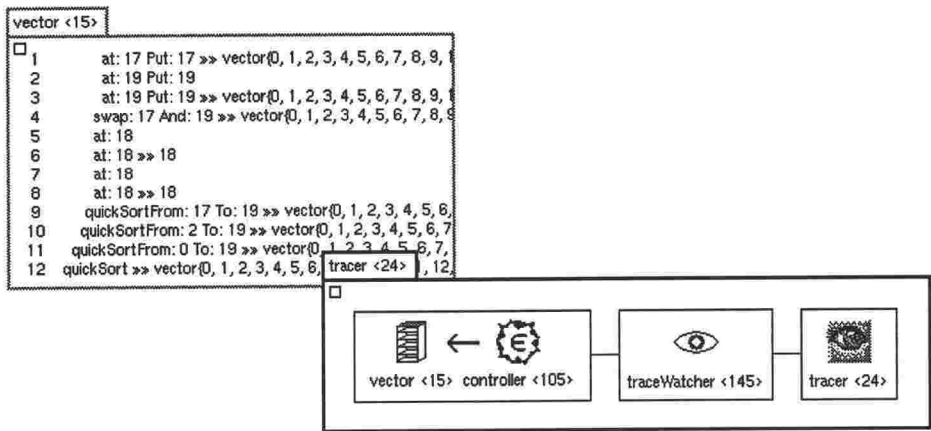
Figure 10.12: Simple Trace View

as described for the dots view in Section 6.3. The view is only notified of top level return actions, so that the execution of the callback is synchronised with the target program (§4.2.1).
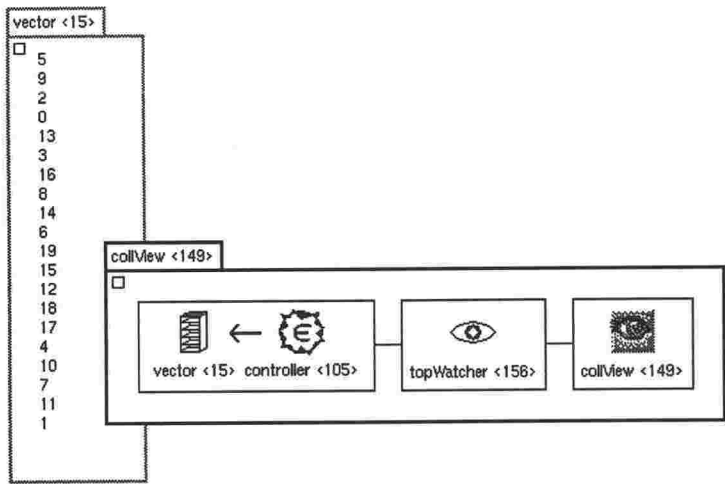
Figure 10.13: Simple Collection View

Dispatch Database

The three views described above have a similar structure, and share the same target object and controller. Just as each is a different type of view, each has a different type of watcher to provide a monitoring strategy suitable for that particular view.

When a watcher is attached to its target object, it registers its interest in the target object's actions with the target object's controller (§7.2). As different watchers embody different strategies, they monitor the target object in different ways. The immediate consequences of each strategy (i.e., each watcher's monitoring plan §4.3) can be seen in the controller's dispatch database (§8.1.3) as displayed in Figure 10.14. This view lists each watcher registered with a controller, and shows which events will be reported to that watcher. The figure displays the controller for the vector <15> object, so it lists three watchers, one from each of the views discussed in this section. The preLocalWatcher <66> is monitoring receipts of the at:Put:IfAbsent message to dynamically update the dots view, the traceWatcher <154> is monitoring all

actions of all types for the trace view, and the topWatcher <156> is monitoring all top level completed actions for the collection list view.
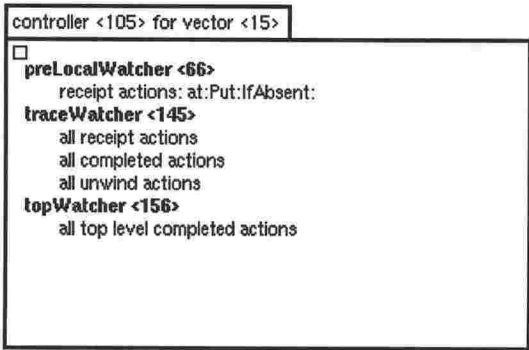
Figure 10.14: Controller Dispatch Database

10.2.2 Multiple Subviews

Section 6.5 describes the implementation of hierarchical views, which use subviews to produce their display. This section presents two examples of hierarchical views: an icon browser and a generic tree view.

Icon Browser

Figure 10.15 illustrates an icon browser view (§6.5.1) and a view structure view of that icon browser. The icon browser is titled unordered <176> because it is displaying the unordered collections category object, which has the object ID <176>. The icon browser is titled browserView <186> because it is displaying the structure of the icon browser view which has the object ID <186>.

The main browser view (browserView <186>) has three slotView subviews, numbered <183>, <184>, and <185>. Each of these subviews displays an icon and a name for a single slot in the target object. The main browserView uses a localWatcher, which detects changes in its target object's local state, and (unlike the preLocalWatcher of Figure 10.11) notifies its associated view after the changes have taken place. However, the category object unordered <176> is immutable — it has no local mutable state, as all its slots are constant slots. Thus it does not need to be monitored, so the localWatcher is not attached to a controller or target object, and accordingly none are shown in the figure. Similarly, the slotViews are configured to use nullWatchers (§7.2.1).

Many of Tarraingím's views, especially those implementing the user interface, use this structure. For example, the various menus (e.g. Figures 10.7 and 10.8) and the view property sheet (Figure 10.9) are all implemented in this way. Of course, menus use different types of subviews (for example, the view menu uses viewButtons) rather than the slotViews used by the icon browser.

Tree View

Figure 10.16 illustrates a view of a tree. The tree consists of two generic tree (genTree) objects labelled "phrase tree" and "noun tree" comprising the interior nodes, and four token nodes at the leaves.

Figure 10.17 shows the structure of the tree view from Figure 10.16. The view is managed by an nTreeView object, which itself uses a nullWatcher, since the actual work of displaying the tree is handled by the tree subviews. Interior nodes (genTree objects) are displayed by treeNodeViews, while leaf nodes (token objects) are displayed by tokenViews. Both types of views use localWatchers to monitor their target objects. No special icon for token objects has been supplied, so they are drawn by the default icon. The lines between the nodes are drawn by the nTreeView and are updated whenever the arrangement of treeNodeViews changes.
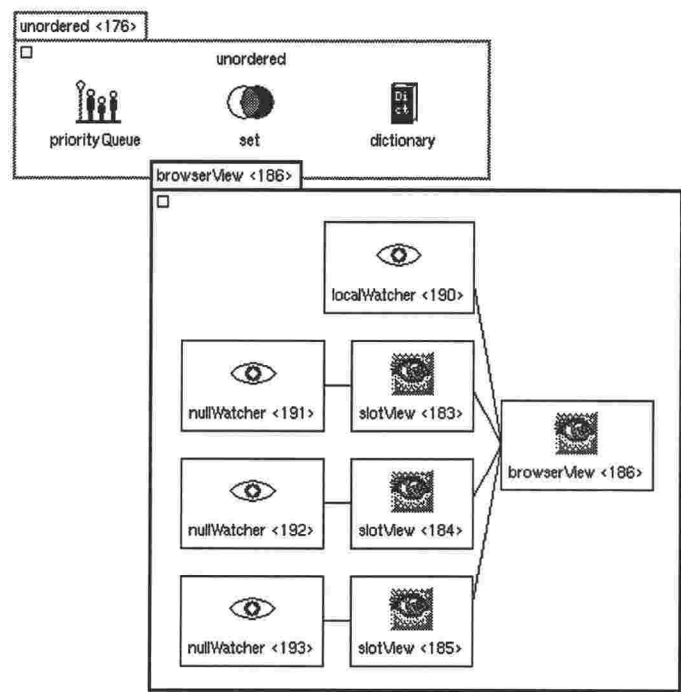
Figure 10.15: Icon Browser
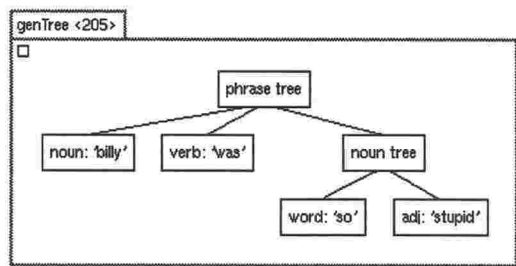
Figure 10.16: Tree View

10.2.3 Multiple Watchers

In the same way that a single display can be implemented by more than one subview, a single view can use more than one watcher (§5.3). Section 7.5 describes several kinds of strategies implemented using multiple watchers. This section focuses on two examples: the use of indirect watchers to build an aggregate view, and passing models by reference in an implementation view.

Aggregate Abstractions

Views of aggregate abstractions (such as those illustrating the target program's performance or structure, §3.1.3) can be implemented by first gathering information about the abstraction into a database object, then displaying that database.

Figure 10.18 shows two views: a cubist picture of a trafficLight object (§6.2) and a bar graph showing the number of times each message has been sent to the trafficLight object. The structure of the profile view is shown in Figure 10.19. The profiling is completely implemented within the mapping component, that is by watchers, and any collection view can be used to display the resulting profile database.
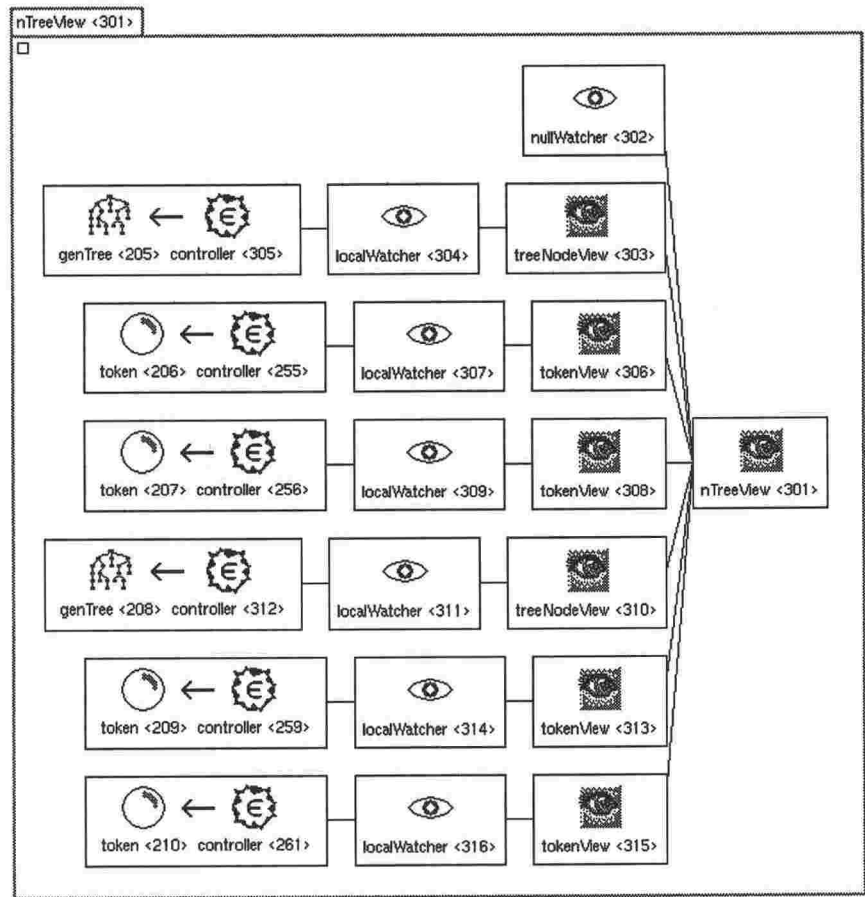
Figure 10.17: Tree View Structure
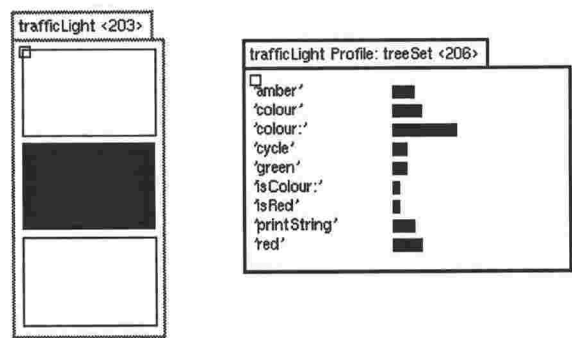
Figure 10.18: trafficLight and barProfile Views

The actual profile view (barGraph <210>) is directly connected to one watcher, profileWatcher <204>, which itself uses two subwatchers. One subwatcher, recvWatcher <207>, monitors all messages received by the traffic light object trafficLight <203>, and forwards these to the profile watcher. In response to these messages, the profile watcher constructs the profile by inserting or updating entries in its tally database, here implemented by the object treeSet <206>. A second subwatcher, changeWatcher <205> monitors the database: this subwatcher's event notifications are routed to the view, which displays the tally object. Using two subwatchers (rather than using a single monolithic profileWatcher) means that the strategies used to select events to be profiled and to update the view can be chosen independently of

the construction of the profile. A different view may need to use a different strategy to monitor the tally — if so, a subwatcher can be changed, without otherwise affecting the profileWatcher itself. Section 7.4.1 describes the implementation of profileWatchers in more detail.
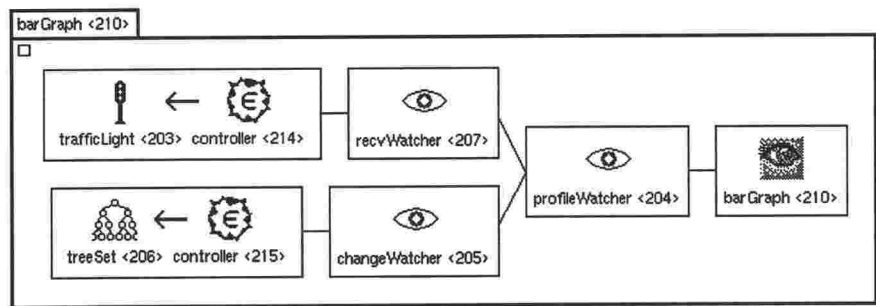
Figure 10.19: barProfile View Structure

Figure 10.20 shows the controller dispatch database for the traffic light and tally set objects. The traffic light's controller has two registered watchers: recvWatcher <207>, which is monitoring all receipt actions to build the tally (this watcher appears in the upper branch of Figure 10.19); and localWatcher <202>, which is used to produce the cubist view in Figure 10.18. The tally treeSet's controller has only one registered watcher, changeWatcher <205>, which detects change actions (additions and removals) in the database: this watcher appears in the lower branch of Figure 10.19. The localWatcher <202> does not appear in Figure 10.19 because it is used to produce the cubist traffic light view, rather than the profile, and Figure 10.19 shows the structure of the profile view.
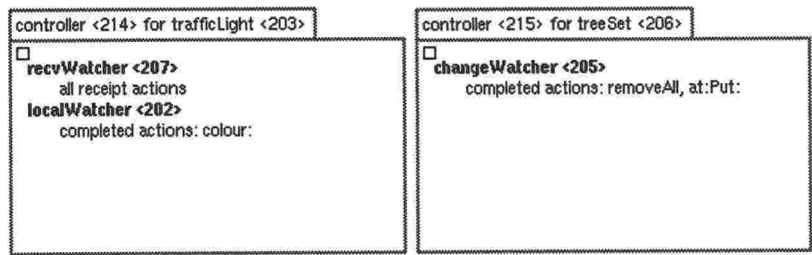
Figure 10.20: Dispatch Databases for the barProfile View

Passing Models by Reference

In some circumstances, a view's target object must be determined indirectly, as the value of some other object's slot or the result of a message send (§7.4.2).

Consider SELF's dictionary object which implements a hash table to provide a mapping from *keys* to *values*. Figure 10.21 displays a textual view of a dictionary (dictionary <983>) containing a small telephone directory. The dictionary's keys are the names of telephone users, and its values are their telephone numbers.

Figure 10.22 shows an icon browser view displaying this dictionary object. Note that the icon browser in this figure uses a different slotView from the standard browser (§10.2.2). This alternative slotView shows each slot's content's object ID.

The dictionary <983> object shown in Figure 10.22 stores its entries as a hash table in two parallel arrays. These arrays are kept in the dictionary's keys and values slots and are the vector objects with ID's <1088> and <1089> respectively. The hash table can be searched for a particular key: this is implemented by open address hashing with rehashing on the keys array.

```
dictionary <983>
┌────────────────────────┐
│□                       │
│   'Lindsay'   -> 8070  │
│   'Mark'      -> 5055  │
│   'Robert'    -> 8546  │
│   'Julian'    -> 8579  │
│   'Duncan'    -> 5054  │
│                        │
└────────────────────────┘
```
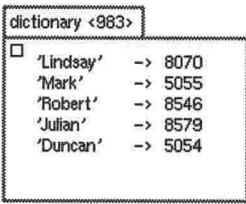
Figure 10.21: A dictionary containing a Telephone Directory

The dictionary includes a parent slot, two sentinels (emptyMarker and removedMarker) and several integer variables. The two sentinels mark empty slots in the keys array. An emptyMarker marks an unused slot: if a hash probe reaches an emptyMarker, the key is not contained in the array. The removedMarker marks a slot from which an element has been removed: if a hash probe reaches a removedMarker, this particular slot does not contain the object for which the probe is searching, but the hash probe must continue. The size slot contains the number of elements in the dictionary. The highMark and lowMark slots contain sizes at which the hash table must be expanded or contracted. The minBuckets slot holds the minimum number of buckets to be kept by an empty hash table.
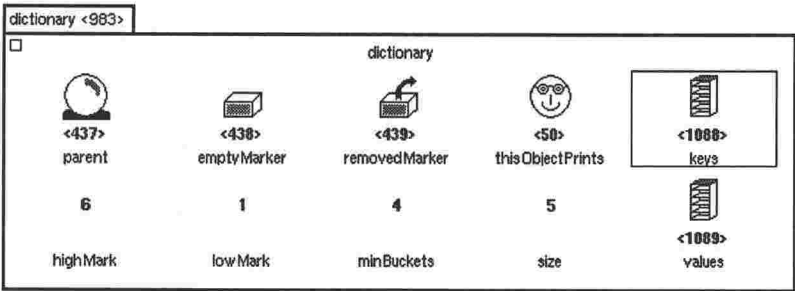
Figure 10.22: dictionary Implementation

The two parallel vectors are illustrated in Figure 10.23. The keys vector (on the left of the figure) contains the keys of the dictionary (the names). The values vector contains values (telephone numbers) stored at the same position as their associated key in the keys array. Empty hash table entries are indicated by the emptyMarker in the keys vector: the corresponding values vector entries contain nil, as they will never be accessed.

```
dictionary <983> Slot: keys Watch: vector <1088>        dictionary <983> Slot: values Watch: vector <1089>
┌────────────────────────────────────┐  ┌────────────────────────────────────┐
│□                                    │  │□                                    │
│  0   -> emptyMarker                 │  │  0   -> nil                         │
│  1   -> 'Lindsay'                   │  │  1   -> 8070                        │
│  2   -> emptyMarker                 │  │  2   -> nil                         │
│  3   -> 'Mark'                      │  │  3   -> 5055                        │
│  4   -> 'Robert'                    │  │  4   -> 8546                        │
│  5   -> emptyMarker                 │  │  5   -> nil                         │
│  6   -> 'Julian'                    │  │  6   -> 8579                        │
│  7   -> 'Duncan'                    │  │  7   -> 5054                        │
│                                     │  │                                     │
└────────────────────────────────────┘  └────────────────────────────────────┘
```

Figure 10.23: keys and values Vectors

The two views illustrated in Figure 10.23 are coll2Views, which are similar to the basic collView

collection view (§10.2.1), except that they display both vector indices and elements (i.e., keys and values). The abstract display of the whole dictionary in Figure 10.21 is produced by another coll2View.

The coll2Views in Figure 10.23 use a slotWatcher to specify their target object by reference. The views' titles include the name and ID of the dictionary object, the slot name, and the name and ID of the vector object they are displaying.

Figure 10.24 contains a view structure view reflexively displaying the keys vector view from Figure 10.23; the values view has the same structure. The target object (vector <1088>, in the bottom left of Figure 10.24) is monitored by the localWatcher <986> object. A localWatcher is used instead of a topWatcher, because vectors are primitive objects and are updated atomically. Thus their representation can never be in an inconsistent state, and callbacks can be sent at any time (§7.2.4).
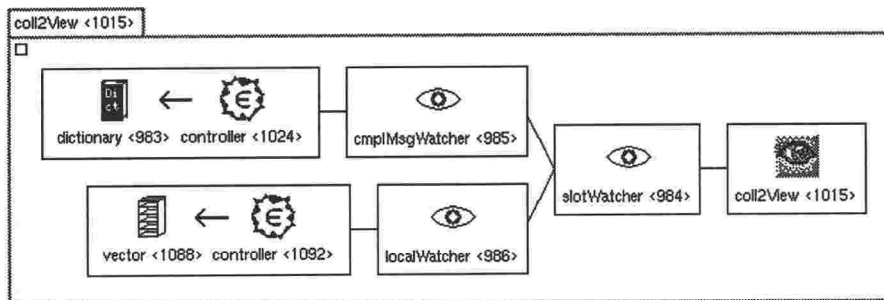
Figure 10.24: dictionary View Structure

The localWatcher is not connected to the view, rather, it is a subwatcher of slotWatcher <984>. The slotWatcher implements the indirection, and uses the cmplMsgWatcher <985> to monitor the keys slot of the target object. Should the keys slot change (i.e. another object replace vector <1088> as the dictionary's keys array), the coll2View's model and the localWatcher's target object will be changed by the slotWatcher to refer to the slot's new contents.
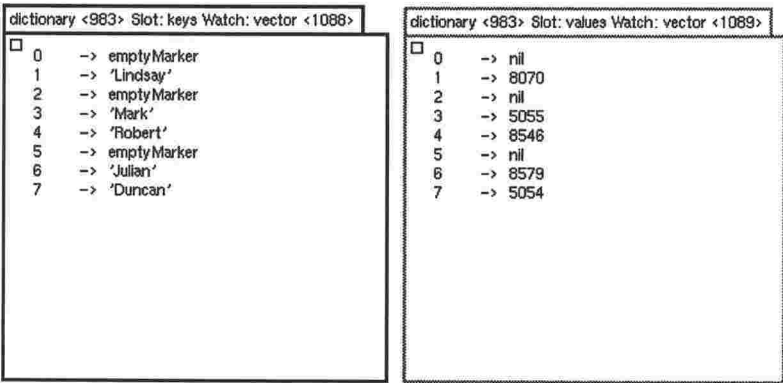
Figure 10.25 shows the controller configurations of both the dictionary and vector objects. The dictionary has four registered watchers:

- localWatcher <1025> This is used to maintain the icon browser view in Figure 10.22 (see §10.2.2).

- cmplMsgWatcher <989> This monitors the values slot for the indirect values view in Figure 10.23.

- cmplMsgWatcher <985> This monitors the keys slot for the indirect keys view in Figure 10.23. This watcher appears in the top branch of Figure 10.24.

- topWatcher <1026> This monitors the whole dictionary for the abstract collection view in Figure 10.21.

The vector's controller has only a single localWatcher registered, which is used to update the view of the array shown in Figure 10.23.

Updating the Dictionary

The dictionary can be updated by adding another name and telephone number. Figure 10.26 shows the dictionary's contents after James has been added with number 8577. Figure 10.27 shows the resulting implementation.

Compared with Figures 10.21 and 10.22, the dictionary now contains six elements. The dictionary's size slot now contains six, which is the same as the value of highMark slot before a new element was added (see Figure 10.22). The dictionary has expanded the size of the hash table to retain good hashing performance. The old keys and values vectors (with room for eight elements) have been increased in
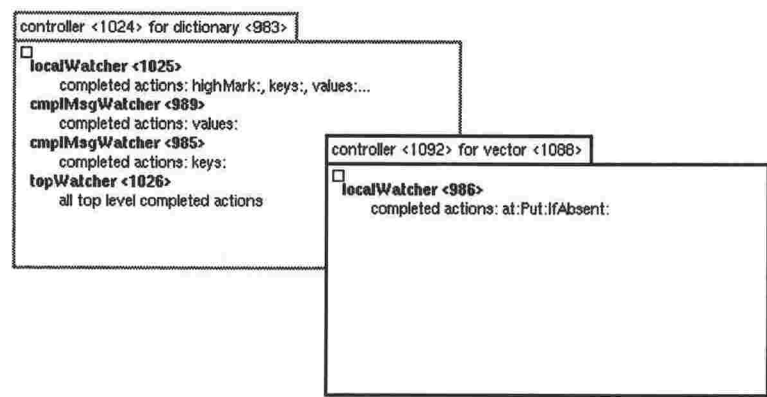
Figure 10.25: dictionary and keys Vector Controllers

Figure 10.26: dictionary after Update

length to sixteen (see Figure 10.28). Since SELF vectors are fixed size, two new vectors have been created (ID <1095> and <1096>) and have replaced the old contents of the keys and values slots respectively. The old vector's contents are rehashed into the new arrays (this is obvious in the dynamic display, as each entry is added to the parallel arrays in turn). The lowMark and highMark variables have also been increased.
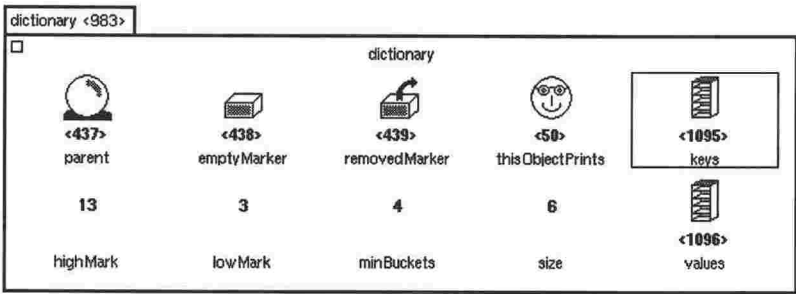
Figure 10.27: dictionary Implementation after Update

The vector views from Figure 10.23 have responded to this change, and now display the new longer vector objects, as shown in Figure 10.28. This is the result of the slotWatcher: the cmplMsgWatcher detected when the dictionary's slots changed, and the slotWatcher then reconfigured the views.

The resulting view structure is shown in Figure 10.29 (compare with Figure 10.24). The watchers and views are the same, but the target object (in the bottom left of the figure) is now the new keys vector (vector <1095>, compared with vector <1088> in Figure 10.24), and therefore has a new controller (controller <1098> as against controller <1092>).

The controller dispatch databases are shown in Figure 10.30, and may be compared with Figure 10.25. While the main dictionary object's controller has not changed, the old keys object (vector <1088>) now

dictionary <983> Slot: keys Watch: vector <1095>		
0	->	empty Marker
1	->	empty Marker
2	->	empty Marker
3	->	'James'
4	->	'Robert'
5	->	empty Marker
6	->	empty Marker
7	->	empty Marker
8	->	empty Marker
9	->	'Lindsay'
10	->	empty Marker
11	->	'Mark'
12	->	empty Marker
13	->	empty Marker
14	->	'Julian'
15	->	'Duncan'

dictionary <983> Slot: values Watch: vector <1096>		
0	->	nil
1	->	nil
2	->	nil
3	->	8577
4	->	8546
5	->	nil
6	->	nil
7	->	nil
8	->	nil
9	->	8070
10	->	nil
11	->	5055
12	->	nil
13	->	nil
14	->	8579
15	->	5054

Figure 10.28: dictionary keys and values Vectors
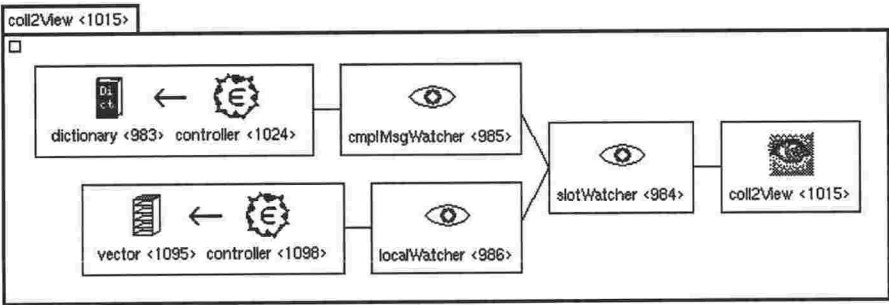
Figure 10.29: dictionary View Structure

has no interested watchers, and localWatcher <986>, which used to monitor it, is now registered with the new keys vector.
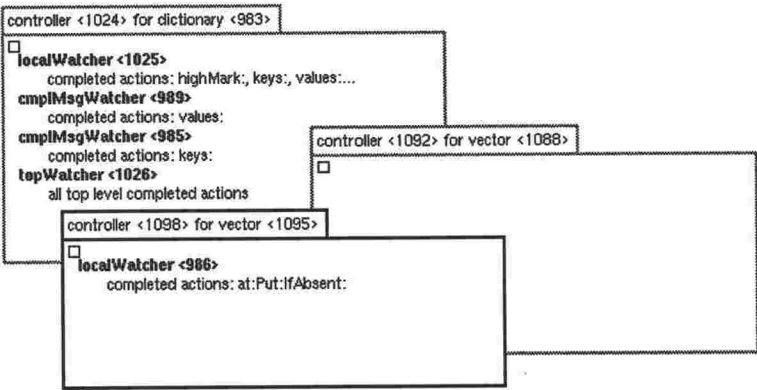
Figure 10.30: dictionary and keys Vector Controllers

10.3   Parser

The ILLUSTRATED COMPILER (ICOMP, see [7] and Section 2.3.2) displayed twenty custom views of a PL/0 compiler [229]. Both the compiler and visualisation were implemented specially in LISP, and linked by annotations. We have implemented a lexical analyser and top down parser for a PL/0-like language in SELF (ICOMP also contained a pseudocode generator, and an interpreter) and have used views from Tarraingím's library (and a couple of custom views) to illustrate our parser.

Figure 10.31 illustrates the abstraction structure of the example parser. The parser itself uses a lexer, a grammar , a stack, and a genTree (the genTree generic tree holds the abstract syntax tree constructed by the parser). The lexer in turn uses an inputStream, and an fsm comprised of states and dictionarys. The grammar is comprised of a dictionary, and rules and nodes.
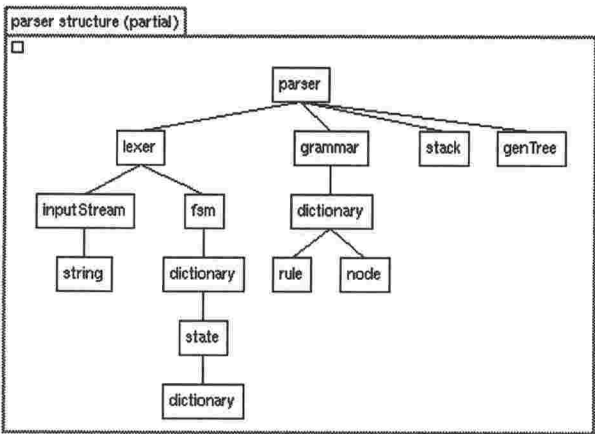
Figure 10.31: Parser Abstraction Structure

Figure 10.32 shows an icon browser displaying the parser object. The parser object includes lex, gram, stack and parseTree slots containing the lexer, grammar, parse stack and parse tree respectively, and also two auxiliary variables, printParse and keepTerminals.
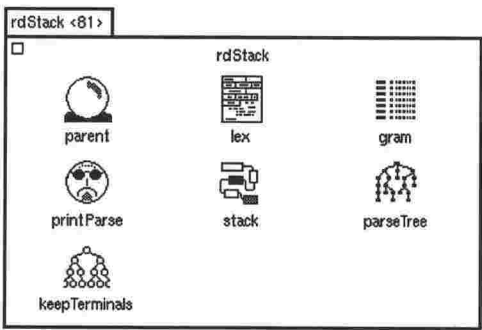
Figure 10.32: The parser

10.3.1   Lexical Analysis

Figure 10.33 displays the implementation of the lexer object. The most important slots in the lexer are the inputStream and fsm slots, containing the input to the lexer and a Finite State Machine (FSM) used to recognise tokens respectively.

Figure 10.33: The lexer

The lexer's input handling is displayed in Figure 10.34. This figure contains four views: two, titled "lexer <7> Slot: inputStream...", display the inputStream slot of the lexer, a third view displays the tokenChars slot of the lexer, and a fourth view, "lexer <7>", displays the output of the lexer.

Figure 10.34: The inputStream

The left and right hand views in Figure 10.34 are Tarraingím's standard textual trace views (§6.7). In the figure, these views are not displaying message traces. Instead, they provide traces of the inputStream's and lexer's output. These views use a custom nextWatcher, which updates its associated view when the target object finishes executing the next message. SELF streams respond to the next message by returning their next element, so that these views provide a trace of the values returned by successive calls to next. Both the inputStream and the lexer are implemented as SELF streams — the inputStream produces characters, and the lexer produces tokens. Figure 10.34 thus displays the parser's lexical analysis as an abstract stream transformation: the input to the lexer (in the left hand view) is a stream of characters, and the output (in the right hand view) is a stream of tokens.

The two centre views in Figure 10.34 display further information about the lexer's input processing. The centre lexer inputStream view displays the text of the program being parsed. The input stream's cursor position is shown by a caret: in the figure it is just after the "tot" of total. This view was adapted from the basic textView so that it retrieves and displays a caret at the cursor position.

The lexer tokenChars view displays a buffer in the lexer accumulating characters from the input stream

into the next token to be returned by the lexer. It currently contains the characters "to". Note that the input stream (displayed in the left and centre inputStream views), has actually returned three characters of total — presumably the lexer has read three characters, but transferred only two into the buffer. The tokenChars view was also adapted from the basic collection view: it simply draws elements in a horizontal rather than vertical list.

10.3.2 Finite State Machine

The main view we built specially for the parser visualisation is illustrated in Figure 10.35. This view displays the nondeterministic finite state machine used to direct the lexer.

Figure 10.35: Finite State Machine

State machine states are drawn as named rectangles; the start state (named start) is drawn with a triangle before its name, and terminal states are drawn with a double border. Trans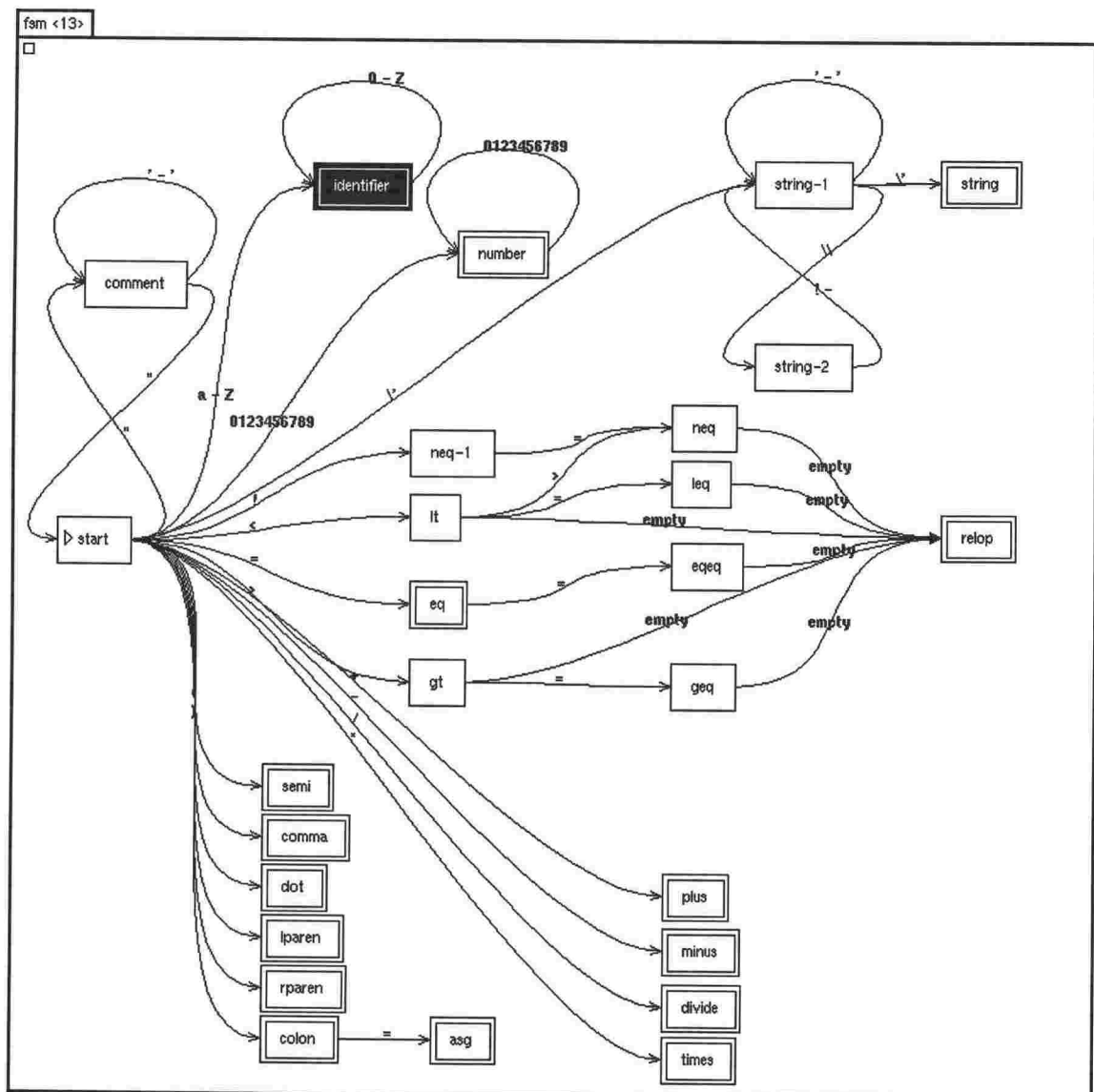itions are represented by arrows linking states and are labelled with the input characters or character ranges which cause the

transitions to be taken. The current state of the FSM is shown by a state drawn in reverse video. The FSM is shown in the state identifier, which is a final state.

This view is a hierarchical view, with essentially the same structure as the tree and icon browser views described in Section 10.2.2 and illustrated in Figure 10.17. Each FSM state is displayed by a fsmStateView, which uses a localWatcher to monitor the corresponding FSM state and sends callbacks to determine whether it is a start or a terminal state. The main FSM view draws the transition lines linking states, using auxiliary objects to position the splines and labels. State views are laid out manually by the visualiser, although any particular layout can be stored and recalled by the FSM view.

10.3.3 Grammar Rules

The grammar rules view lists the EBNF rules for the language grammar: it is a stock collView collection view (§10.2.1). The parser builds an abstract syntax tree top down. Rules marked with an asterisk "*" cause elements to be added to the abstract syntax tree.

```
dictionary <164>

rule *relation ::= expression rel expression.
rule *var ::= 'var' 'identifier' {'comma' 'identifier'} 'semi'.
rule factor ::= ['identifier' | 'number' | 'lparen' expression 'rparen'].
rule *const ::= 'identifier' 'eq' 'number'.
rule *assignment ::= 'identifier' 'asg' expression 'semi'.
rule *compound ::= 'begin' {statement} 'end' ['semi'].
rule *while ::= 'while' condition 'do' {statement} 'od' ['semi'].
rule *odd ::= 'odd' expression.
rule *call ::= 'call' 'identifier' 'semi'.
rule *program ::= 'prog' 'identifier' 'semi' {declaration} 'end' 'dot'.
rule *expression ::= ['plus' | 'minus' | ] term {['plus' | 'minus'] term}.
rule *term ::= factor {['times' | 'divide'] factor}.
rule consts ::= 'const' const {'comma' const} 'semi'.
rule *proc ::= 'proc' 'identifier' 'semi' block.
rule statement ::= [assignment | call | compound | if | while | ].
rule *if ::= 'if' condition 'then' {statement} 'fi' ['semi'].
rule block ::= {declaration} 'begin' {statement} 'end' ['semi'].
rule rel ::= ['relop' | 'eq'].
rule declaration ::= [consts | var | proc].
rule condition ::= [odd | relation].
```

Figure 10.36: Grammar Rules

The grammar rules themselves are represented as trees of grammar nodes. Figure 10.37 shows the inheritance hierarchy of grammar node objects — the basic nonNode and termNode representing terminal and nonterminal symbols, and the seqNode, selNode, it0Node, and it1Node representing sequence, selection, and iteration (zero or more times, and one or more times, respectively).

```
grammar node inheritance hierarchy

                          basicNode

  nonNode     termNode              node

                    seqNode   selNode   it0Node   it1Node
```

Figure 10.37: Grammar Node Inheritance Hierarchy
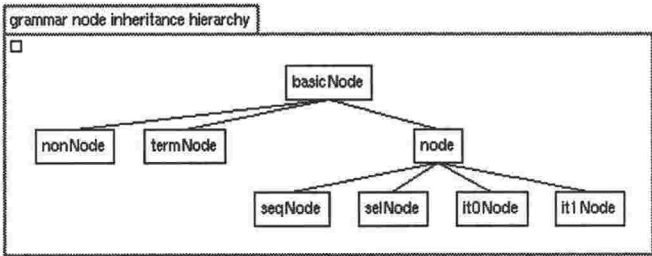
Figure 10.38 illustrates two rule trees, for expressions and terms. In Figure 10.38, nodes labelled "?" are selection nodes, nodes labelled "o" are selection nodes, and nodes labelled "*" are zero-or-more it0Nodes. The parser recursively traverses these grammar nodes, building an abstract syntax tree. Both Figure 10.37 and Figure 10.38 are constructed using Tarraingím's tree view.
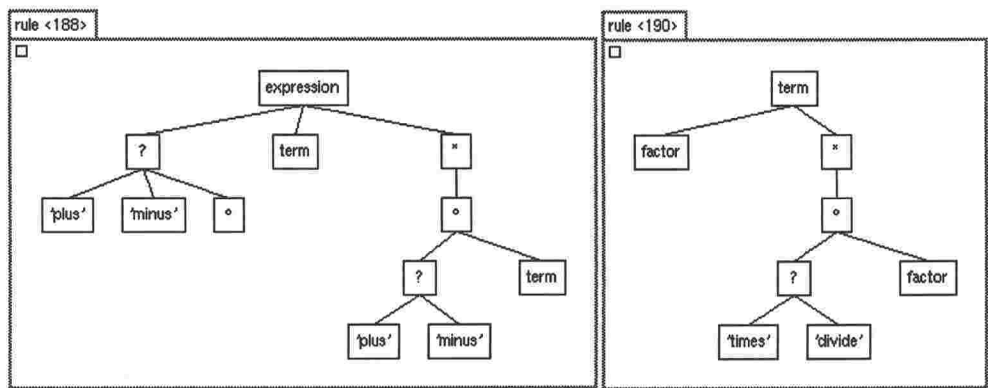
Figure 10.38: Rules for Expressions and Terms

10.3.4 Parsing

The **stack** view in Figure 10.39 displays the parse stack: it is drawn so that the top of the stack is on the left and the arrows point down the stack. Each stack item represents a single recursive call inside the parser. The top line in each stack item is the part of the grammar being processed by that recursive call. The grammar is EBNF, so this can include both nonterminal nodes and rule subnodes. The bottom line of each stack item is the portion of the abstract syntax tree being built by that invocation, which is eventually returned from the recursive call. This view is implemented by another custom version of the basic collView.
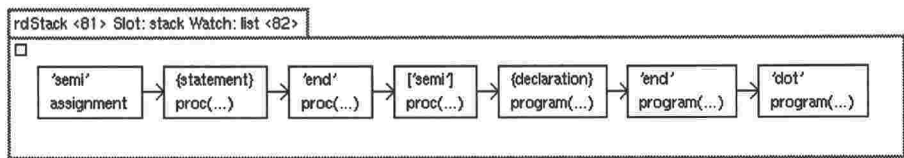
Figure 10.39: Parse Stack

The **parseTree** views in Figure 10.40 display the abstract syntax tree being constructed by the parser, at two different magnifications. The large displays are created by the generic tree view, using **treeNodeViews** and **tokenViews** in exactly the same way as the tree view described in Section 10.2.2. The smaller display is essentially the same view, but configured differently: in particular, **boxViews**, which appear as small rectangles, are used to display the nodes in place of **treeNodeViews** and **tokenViews**. The nodes in the tree correspond to the rules in the grammar marked with an asterisk.

10.3.5 Summary

While our example parser and its visualisation here are less complete than those of ICOMP, we consider that this is a matter of the time and effort available, rather than a shortcoming of the visualisation approach itself. Our display of the lexer and parser shows that our abstraction-based approach to visualisation can produce displays of similar quality to hand-crafted techniques such as those used in ICOMP. More importantly, only a few of the views presented in this section were written specially for this example: the majority were taken from our view library.

Figure 10.40: Parse Tree

*74. Is it possible that software is not like anything else, that it is meant to be
discarded: that the whole point is to always see it as soap bubble?*

Alan Perlis, *Epigrams On Programming* [168]

O is for Object, which is the granddaddy of all soap bubbles.
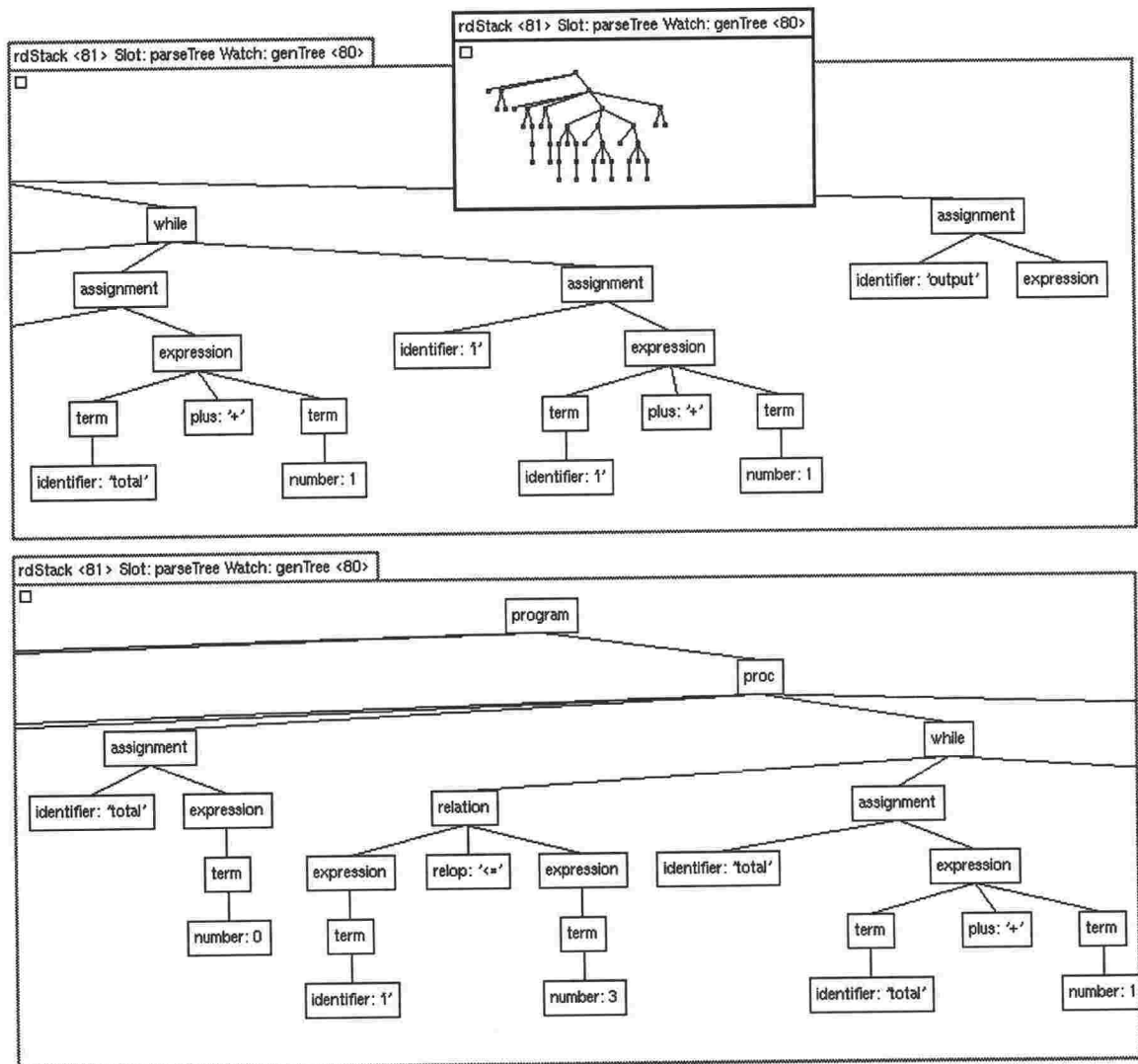
Brian Alexander, *ABC's for object-gifted children* [4]

11

Conclusions

The aim of the research described in this thesis was to investigate the use of abstraction in program visualisation. In pursuit of this aim, we developed the APMV model of abstract program visualisation, and designed and implemented the Tarraingím prototype as a proof-of-concept for that model.

The first four sections in this chapter present the major contributions of this research. Section 11.1 reviews the APMV model of abstract program visualisation. Section 11.2 describes how the model evolved during our experiments with the Tarraingím prototype. Section 11.3 identifies some implications of the use of the APMV model, and Section 11.4 reflects on our experience with the SELF language.

This thesis concludes by placing this research into the context of related work (§11.5) and outlining possible future developments (§11.6).

11.1 The APMV Model

The core of this thesis is the APMV model of program visualisation, which can produce animated views of programs at multiple levels of abstraction, including views of algorithms and views of data structures. In this section we review the most important and novel features of this model — the use of a top down approach to visualise abstractions, the visualisation of all kinds of abstractions, and the separation of objects and visualisations from the mappings between them.

11.1.1 Top Down Visualisation

The APMV model takes a *top down* approach to program visualisation: views of abstractions are produced by using the objects in the program to work from abstractions' definitions down to their implementations. These abstract visualisations display pictures specific to the target program, using images drawn from the program's domain. Section 3.8 summarised this in the following principle:

> The *pictures* we draw correspond to the *abstractions* in the *design*, which are the *objects* in the program.

Working top down, using objects which represent the program's abstraction structure, provides several intrinsic benefits for visualisation. Most importantly, abstract views depend upon the external interfaces of the abstractions they visualise, not the details of the abstractions' implementations (§3.5.2, §3.6.2). Thus, to produce abstract views, the visualiser only needs to understand abstractions' interfaces, not their

implementation details. The visualiser does not have to modify the target program to permit monitoring, because the target program can be monitored automatically. The visualiser does not have to specify which implementation procedures or data structures need to be monitored to produce a particular view, although strategies can be used to provide fine control over monitoring if required. Objects' interfaces provide a loose coupling between the program and the visualisation, which insulates the visualisation from changes in the implementation of the abstractions in the program, and vice versa.

Top down visualisation, which bases visualisations of abstractions upon the objects which represent those abstractions, is the most novel and significant idea in this thesis. It underlies the APMV model, a simple, radical model of abstract program visualisation. Indirect visualisation, which works top down to build models of abstractions that are not explicitly represented in the program, ensures the APMV model can produce most program visualisations found in the literature.

11.1.2   Visualisations of Algorithms and Data Structures

The APMV model does not distinguish between algorithmic visualisation (which illustrates the important operations of the target program, see Figure 3.2 in §3.1.1) and data structure visualisation (which illustrates the target program's data structures, see Figure 3.1 in §3.1.1). The abstractions making up a program's design, and the objects implementing those abstractions, may be algorithms, data structures, or any combination of the two. This is reflected in the equal use of two complementary connection mechanisms — changes and callbacks — in both algorithmic and data views (§3.7.4). Algorithmic views receive changes to notify them of the progress of the algorithm, and send callbacks to initiate computations or retrieve results. Data structure views similarly receive changes which describe modifications to data structures, and send callbacks to retrieve values.

The lack of distinction between algorithmic and data visualisation within the APMV model is especially important for producing abstract views. Many abstract algorithmic views require information about data structures, and data structure views similarly need algorithmic information. For example, views which primarily display algorithmic information, such as the target program's important operations and proof properties (§4.1.3), often need information about the values of data items manipulated by the program, especially those passed as parameters to the operations. Similarly, views which primarily display information about programs' data structures need dynamic information about modifications to those structures — effectively algorithmic information.

The lack of distinction between algorithm and data structure visualisation is also valuable in the implementation of indirect views. Data structure views can be used to display algorithmic information; for example, bar graph views can display operation profiles, or tree views display call trees. Algorithmic views can similarly be used to display information about data structures; for example, trace views can display histories of the values of data structures.

The complementary mechanisms of callbacks and changes also handle visualisation and graphical editing in a uniform manner. Callbacks can implement users' editing commands, assigning values or invoking target program computations, in exactly the same way that they retrieve information for display. Similarly, changes can notify other views of the consequences of editing commands in exactly the same way that they notify views of the program's internal actions.

The uniform treatment of algorithmic structure visualisation and data structure visualisation greatly increases the generality of the APMV model, while simplifying its implementation.

11.1.3   Separation of Mappings from Abstractions and Visualisations

The APMV model separates the abstractions in the target program, the visualisation of those abstractions, and the mapping strategies required to connect abstractions and visualisations. The mapping component of the APMV model manages the interface between programs and views — selecting changes, synchronising callbacks, constructing monitoring plans, and building indirect visualisations. This is in contrast to the generic PMV model (§2.1), and most other program visualisation techniques, where the

mapping component must both recover abstractions bottom up and map the resulting abstractions to meet the visualisations' requirements.

This novel separation allows the mapping component's strategies to be used for a variety of purposes — to adapt a view to the properties of the particular implementation of an abstraction, to control the amount of information a view displays, and to increase the efficiency with which the program is monitored. The separation similarly facilitates indirect visualisation because views simply display abstractions, and any indirection is left to the strategy. The same view can be used to display abstractions present in the program and to display abstractions manufactured by indirect strategies.

This separation of concerns supports the separate development, debugging, and use of views and strategies. When visualisations are being developed, the details of monitoring and synchronisation can be separated from the abstractions to be visualised and from the views of those abstractions. This separation also promotes the independent reuse of both views and strategies, since a particular view may be used with many strategies, and vice versa.

11.2 The Evolution of the APMV Model

The APMV model as presented in this thesis is the outcome of an evolutionary process — our conception of the APMV model and the design of the Tarraingím prototype evolved together over time. Starting from a simple initial version of the model, we designed Tarraingím (and, in later iterations, updated it) to implement the model. We then experimented with the resulting system by using it to construct a variety of visualisations modelled on those in the literature. At the end of each evolutionary iteration, we refined the model in the light of the experiments, adding or elaborating components to meet the needs revealed by these experiments. The views we produced in the early iterations were necessarily simple, such as the trafficLight view (§6.2). As the model and prototype were refined, we were able to experiment with progressively more complex views, such as BALSA-style sorting views (§3.1) and the browser views of the SELF world (§10.1). The experiments culminated with the parser visualisation illustrated at the end of Chapter 10.

We had two aims in working with an experimental prototype as well as an abstract model. First, we wished to verify experimentally that the model could feasibly produce abstract visualisations, and to do so required an experimental prototype. Second, we needed to determine the components required by a program visualisation system which could implement the model, and this also required experimenting with a prototype implementation. As a result of the evolutionary process, the developing APMV model has always been embodied in a system which can produce actual visualisations, and the final form of the model contains only those components required in practice.

In this section we discuss the insights provided by these experiments in more detail. In particular, we describe our initial explorations (§11.2.1), the refinement of the program component (§11.2.2) and the mapping component (§11.2.3), and our developing understanding of the use of callbacks and changes (§11.2.4) and indirect visualisation (§11.2.5).

11.2.1 Initial Investigations

The starting point for the development of the APMV model was Brown's annotation based approach [33]. Brown popularised the idea of algorithm animation, and developed annotations as a means to this end. As mentioned in Chapter 4, Brown suggested and then dismissed the approach we eventually chose — basing visualisations on abstractions represented by objects in the target program.

At this stage, as well as exploring the literature on program visualisation and program design, we experimented with several program visualisation systems, including BALSA (§2.3), XTANGO (§2.3.1), and ANIM (§2.3.2). Inspecting the designs of visualisations used in these systems, it became apparent that many of the visualisations were closely related to the important abstractions in their target programs' designs. It became equally apparent that these abstractions could not be used directly by a visualisation

system, because they could not be represented by the procedural decomposition and structured programming models used in these systems (§3.3 and §3.4). As we described in Section 3.6, these abstractions can be represented by the structure of the objects in an object oriented program.

With this background, we developed the first version of the APMV model [158, 159]. This version was quite simple — based on object orientation, it linked objects in the program which represented abstractions in the program's design directly to their visualisations. The program component of this version of the model consisted solely of encapsulators, and the mapping component was nonexistent. Based on this model, we designed and built the first version of the Tarraingím prototype with an equally simple design, consisting solely of views, encapsulators, and the objects in the target program.

11.2.2   Program Component

The first version of the APMV model was strongly centred on one particular monitoring mechanism, namely encapsulators. The corresponding version of Tarraingím relied directly upon one particular implementation of encapsulators. This model was over-specific, as the monitoring mechanism could not be changed. This problem became acute when we wished to improve the speed of the overall system by changing the encapsulators' implementation. We realised the model required a separate component which would be responsible for monitoring the target program and transmitting detected changes to the rest of the visualisation system. This component could insulate the rest of the system from changes in the monitoring mechanism.

This realisation was the genesis of the program component in the final APMV model, and we subsequently modified the Tarraingím program to reflect the revised model. In the first prototype, an encapsulator was a single object which monitored the program, dispatched the resulting changes, and provided an interface by which views could control the monitoring [160]. In the revised prototype, we reduced the function of the encapsulator objects solely to monitoring the program, and introduced controller objects to handle the dispatching and to provide the interface to the views. This version of the model proved its worth when we later needed to modify the implementation of encapsulators (due to a change in the SELF language). According to the revised model, this modification should have affected only the program component, and in the version of Tarraingím based on this model, the modifications were indeed limited to the encapsulator objects.

To summarise, we identified the need for a separate program component in the APMV model as the result of our experience with the prototype, and the benefit of the separation was borne out by our subsequent experience with the revised prototype.

11.2.3   Mapping Component

Just as the early versions of the APMV model lacked a well-defined program component, they also lacked any notion of a separate mapping component. Instead, the visualisation component communicated directly with the program component. In the early prototypes, views communicated with controllers (or directly with encapsulators, before the developments described in Section 11.2.2). This communication was very simple, as encapsulators monitored every action of their target objects and sent views change events describing every action.

After experimenting with this design, it became apparent that many views were receiving a large number of change events which were not related to the particular visualisation they were displaying, and which they therefore ignored. A large amount of the prototype's execution time was spent generating and processing these unnecessary events. Some kind of strategy mechanism was needed to reduce this overhead. We consequently expanded the monitoring system so that it could selectively monitor the actions of the objects in the program, and added monitoring strategies into views so that they could request only the monitoring that they actually required. Since, in this version of the model, the visualisation component communicated directly with the program component, we implemented the monitoring strategies within the visualisation component, that is, directly in the view objects. The monitoring strategies could not have been implemented within the program component because different views needed to monitor the same object in different ways.

After some experience with this design, we realised that many views effectively re-implemented the same strategies for selecting monitoring, and several apparently independent views differed only in the strategies they implemented. This appeared to indicate another problem with the model itself — strategies were obviously quite important, but there was no place for them in the model. To deal with this problem, we introduced the mapping component, producing the final version of the APMV model as presented in this thesis. The mapping component managed the monitoring strategies required by views, and the visualisation component was now solely responsible for handling user input and graphical output. The prototype was updated to match the revised model, with the responsibility for implementing the monitoring strategies being removed from the view objects and placed into the new watcher objects contained in the mapping component [161].

The advantages of a separate mapping component became evident in our later experimentation with the Tarraingím prototype. By factoring the common implementations out of the old view objects, we constructed a library of watchers implementing most of the common strategies (§7.6). Whenever a new visualisation needed to use one of the common strategies, the view object concerned could simply reuse a strategy object taken from the library.

The insight required to recognise the need for a separate mapping component, and the confirmation of the advantages of this component, were in large part due to our experience in using the prototype.

11.2.4 Changes and Callbacks

All the versions of the APMV model used both callbacks and changes to link visualisations to the target program. We did not appreciate, at first, the full capability of these two mechanisms and the relationship between them. We were also unclear about whether they would suffice to represent the information needed to visualise the all abstractions in a program (including both procedural abstractions and data abstractions) or whether some additional mechanisms would be required.

Based on the initial version of the APMV model, the early Tarraingím prototypes used callbacks and changes to implement simple abstract views of objects. Indeed, the first abstract view we implemented using the APMV model, the trafficLight view (§6.2), used both callbacks and changes [159]. As the rest of the model evolved, we were able to develop and experiment with more sophisticated views and strategies. Some of these views displayed algorithmic structure, such as views of the call stack, call trees, or the recursive behaviour of quicksort. Others displayed data structures, such as grammar rule views and the reflexive visualisations of Tarraingím's internal objects.

Although the experiments inspired no alterations to the treatment of changes and callbacks within the APMV model or the design of the prototype, our understanding of the model developed as a result. Originally, we had considered changes and callbacks as minor, unrelated parts of the model, which simply made the necessary link between the visualisations and the target program. We had expected that changes would be used exclusively by views displaying algorithmic structure, and that callbacks would be used exclusively by views displaying data structure.

On inspecting the implementations of many abstract views, however, it became obvious that almost all views used both changes and callbacks, irrespective of whether they were algorithmic views, data structure views, or some combination. Reflecting on this, we recognised the complementary nature of changes and callbacks, and thus the major difference between them — changes originate from the target object, while callbacks originate from the view (§3.7.4). We also realised that no additional mechanisms were needed to link visualisations to the target program — changes could represent all the procedural actions of every object within the target program, and callbacks could retrieve all the data contained within every object. For these reasons, rather than treating callbacks and changes merely as interesting artifacts, we now consider them as important parts of the whole APMV model. The discipline of building and experimenting with the prototype was instrumental in helping us to recognise the sufficiency of callbacks and changes, and their significance within the APMV model.

11.2.5   Indirect Visualisation

The development of our understanding of indirect visualisation parallels that of callbacks and changes. All versions of the model have supported some form of indirect visualisation, but we did not fully appreciate the utility and generality of this part of the APMV model. The initial indirect visualisations were modelled on the indirect visualisations supported by BALSA [33]. They were basically algorithmic views which used indirection to maintain a database of historical information. For example, a trace view's database would contain the last few actions of the view's target object, while a profile view's database would contain a count of the number of times each action was executed.

As we developed more sophisticated views, we were able to experiment with more powerful algorithmic indirect visualisations, such as views of call trees, but also with indirect visualisations based on data structures, such as histories of data structure accesses and views with models determined by reference (§7.4.2). Eventually, after studying the indirect visualisations we had built, we realised that indirect visualisation was a very general technique, although building an indirect visualisation could be quite complex in practice. Ultimately, an indirect visualisation could monitor every action of every object in the target program, then construct an indirect model to support any computable abstraction of the target program. Because of the generality of indirect visualisations, the APMV model can produce essentially any visualisation which can be produced by another visualisation technique, if necessary by incorporating the other technique directly into the model (§4.3.4). Indirect views thus ensure that the APMV model is not limited by its reliance on the interfaces of objects in the target program.

The experimental prototype again proved very useful in investigating indirect visualisation. Even though the actual design of the model was not changed as a result, the experiments the prototype made possible gave us insight into the strengths and weaknesses of this part of the model, and increased our confidence in the model as a whole.

11.3   The Implications of the APMV Model

The APMV model provides the most leverage when program abstractions and operations upon those abstractions are captured as objects and messages in the target program's design. In our experience with Tarraingím, we have found this style of design quite practical — after all, clear representation of design abstractions is a major tenet of good object oriented design [26, 227]. Probably for this reason, we have found little problem visualising objects which were not designed with visualisation in mind, such as the collection objects in the SELF library.

Unfortunately, good object oriented designs are not always immediately obvious, and can depend upon some quite subtle details. For example, in our design of the example parser (§10.3), the decision to structure both the input stream and lexer as independent SELF objects is very important, as the input stream and lexer objects then represent the input stream and lexical analyser abstractions in the program's design. Making both these objects SELF stream objects captures the commonality between the two abstractions, and this in turn facilitates the reuse of Tarraingím's trace views. This feature of the design was not obvious to us before we started work on the parser example, although it is very similar to more recent work on object oriented parser design [1].

The corollary to good object oriented design being required for program visualisation is that a program which is amenable to visualisation is likely to be well designed. Therefore, considering possible visualisations of a design may be a way to improve it. When designing a program, a programmer can investigate possible visualisations of that program, and can improve both the program design and the visualisation design as a result. Similarly, a programmer can experiment with visualisations when re-designing an existing program. Based on our experience with Tarraingím, an APMV model visualisation system is a very good tool to support this development style.

Of course, some kinds of object oriented designs provide the APMV model with less leverage than others. In particular, the applicability of the model can depend quite heavily on some aspects of the style by which objects capture abstractions (§3.2.3). Two aspects of style which we have found particularly important for the APMV model are aliasing and retrievability.

11.3.1 Aliasing

Aliasing is endemic in the design of most object oriented programs [102]. Aliasing has the potential to make program visualisation via the APMV model very difficult, if objects are modified via aliases, rather than by sending messages via their interfaces (§4.4.1).

We have not found aliasing to be a problem in practice (§7.6.4). This is mostly due to the programming style used in Self, which generally avoids modifying objects through aliases. Self also uses immutable objects (which are immune to problems caused by aliasing) much more often than other object oriented languages such as Smalltalk or C++. For example, Self's point and most commonly used string objects are immutable. In general, it is clear that program designs which make less use of aliasing and more use of immutable objects will be easier to visualise using the APMV model.

11.3.2 Retrievability

The APMV model uses callbacks to retrieve abstract information from target program objects (§3.7). For callbacks to be effective, the target objects' interfaces must provide messages which can be used to process callbacks — that is, target objects must implement accessor messages which return abstract information without changing the object (§4.1.4).

Unfortunately, objects do not have to provide such messages if the overall design of the target program does not require them. Many useful objects provide access to the information they store only via messages which change that information. For example, Section 4.1.4 discussed an interface for a stack object where the stack's top element was the only element directly accessible — other elements could only be retrieved by popping the elements above them off the stack, thus changing the stack.

The APMV model provides the visualiser with several alternatives when information cannot be retrieved directly from a target object via its interface. The simplest alternative is to design an APMV top-down view which only displays information which can be retrieved via accessor messages — for the stack discussed above, such a view would show only the top element. This kind of view can be created easily, as the visualiser can consider only the object's interface and does not need to understand the object's implementation in detail. Also, the programmer may intentionally have decided to restrict the information retrievable via an object's interface. A view displaying only the information which can be retrieved via accessor messages has the advantage that it respects the programmer's intention by not displaying restricted information.

The APMV model can also be used to build indirect views which display more information to the user. Because indirect views can perform any computation, they can construct essentially any visualisation of an object. In particular, an indirect view can build an abstract model containing all the information stored in an object, including information which is not retrievable through that object's interface, either by directly accessing an object's implementation data structures or by tracing the object's actions and simulating their effects. Of course, to build these kinds of views the visualiser needs to understand the implementation of the abstraction in detail.

Finally, the programmer or visualiser could modify the target program to add one or more accessors to objects which to not provide them. Modifying objects has several disadvantages — the visualiser must understand the implementation of the objects in detail and have the skill to modify them, the modifications (if incorrect) can create new bugs in the objects, and, if the accessors were intentionally omitted from the program, adding accessors may destroy the integrity of the objects' original design (especially if they are later used in a revised version of the program). The advantage of modifying objects to add accessors is that once the accessors have been added, the objects can be visualised easily by APMV model views which depend upon the extended interfaces.

This is a good example of the possible feedback between visualisation design and program design. The programmer can choose to facilitate abstract visualisations that display particular information by providing accessors which return that information, even if the accessors are not required by the overall design of the program. Similarly, the visualiser can choose to simplify the visualisation design by displaying only the information which can be retrieved directly, or can choose to display more information by constructing more complex indirect views.

In our experience visualising SELF programs, we have not found retrievability to be a large problem. Following SMALLTALK, SELF objects typically have rich interfaces which provide accessor messages which can be used to retrieve all the abstract information stored in the objects without side effects. It is clear that a programming style which makes objects' information retrievable via accessors provides good support for the APMV model.

11.4   The Influence of SELF

As Chapter 5 describes, we selected SELF as the implementation language for Tarraingím. This decision was motivated as much by theoretical concerns as by the practical considerations involved in building a prototype system. We chose SELF because, as a pure object oriented language, it would support target programs written in an object oriented style, and it would ensure that only one visualisation mechanism would be needed to provide both concrete (language level) and abstract views. As a prototype-based language, we believed SELF would be easier to monitor than class-based languages such as SMALLTALK.

Overall, the choice of SELF worked out well in practice. Tarraingím's target programs have to be structured using objects, since objects are the only program structuring feature provided by SELF. SELF's object library, especially the collection objects, were useful in building Tarraingím and the example programs, and also provided good stand-alone test cases for abstract views.

The APMV model was influenced by SELF's philosophy [198]. Most importantly, SELF's unification of variable accessing, control flow, and computation into message passing [218] pointed the way towards the APMV model's unification of algorithmic and data structure visualisation, and its independence of the levels of abstraction in the target program. SELF's emphasis on behaviourism — *"an object is completely defined ... by how it responds to messages"* [2] — is similar to our emphasis on treating objects as abstractions, although we see abstractions as encompassing both algorithms and data, rather than subsuming data into algorithmic behaviour.

On the other hand, the APMV model diverges from SELF's philosophy in some important respects. The APMV model focuses on the abstract, while SELF focuses on the concrete. This is seen in other visualisation systems for SELF, such as SEITY [45, 46] and the SELF UI [197], which have been developed by the SELF group in conjunction with the language. Where the APMV model displays objects as abstractions, the SELF systems display objects as slots and slot contents. While Tarraingím's graphic design is quite spartan — minimal, black and white, and two dimensional — the SELF systems are very rich, using colour, three-dimensional objects, and smooth animation. Although using a rich graphical design, the SELF systems are conceptually quite minimal — every object has at most one graphical image, to maintain the illusion that every image represents a single language level object. In contrast, Tarraingím adopts a minimal graphical language to a much more complex end: there is no one presentation of an object, just a multitude of graphical perspectives and levels of abstraction.

Some aspects of SELF's design cause Tarraingím problems, especially concerning program monitoring (the design of encapsulators) and as a result, monitoring SELF programs proved to be more difficult than we expected. For example, SELF's primitive messages (§9.3.2) bypass the message lookup algorithm. They cannot be monitored by encapsulators, and in general they make it impossible for SELF programs to provide any guarantees about their behaviour, as primitives can be arbitrarily applied to objects. Restricting primitives so that they could only be sent to self, effectively ensuring they would always be used from within wrapper methods attached to the object they affect, and then providing a suitable set of wrapper methods in the SELF library, should resolve this in practice, but would require modification of the SELF language definition and the SELF VM.

SELF's support for inheritance and delegation is also problematic. SELF supports both dynamic inheritance (an object's parent slots can be variable, §5.4.4) and explicit delegation (the object to which a message is delegated is evaluated at runtime, §9.2.1). SELF's support for explicit delegation in particular is very tentative, and the interactions between delegation, inheritance, and privacy are not resolved well. SELF programs overwhelmingly use only inheritance, and so bypass this problem. We have used both delegation (§9.2.2) and dynamic inheritance (§9.2.3) to construct encapsulators, but nevertheless consider that both facilities could be better integrated into the core SELF language.

The version of SELF we used did not include graphical display facilities, and we assumed that providing graphical support would not prove to be too problematic. In practice, we built several versions of the NAVEL graphics library to supply these facilities (§6.1.3). Tarraingím would obviously benefit from a better interface library, which could provide support for colour, three-dimensional graphics, sound, and more advanced input devices [97]. Since presentation and interaction design were peripheral to this project's main aim of investigating abstraction in program visualisation, the lack of these facilities has not impeded our research.

The SELF language was itself being developed concurrently with the work described in this thesis, and some additions were made specifically to assist this project. The most important of these was unwind protection, which was added to enable Tarraingím to catch non-local returns, thus ensuring controllers can maintain the correct depth (§8.3.3 and §9.1.2). Extra behaviour was also added to the standard library mirror objects to identity immutable objects. In the course of developing Tarraingím, we also identified several bugs and limitations in the compiler and runtime system, which the SELF group subsequently repaired. Section 9.2.5 describes how our choice of encapsulator implementation was ultimately determined by details of the current dialect of SELF.

Overall, we found SELF a good language within which to experiment. We consider that using any of the other languages we evaluated would have led to more serious problems than those we encountered with SELF (§5.2.3). The simplicity of SELF's design, and especially the uniform use of objects and messages, contributed markedly to the success of this research.

11.5 Comparison with Related Work

In this section we compare the APMV model (and the Tarraingím prototype) with the related work described in Chapter 2 — visual programming tools (§2.2), and algorithm animation systems based on annotation (§2.3) or mapping (§2.4). Graphical debuggers (§2.2.1) are discussed separately, since Tarraingím's architecture is closer to recent object oriented graphical debuggers than to other abstract animation systems.

11.5.1 Visual Programming Tools

Visual programming tools such as programming environments (§2.2.2), visual programming languages (§2.2.3), and reverse engineering systems (§2.2.4), provide many examples of the possibilities offered by graphical views in interactive environments for software development. Visual programming tools have some important strengths — they can be applied without special preparation of the target program, they can display fine details of algorithmic and data structure, and they can be used to create and edit programs from scratch. The focus of visual programming tools is the text, structure, and properties of the program itself — they display the program's code and structure, and these displays can be edited to change the program in the course of development.

In contrast, the focus of abstract program visualisation in general, and the APMV model in particular, is upon providing abstract views which are updated as the program runs. The major advantage of these abstract views is that they display abstractions important to the user at the level of abstraction that the user understands, rather than the level of the program source text.

Displays produced by visual programming tools can be considered reflexive, in that they display the properties and structure of the target program, rather than abstractions from the target program's domain (§2.5.5). The APMV model can produce these kinds of displays, by visualising the program's objects reflexively, rather than directly. That is, callbacks can be used to retrieve meta-level information about objects' structures and properties, rather than base-level information from the objects, and changes can represent modifications to the objects' structures, rather than the actions of the executing program. As examples, Tarraingím's view library includes several views of program's structure which are implemented using mirror objects to reflect on target program objects (§5.4.5). These include the structural and profile views from Figure 3.4, views of the inheritance hierarchy (including Figures 6.1 and 7.1), and the object browser and method source views from Figure 10.6.

As a result, VICK cannot visualise SMALLTALK's collection classes, probably the most important part of SMALLTALK's library.

Gelo

GELO [183] (see also §2.2.1) is a data structure and program visualisation tool embedded within the FIELD programming environment and build on top of FIELD's distributed message-passing architecture [182, 180] (see also §2.2.2). In FIELD, information about a program's execution is gathered by a debugger, and so the user does not need to annotate the target program's source code (§2.5.2). GELO uses information from the debugger to visualise programs' data structures, presenting default displays based on data types and custom displays designed by the visualiser using a graphical editor.

GELO and the APMV model have quite different models of the target program. GELO considers the program as a collection of structured data type instances laid out in linear memory, and displays those data types. In contrast, the APMV model considers the target program as a collection of interacting abstractions, and displays those abstractions and their interactions, including algorithmic events and changes to data structures. Thus Tarraingím's views can be defined top down, in terms of the definitions of the abstractions in the program, while GELO's views must be defined bottom up, in terms of abstractions' implementations. With respect to the PMV model, GELO does not really have a mapping component, because the graphical editor defines views directly in terms of the program's implementation, similarly to direct annotation systems (§2.3.3). GELO also amalgamates abstraction recovery, monitoring strategies, and view definition, where the APMV model separates these concerns.

Cerno-II

CERNO-II [72, 73, 74] (see also §2.2.1) is a graphical debugger implemented as part of the SPE programming environment [89, 91]. CERNO-II and SPE are written in SNART, a hybrid object oriented language based on PROLOG, and CERNO-II visualises programs written in this language. Since SNART is a hybrid language, it includes structured data types such as PROLOG's lists and terms, as well as objects. SNART is also computationally reflexive (§2.5.5), and CERNO-II uses SNART's reflexive extension to monitor the target program.

CERNO-II has a three layer architecture, consisting of a program layer, an abstraction layer, and a display layer. The three layers correspond to the three components of the PMV model, and are implemented as an object oriented framework. The program layer holds the target program, and includes SNART objects, PROLOG data structures, and the call stack. The abstraction layer is made up of *abstractor* objects, which monitor objects in the program layer and produce a *display list*. An abstractor can monitor a single SNART object directly, or can combine the output of one or more subordinate abstractors. The display layer consists mainly of *icon description* objects, which interpret the display list and create SPE graphical objects for display to the user. SPE's change propagation mechanism, *update records* [90], communicates changes between abstractors, and ensures the display is redrawn whenever the display list is modified. CERNO-II has been extended with the SKIN visual language, so that new visual designs can be created by visual programming [105].

The most important difference between CERNO-II and the APMV model is their support for abstract visualisation. CERNO-II is essentially a bottom up system: abstractors treat objects as structured data types, and have to recover abstractions from their implementations. In contrast, the APMV model primarily works top down, visualising abstractions based on object's interfaces. For example, CERNO-II's visualisation of a collection object (e.g. a SNART list) is built from a large number of abstractors, which know about the implementation of the list in detail. Tarraingím's view of a SELF collection object uses one watcher for the whole collection, and that watcher is only aware of the abstract collection interface.

The APMV model is based upon changes and callbacks — i.e. monitoring messages received by target objects, and sending messages to target objects. CERNO-II's abstractors can monitor messages within the target program, however, this facility is used only to produce algorithmic displays, such as timing diagrams and road maps [104]. Similarly, although CERNO-II includes a protocol for changing displayed data, icon

descriptions do not typically send callbacks to their target objects. Like GELO, when visualising data structures CERNO-II only monitors assignments — it does not use algorithmic monitoring to optimise the visualisation of data structures. Thus, although CERNO-II's facilities are about as powerful as Tarraingím's, they are used in much more straightforward ways. CERNO-II's architecture is sufficiently flexible, however, that abstractors could be written to use changes and callbacks to recover program abstractions, effectively implementing the APMV model within CERNO-II.

11.5.5   Summary

The most important difference between the APMV model and the work discussed in this section and in Chapter 2 is that the APMV model uses a top down approach to visualisation. Program abstractions are visualised based on the interfaces of the objects representing them, so a visualiser does not have to understand an abstraction's implementation in detail to be able to visualise it.

A second difference is that the systems reviewed separate the visualisation of data structure from the visualisation of algorithmic structure, especially where abstract visualisation is concerned. The APMV model can use algorithmic information even if it is producing a view of a data structure, and vice versa; in fact, these cannot really be distinguished. The APMV model's complementary mechanisms of callbacks and changes also integrate visualisation with graphical editing.

Finally, the APMV model separates the strategies used for monitoring programs from the rest of the visualisation system. Those systems reviewed in this section which provide abstract visualisation include one component which both recovers information about program abstractions and controls how the program is monitored to recover that information.

11.6   Future Work

In this section, we describe some possibilities for the further evolution of the APMV model and the Tarraingím prototype.

11.6.1   An Alternative Programming Language

Although SELF is becoming recognised in the object oriented programming language research community, it is not used in commerce or industry. It would be useful to investigate the reimplementation of the APMV model in an alternative programming language. The APMV model is language independent, as it relies only upon broad assumptions of object oriented structure in the target program. The design of Tarraingím's framework is also not particularly SELF-specific.

Pure object oriented languages such as SMALLTALK or DYLAN, which are at least structurally reflexive, should support the construction of a system very similar to the Tarraingím prototype described in this thesis. Transferring these ideas into a more conventional language such as C++ would make an interesting experiment.

11.6.2   Watcher and View Trees

Tarraingím's watchers and views are organised into trees. This can cause a duplication of effort. For example, both a bargraph view and a textual collection view could be used to display operation profiles of the same object. Each view will be attached to the same type of watcher, subwatcher, and profile tally, but each view will use its own copies of these objects, and this results in parallel watcher trees containing the same kinds of watchers attached to the same target object. The profile will be calculated twice, once for each view.

This duplication could be avoided if watchers and views could be shared, that is, if watchers and views could be organised as directed acyclic graphics instead of trees. A single instance of each kind of

watcher or view would be constructed for each target object. If the same kind of watcher or view was requested a second time for that object, the existing watcher or view would be substituted.

In the case of watchers, this change should be quite simple to implement. Watchers would have to keep track of multiple parent watchers rather than one (for example, upEvents would have to hold a set rather than a single object, §7.1.4). The interface between watchers and views (§7.1.3) would have to change, with views registering themselves with a suitable watcher, rather than simply creating and using watchers directly.

This change would be more complicated to implement in the case of views, primarily because **X** windows must form a strict hierarchy. The display subsystem would need to be altered to allow a single view to be displayed in several different **X** windows.

11.6.3   Composite Views

Tarraingím would benefit from more support for composite views — that is, for views which simply group several other views, such as the stack implementation view in Figure 3.3. In the current version of Tarraingím, these views must be specially written by the visualiser. A generic composite view, parameterised by a list of views and some layout information would allow these types of composite views to be created without additional programming.

Composite view support would also allow some existing views to be built solely by composition. Consider the tlView traffic light view, described in Section 6.2. This view consists of three rectangles, one of which is filled to represent the traffic lights active aspect, and is currently specially written (see Figure 6.5). The tlView could be built by a composition of three booleanViews, each attached to a different aspect of the traffic light via the isRed, isAmber, and isGreen messages, which all return a boolean value.

11.6.4   Views of Multiple Objects

The APMV model provides little support for views of the implementation of an ADT, or overviews of the collaboration between several objects which are not simply components of another ADT. The implementation views shown in this thesis are typically simple composite views of each of the objects, for example, see the stack implementation view in Figure 3.3, or the FSM view in Figure 10.35. This is because the APMV model and Tarraingím's framework assume each view is primarily linked to one object in the target program. Some implementation views need to produce an integrated display showing several objects, and do not have one particular target object.

Two features of the APMV model — watchers and hierarchical views — could be extended to support views of multiple objects. We discuss each of these in turn.

Multiple watchers (§7.5) allow several objects to be associated with a single view. This view will receive information about the changes of all of these objects, and can use these to produce a display like any other view. The view can combine information about all objects of interest to produce an integrated display. The graphical display is completely unconstrained by the objects used to produce it, because it is drawn by a single view. This approach has two disadvantages: a custom multiple watcher will probably have to be written to monitor the correct set of objects, and the view is monolithic — all the individual objects are displayed by a single view definition.

Hierarchical views (§6.5) use multiple subviews to display multiple objects — each object is displayed by a single subview. Hierarchical views are more flexible than monolithic views, for example, the display of a particular object can be changed without affecting the rest of the view. Hierarchical views are also easier to write than monolithic views, as the visualiser can write and debug each subview without reference to the rest of the view. In a hierarchical view, each subview is displayed in an independent **X** window.

11.6.5 Mapping Component

The most promising avenue for the further development of the mapping component is the design of watchers which use information about the target program to implement higher-level strategies, and adapt their monitoring to suit the details of their target objects. For example, the localWatcher (§7.2.4) monitors changes to an object's local slots. A localWatcher inspects its target object, determines that object's variable slots, and then requests that these slots are monitored. Any constant slots in the target object are ignored, and so a target object which contains no variable slots (such as the unordered category object §10.2.2) will not be monitored.

More sophisticated strategies could use other information about the target program in similar ways. Consider the topWatcher (§7.2.3), which is widely used within Tarraingím to synchronise views which use callbacks (§4.2.1). This strategy is conservative, because it reports a change whenever a top level message returns, whether or not that message has actually modified the target object. A more sophisticated topWatcher could use information about accessor and mutator messages to optimise its monitoring, in the same way a localWatcher uses information about constant and variable slots (§4.3.1). We call this kind of watcher a globalWatcher, because it reports changes like a localWatcher, but for the whole of the abstraction represented by its target object. A globalWatcher would ignore accessor messages (that is, the globalWatcher would not request monitoring of these messages) while mutator messages would be reported as changes to the view.

A globalWatcher could be implemented in the current version of Tarraingím, but the visualiser would have to explicitly list the messages to be monitored, as in a changeWatcher (§7.2.3). A generic globalWatcher would require some way to dynamically determine which messages of its target object are mutators and which are accessors. SELF does not make this distinction, unlike C++ and EIFFEL. New versions of SELF include slot annotations which could be used to carry this information.

11.6.6 Program Component

The addition of custom encapsulators (§9.2.4) is the most obvious future extension to the program component. This should not be too difficult, as custom encapsulators are essentially a variation on the inheriting encapsulators which are currently used by Tarraingím. Implementing custom encapsulators would require some minor modifications to the procedure that creates inheriting encapsulators' wrapper methods, so that only those wrapper methods required by a particular custom encapsulator are created, instead of one wrapper method for each message in the SELF system. Controllers' dispatch databases contain all the information required to use custom encapsulators (§8.1.3).

The monitoring system's architecture could also be changed to represent monitoring plans explicitly (§4.3). Monitoring plans are currently represented implicitly in Tarraingím, being transmitted from watchers to controllers via the messages in the controller registering protocol (§8.1.2). Explicit monitoring plan objects would simply package the information carried by this protocol. A watcher would construct a monitoring plan object, initialise it to describe the watcher's monitoring requirements, and then send the monitoring plan object to the view. The controller registering protocol would be reduced to a single message accepting a monitoring plan object as an argument, and a simpler version of this protocol would be used to initialise monitoring plan objects.

11.6.7 Syntax for Visualisation

Tarraingím does not provide any textual syntax for representing visualisations (that is, for describing trees of views and watchers). The visualiser must therefore construct these trees piece by piece. For example, the profile view from Figure 10.19 must be constructed as follows:

```
| pw. barProfile |

pw: profileWatcher copy.
pw auxWatcher: recvWatchger copy.
```

```
pw mainWatcher: changeWatcher copy.

barProfile: barGraph copy.
barProfile watcher: pw.
barProfile name: 'bar profile'.
```

We have not found this a problem in practice, because we have generally stored all the component objects separately in the SELF library. This has led to a proliferation of objects differing only in parameter settings. For example, we have ten nTreeView prototypes and five browserView prototypes which differ only in the type of view to use to display tree nodes or browser slots respectively.

If Tarraingím was extended to include a more declarative syntax for combinations of watchers and views, the visualiser could describe a particular view by directly parameterising a prototype. For example the barProfile view above could be described as follows:

```
| barProfile |

barProfile:
    ('name' ← 'bar profile') &
    ('watcher' ←
      profileWatcher:
          ('aux' ← traceWatcher) &
          ('main' ← changeWatcher)).
```

This syntax would allow the visualiser to construct correctly parameterised trees of views and watchers declaratively, instead of constructing them piece by piece from specially parameterised prototypes.

Visual Syntax

Tarraingím is a visual system, so a visual syntax for visualisations would be preferable to a textual syntax. A visual syntax could be provided by extending the existing view structure views and property sheet views. The existing property sheets can set properties which are strings or numbers, such as a view's name, or the size of dots in a scatterplot. Some properties, such as a view's watcher, need to specify arbitrary types of objects, and extending property sheets to handle arbitrary objects should not be difficult. The view structure view could similarly be extended to give direct access to the property sheets of the watchers and views it displays, and to allow a view's structure to be edited directly.

If Tarraingím was extended to support composition (§11.6.3), composite views could be constructed using a graphical user interface. Composite views could be built indirectly, using their property sheets which would describe the composite view's subviews, or directly, by allow existing views to be selected and "dragged-and-dropped" into the composite view.

31. Simplicity does not precede complexity, but follows it.

Alan Perlis, *Epigrams On Programming* [168]

Bibliography

[1] Ole Agesen. Mango - A parser generator for Self. Technical Report TR-94-27, Sun Microsystems Laboratories, 1994.

[2] Ole Agesen, Lars Bak, Craig Chambers, Bay-Wei Chang, Urs Hölzle, John Maloney, Randall B. Smith, David Ungar, and Mario Wolczko. *The Self Programmer's Reference Manual.* Sun Microsystems and Stanford University, 4.0 edition, 1995.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, Inc., 1986.

[4] Brian Alexander. ABC's for object-gifted children. *ParcPlace Newsletter*, 5(1), Spring 1992.

[5] Allen L. Ambler and Margaret M. Burnett. Influence of visual technology on the evolution of language environments. *IEEE Computer*, 22(10), October 1989.

[6] American National Standards Institute, Inc. *The Programming Language Ada Reference Manual*, volume 115 of *Lecture Notes in Computer Science.* Springer-Verlag, 1983.

[7] Kristy Andrews, Robert Henry, and Wayne Yamamoto. The design of the UW illustrated compiler. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, 1988.

[8] Robert S. Arnold, editor. *Software Reengineering.* IEEE Computer Society Press, 1993.

[9] Gideon Avrahami, Kenneth P. Brooks, and Marc H. Brown. A two-view approach to constructing user interfaces. *ACM Computer Graphics*, 23(3), July 1989.

[10] Ronald M. Baecker and David Sherman. Sorting Out Sorting. 16 mm colour sound film, 1981. Shown at ACM SIGGRAPH Conference.

[11] Henry G. Baker. Equal rights for functional objects or, the more things change, the more they are the same. *OOPS Messenger*, 4(4), October 1993.

[12] Marla J. Baker and Stephen G. Eick. Visualizing software system. In *IEEE International Conference on Software Engineering*, May 1994.

[13] Ed Baroth and Chris Hartsough. Visual programming in the real world. In Margaret M. Burnett, Adele Goldberg, and Ted G. Lewis, editors, *Visual Object-Oriented Programming.* Prentice-Hall, 1995.

[14] Kent Beck. Pictures from the object explorer. Technical report, First Class Software, Inc., 1994.

[15] Brigham Bell and Clayton Lewis. ChemTrains: A language for creating behaving pictures. In *IEEE Symposium on Visual Languages*, 1993.

[16] John K. Bennett. The design and implementation of Distributed Smalltalk. In *OOPSLA Proceedings*, 1987.

[17] Jon L. Bentley. Pictures of programs. In *UKUUG Summer Proceedings*, 1990.

[18] Jon L. Bentley and Brian W. Kernighan. A system for algorithm animation. *USENIX Computing Systems*, 4(1), Winter 1991.

[19] Thomas Berlage. Using taps to separate the user interface from the application code. In *Proc. ACM Symposium on User Interface Software and Technology (UIST)*, 1992.

[20] Ted J. Bickerstaff. Design recovery for maintenance and reuse. *IEEE Computer*, July 1989.

[21] Ted J. Bickerstaff, Bharat G. Mitbander, and Dallas E. Webster. Program understanding and the concept assignment problem. *Communications of the ACM*, 37(5), May 1994.

[22] Heinz-Dieter Böcker and Jürgen Herczeg. TRACK — a trace construction kit. In *Human Factors in Computing Systems (ACM CHI Conference Proceedings)*, 1990.

[23] Heinz-Dieter Böcker and Jürgen Herczeg. What tracers are made of. In *OOPSLA Proceedings*, 1990.

[24] Heinz-Dieter Böcker and Michael Pawlitschek. VICK: a visualization construction kit. *Journal of Object-Oriented Programming*, January 1992.

[25] Grady Booch. *Software Engineering with Ada*. Benjamin Cummings, 1983.

[26] Grady Booch. *Object Oriented Analysis and Design with Applications*. Benjamin Cummings, second edition, 1994.

[27] Isabelle Borne. A visual programming environment for Smalltalk. In *IEEE Symposium on Visual Languages*, 1993.

[28] Alan Borning. Thinglab — a constraint-oriented simulation laboratory. Technical Report SSL-79-3, Xerox PARC, July 1979.

[29] Alan Borning. The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4), October 1981.

[30] Alan Borning and Robert Duisberg. Constraint based tools for building user interfaces. *ACM Transactions on Graphics*, 5(4), October 1986.

[31] Ivan Brakto. *Prolog Programming for Artificial Intelligence*. Addison-Wesley, 1986.

[32] Frederick P. Brooks, Jr. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20(4), April 1987.

[33] Marc H. Brown. *Algorithm Animation*. ACM Distinguished Dissertation. MIT Press, 1988.

[34] Marc H. Brown. Exploring algorithms using Balsa-II. *IEEE Computer*, 21(5), May 1988.

[35] Marc H. Brown. Perspectives on algorithm animation. In *Human Factors in Computing Systems (ACM CHI Conference Proceedings)*, 1988.

[36] Marc H. Brown. Zeus: A system for algorithm animation and multi-view editing. In *IEEE Workshop on Visual Languages*, October 1991.

[37] Marc H. Brown. The 1993 SRC algorithm animation festival. In *IEEE Symposium on Visual Languages*, August 1994.

[38] Marc H. Brown and John Hershberger. Color and sound in algorithm animation. *IEEE Computer*, 25(12), December 1991.

[39] Marc H. Brown and Robert Sedgewick. A system for algorithm animation. *ACM Computer Graphics*, 18(3), July 1984.

[40] Bernd Bruegge, Tim Gottschalk, and Bin Luo. A framework for dynamic program analyzers. In *OOPSLA Proceedings*, October 1993.

[41] Margaret Burnett and Benjamin Summers. Some real-world uses of visual programming systems. Technical Report TR 94-60-7, Oregon State University, December 31 1994.

[42] Scott Burson, Gordon B. Kotik, and Lawrence Z. Markosian. A program transformation approach to automating software re-engineering. In *14th Annual International Computer Software & Applications Conference (COMPSAC)*, 1990.

[43] Craig Chambers, David Ungar, Bay-Wei Chang, and Urs Hölzle. Parents are shared parts of objects: inheritance and encapsulation in Self. *Lisp And Symbolic Computation*, 4(3), 1991.

[44] Craig Chambers, David Ungar, and Elgin Lee. An efficient implementation of Self, a dynamically-typed object-oriented language based on prototypes. *OOPSLA Proceedings*, 1989.

[45] Bay-Wei Chang and David Ungar. Animation: From cartoons to the user interface. In *Proc. ACM Symposium on User Interface Software and Technology (UIST)*, 1993.

[46] Bay-Wei Chang, David Ungar, and Randall B. Smith. Getting close to objects: Object-focused programming environments. In Margaret M. Burnett, Adele Goldberg, and Ted G. Lewis, editors, *Visual Object-Oriented Programming*. Prentice-Hall, 1995.

[47] Shi-Kuo Chang. Visual languages: A tutorial and survey. In Peter Gorny and Michael J. Tauber, editors, *Visualization in Programming*. 5th Interdisciplinary Workshop in Informatics and Psychology, Springer–Verlag, May 1986.

[48] Colin C. Charlton, Paul H. Leng, and D. W. Wilkinson. Program monitoring and analysis: Software structures and architectural support. *Software—Practice and Experience*, 20(9), September 1990.

[49] Yih-Farn Chen, Michael Y. Nishimoto, and C. V. Ramamoorthy. The C information abstraction system. *IEEE Transactions on Software Engineering*, 16(3), March 1990.

[50] Elliot J. Chikofsky and James H. Cross, Jr. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1), January 1990.

[51] Christian S. Collberg. *Flexible Encapsulation*. PhD thesis, Lund University, December 1992.

[52] James Coplien. Supporting truly object-oriented debugging of C++ programs. In *USENIX Sixth C++ Technical Conference*, April 1994.

[53] Philip T. Cox. Picture the future. *Object Magazine*, July-August 1993.

[54] Philip T. Cox, F. Richard Giles, and Tomasz Pietrzykowski. Prograph. In Margaret M. Burnett, Adele Goldberg, and Ted G. Lewis, editors, *Visual Object-Oriented Programming*. Prentice-Hall, 1995.

[55] Ward Cunningham and Kent Beck. A diagram for object-oriented programs. In *OOPSLA Proceedings*, November 1986.

[56] Ole-Johan Dahl, Edsger W. Dijkstra, and C. A. R. Hoare. *Structured Programming*. Academic Press, 1972.

[57] Ole-Johan Dahl and C. A. R. Hoare. Hierarchical program structures. In Ole-Johan Dahl, Edsger W. Dijkstra, and C. A. R. Hoare, editors, *Structured Programming*. Academic Press, 1972.

[58] C. J. Date. *An Introduction to Database Systems*, volume 1. Addison-Wesley, 1986.

[59] Robert A. Day. *How to write and publish a scientific paper.* Oryx Press, 1994.

[60] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Proceedings of the ACM Conference on Principles of Programming Languages*, January 1984.

[61] Digital Equiptment Corporation. *The VAX Architecture Handbook*, 1981.

[62] Edsger W. Dijkstra. Notes on structured programming. In Ole-Johan Dahl, Edsger W. Dijkstra, and C. A. R. Hoare, editors, *Structured Programming.* Academic Press, 1972.

[63] Christophe Dony, Jacques Malenfant, and Pierre Cointe. Prototype-based languages: From a new taxonomy to constructive proposals and their validation. In *OOPSLA Proceedings*, October 1992.

[64] Nick S. Drew and Robert J. Hendley. Visualising complex interacting systems. In *Human Factors in Computing Systems (ACM CHI Conference Proceedings)*, 1995.

[65] Robert Duisberg. *Constraint-Based Animation: Temporal Constraints in the Animus System.* PhD thesis, University of Washington, 1986.

[66] Robert Duisberg. Visual programming of program visualizations. In *IEEE Workshop on Visual Languages*, 1987.

[67] Mark Edel. The Tinkertoy graphical programming environment. *IEEE Transactions on Software Engineering*, 14(8), August 1988.

[68] Stephen G. Eick and Joseph L. Steffen. Visualizing code profiling line oriented statistics. In *IEEE Visualization*, 1992.

[69] Stephen G. Eick, Joseph L. Steffen, and Eric E. Sumner, Jr. Seesoft — A tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, 18(11), November 1992.

[70] Marc Eisenstadt, Mike Brayshaw, and Jocelyn Paine. *The Transparent Prolog Machine.* Intellect, 1991.

[71] Marc Eisenstadt, Blaine A. Price, and John Domingue. Software visualisation as a pedagogical tool. *Instructional Science*, 21, 1993.

[72] Stephen J. Fenwick. A visual debugger for a object-oriented language. Master's thesis, University of Auckland, 1994.

[73] Stephen J. Fenwick, John G. Hosking, and Warwick B. Mugridge. Cerno-II: A program visualization system. Technical Report 87, Department of Computer Science, University of Auckland, February 1994.

[74] Stephen J. Fenwick, John G. Hosking, and Warwick B. Mugridge. Visual debugging of object oriented systems. In *TOOLS Pacific*, 1994.

[75] Brian Foote and Ralph E. Johnston. Reflective facilities in Smalltalk-80. In *OOPSLA Proceedings*, 1989.

[76] Lindsey Ford and Daniel Tallis. Interacting visual abstractions of programs. In *IEEE Symposium on Visual Languages*, 1993.

[77] Richard Furuta, P. David Stotts, and Jefferson Ogata. Ytracc: a parse browser for yacc grammars. *Software—Practice and Experience*, 21(2), February 1991.

[78] Richard P. Gabriel, Jon L. White, and Daniel G. Bobrow. CLOS: Integrating object-oriented and functional programming. *Communications of the ACM*, 34(9), September 1991.

[79] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns.* Addison-Wesley, 1994.

[80] Emden R. Gansner, Stephen C. North, and Kiem-Phong Vo. DAG — a program that draws directed graphs. *Software—Practice and Experience*, 18(11), November 1988.

[81] Ephraim P. Glinert, editor. *Visual programming environments: applications and issues*. IEEE Computer Society Press, 1990.

[82] Ephraim P. Glinert, editor. *Visual programming environments: paradigms and systems*. IEEE Computer Society Press, 1990.

[83] Ephraim P. Glinert, Mark E. Kopache, and David W. McIntyre. Exploring the general purpose visual alternative. *Journal of Visual Languages and Computing*, 1(1), March 1990.

[84] Adele Goldberg. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley, 1983.

[85] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[86] Eric J. Golin. Tools review: Prograph 2.0. *Journal of Visual Languages and Computing*, 2(2), June 1991.

[87] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: A call graph execution profiler. *ACM SIGPLAN Notices*, 17(6), June 1982.

[88] Judith E. Grass. Object-oriented design archaeology with CIA++. *Computing Systems*, 5(1), Winter 1992.

[89] John C. Grundy and John G. Hosking. The MViews framework for constructing multi-view editing environments. *New Zealand Journal of Computing*, 4(2), 1993.

[90] John C. Grundy and John G. Hosking. Supporting flexible consistency management via discrete change description propagation. *Software—Practice and Experience*, 26(9), September 1996.

[91] John C. Grundy, John G. Hosking, Warwick B. Mugridge, and Stephen J. Fenwick. Connecting the pieces. In Margaret M. Burnett, Adele Goldberg, and Ted G. Lewis, editors, *Visual Object-Oriented Programming*. Prentice-Hall, 1994.

[92] Steven H. Gutfreud. ManiplIcons in ThinkerToy. In *OOPSLA Proceedings*, 1987.

[93] John V. Guttag and James J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10(1), 1978.

[94] Volker Haarslev and Ralf Möller. Visualisation and graphical layout in object-oriented systems. *Journal of Visual Languages and Computing*, 3(1), March 1992.

[95] Wilfred J. Hansen. The 1994 visual languages comparison (panel session). In *IEEE Symposium on Visual Languages*, 1994.

[96] Robert R. Henry, Kenneth M. Whaley, and Bruce Forstall. The University of Washington illustrating compiler. *ACM SIGPLAN Notices*, 25(6), June 1990.

[97] Tyson R. Henry, Scott E. Hudson, Andrey K. Yeatts, Brad A. Myers, and Steven Feiner. A nose gesture interface device: Extending virtual realities. In *Proc. ACM Symposium on User Interface Software and Technology (UIST)*, November 1991.

[98] Daniel D. Hils. Visual languages and computing survey: data flow visual programming languages. *Journal of Visual Languages and Computing*, 3(1), March 1992.

[99] C. A. R. Hoare. Notes on data structuring. In Ole-Johan Dahl, Edsger W. Dijkstra, and C. A. R. Hoare, editors, *Structured Programming*. Academic Press, 1972.

[100] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1, 1972.

[101] John Hogg. Islands: Aliasing protection in object-oriented languages. In *OOPSLA Proceedings*, November 1991.

[102] John Hogg, Doug Lea, Alan Wills, Dennis deChampeaux, and Richard Holt. The Geneva convention on the treatment of object aliasing. *OOPS Messenger*, 3(2), April 1992.

[103] Urs Hölzle, Craig Chambers, and David Ungar. Debugging optimized code with dynamic deoptimization. In *Programming Languages Design and Implementation*, 1992.

[104] John G. Hosking. Visualisation of object oriented program execution. In *IEEE Symposium on Visual Languages*, 1996.

[105] John G. Hosking, Stephen J. Fenwick, Warwick B. Mugridge, , and John. C. Grundy. Cover yourself with skin. In *OzCHI Proceedings*, 1995.

[106] Dan H. H. Ingalls. A simple technique for handling multiple polymorphism. In *OOPSLA Proceedings*, 1986.

[107] Brian Johnson and Ben Shneiderman. Treemaps: a space-filling approach to the visualization of hierarchical information structures. In Ben Shneiderman, editor, *Sparks of Innovation in Human-Computer Interaction*. Ablex, 1993.

[108] Ralph E. Johnson. Documenting frameworks using patterns. In *OOPSLA Proceedings*, October 1992.

[109] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2), 1988.

[110] Tomihisa Kamada and Satoru Kawai. A general framework for visualizing abstract objects and relations. *ACM Transactions on Graphics*, 10(1), January 1991.

[111] Alan C. Kay. The early history of Smalltalk. In R. L. Wexelblat, editor, *History of Programming Languages Conference (HOPL-II)*. ACM, 1993.

[112] Sonya E. Keene. *Object-Oriented Programming in Common Lisp*. Addison-Wesley, 1989.

[113] Pertti Kellomäki. Psd — a portable scheme debugger. *Lisp Pointers*, 6(1), January—March 1993.

[114] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[115] H. H. Kim, Y.-C. Shim, and C. V. Ramamoorthy. APAS: the Ada programming assistant system. In *IEEE Conference on Software Development Environments*, 1988.

[116] Michael F. Kleyn and Paul C. Gingrich. Graphtrace — understanding object-oriented systems using concurrently animated views. In *OOPSLA Proceedings*, November 1988.

[117] Hideki Koike. The role of another spatial dimension in software visualization. *ACM Transactions on Information Systems*, 11(3), 1993.

[118] Wojtek Kozanczynski, Stanley Letovsky, and Jim Q. Ning. A knowledge-based approach to software system understanding. In *6th Annual Knowledge-Based Software Engineering Conference*, 1991.

[119] Wojtek Kozanczynski and Jim Q. Ning. SRE: a knowlege-based environment for large-scale software re-engineering activities. In *IEEE International Conference on Software Engineering (ICSE)*, 1989.

[120] Mukkai Krishnamoorthy and Ramesh Swaminathan. Program tools for algorithm animation. *Software—Practice and Experience*, 19(6), June 1989.

[121] Wilf Lalonde and John Pugh. *Inside Smalltalk*, volume 1. Prentice-Hall, 1990.

[122] Wilf Lalonde and John Pugh. *Inside Smalltalk*, volume 2. Prentice-Hall, 1991.

[123] James A. Landay. Tools review: Serius—a visual programming environment. *Journal of Visual Languages and Computing*, 2(3), September 1991.

[124] Danny B. Lange and Yuichi Nakamura. Interactive visualization of design patterns can help in framework understanding. In *OOPSLA Proceedings*, October 1995.

[125] Henry F. Ledgard and R. W. Taylor. Two views of data abstraction. *CACM*, 20(6), June 1977.

[126] Ted Lehr, Zary Segall, Dalibor F. Vrsalovic, Eddie Caplan, Alan L. Chung, and Charles E. Fineman. Visualizing performance debugging. *IEEE Computer*, 22(10), October 1989.

[127] Henry Lieberman. Using prototypical objects to implement shared behavior in object oriented systems. In *OOPSLA Proceedings*, November 1986.

[128] Henry Lieberman. A three-dimensional representation for program execution. In *IEEE Workshop on Visual Languages*, 1989.

[129] Henry Lieberman, Lynn Andrea Stein, and David Ungar. Treaty of Orlando. In *Addendum to OOPSLA Proceedings*, May 1988.

[130] Panagiotis K. Linos, Phillppe Aubet, Laurent Dumas, Yann Helleboid, Patricia Lejunne, and Philippe Tulula. Visualizing program dependencies: An experimental study. *Software—Practice and Experience*, 24(4), April 1994.

[131] Mark A. Linton. The evolution of dbx. In *Proceedings of the Usenix Summer Conference*, June 1990.

[132] Barbara Liskov and John V. Guttag. *Abstraction and Specification in Program Development*. MIT Press/McGraw-Hill, 1986.

[133] Ralph London and Robert Duisberg. Animating programs using Smalltalk. *IEEE Computer*, 18(8), August 1985.

[134] Pattie Maes. Concepts and experiments in computational reflection. In *OOPSLA Proceedings*, 1987.

[135] Lawrence Markosian, Phillip Newcomb, Russel Brand, Scott Burson, and Ted Kitzmiller. Using an enabling technology to reengineer legacy systems. *Communications of the ACM*, 27(5), May 1994.

[136] Siamak Masnavi. Automatic visualization of the dynamic behaviour of programs by animation of the language interpreter. In *IEEE Workshop on Visual Languages*, 1990.

[137] Satoshi Matsuoka, Shin Takahashi, Tomihisa Kamada, and Akinori Yonezawa. A general framework for bi-directional translation between abstract and pictorial data. *ACM Transactions on Information Systems*, 10(4), October 1992.

[138] Paul L. McCullough. Transparent forwarding: First steps. In *OOPSLA Proceedings*, 1987.

[139] Hans-Peter Messmer. *The Indispensable PC Hardware Book*. Addison-Wesley, 1993.

[140] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, 1988.

[141] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

[142] Bertrand Meyer. *Reusable Software: The Base Object-Oriented Component Libraries*. Prentice Hall, 1994.

[143] Scott Meyers and Steven P. Reiss. An empirical study of multiple-view software development. *ACM Software Engineering Notes*, 15(5), December 1992.

[144] Ken Miyashita, Satoshi Matsuoka, and Shin Takahashi. Declarative programming of graphical interfaces by visual examples. In *Proc. ACM Symposium on User Interface Software and Technology (UIST)*, 1992.

[145] David Moon, Richard Stallman, and Daniel Weinreb. *The Lisp Machine Manual*. M.I.T. A I Laboratory, 1984.

[146] David Morley, Stephen Chiu, Jason Robbins, Tim Maddux, and Geoffrey Voelker. Reusable objects. In *TOOLS Europe*, 1991.

[147] Sougata Mukherjea and John T. Stasko. Applying algorithm animation techniques for program tracing, debugging, and understanding. In *IEEE International Conference on Software Engineering (ICSE)*, May 1993.

[148] Hausi A. Müller, Mehmet A. Orgun, Scott R. Tilley, and James S. Uhl. A reverse engineering approach to subsystem structure identification. *Software Maintenance: Research and Practice*, 5(4), December 1993.

[149] Hausi A. Müller, Scott R. Tilley, Mehmet A. Orgun, Brian D. Corrie, and Nazim H. Madhavji. A reverse engineering environment based on spatial and visual software interconnection models. In *Fifth ACM SIGSOFT Symposium on Software Development Environments*, 1992.

[150] Brad A. Myers. Incense: A system for displaying data structures. In *SIGGRAPH Proceedings*, 1983.

[151] Brad A. Myers. The state of the art in visual programming and program visualization. Technical Report CMU-CS-88-114, Carnegie Mellon University, February 1988.

[152] Brad A. Myers. Taxonomies of visual programming and program visualization. *Journal of Visual Languages and Computing*, 1(1), March 1990.

[153] Brad A. Myers, Ravinder Chandhok, and Atul Sareen. Automatic data visualization for novice Pascal programmers. In *IEEE Workshop on Visual Languages*, October 1988.

[154] Mark A. Najork and Marc H. Brown. A library for visualizing combinatorial data structures. In *IEEE Visualization*, October 1994.

[155] Greg Nelson. *Systems programming with Modula-3*. Prentice-Hall, 1991.

[156] Kathleen Nichols and Paul W. Oman. Navigating complexity to achieve high performance. *IEEE Software*, 8(5), September 1991.

[157] Jun Q. Ning, Andre Engberts, and W. (Voytek) Kozacynski. Automated support for legacy code understanding. *Communications of the ACM*, 37(5), May 1994.

[158] R. James Noble and Lindsay J. Groves. An introduction to the Tarraingím program animation environment. In *TOOLS 6*, 1991.

[159] R. James Noble and Lindsay J. Groves. Tarraingím — a program animation environment. In *The 12th New Zealand Computer Conference*, 1991.

[160] R. James Noble and Lindsay J. Groves. Tarraingím — a program animation environment. *New Zealand Journal of Computing*, 4(1), December 1992.

[161] R. James Noble, Lindsay J. Groves, and Robert L. Biddle. Object oriented program visualisation in Tarraingím. *Australian Computer Journal*, 27(4), November 1995.

[162] Cherri Pancake. Customizable portrayal of program structure. *ACM SIGPLAN Notices*, 28(2), December 1993.

[163] ParcPlace Systems. *ObjectWorks Smalltalk User's Guide*, 4.1 edition, 1992.

[164] Geoffrey A. Pascoe. Encapsulators: A new software paradigm in Smalltalk-80. In *OOPSLA Proceedings*, 1986.

[165] Wim De Pauw, Richard Helm, Doug Kimelman, and John Vlissides. Visualizing the behavior of object-oriented systems. In *OOPSLA Proceedings*, October 1993.

[166] Wim De Pauw, Doug Kimelman, and John Vlissides. Modeling object-oriented program execution. In *ECOOP Proceedings*, 1994.

[167] Donald P. Pazel. DS-Viewer — an interactive graphical data structure presentation facility. *IBM Systems Journal*, 28(2), 1989.

[168] Alan Perlis. Epigrams on programming. *ACM SIGPLAN Notices*, 17(9), September 1982.

[169] Bernhard Plattner and Jurg Nievergelt. Monitoring program execution: A survey. *IEEE Computer*, 16(11), November 1981.

[170] Jörg Poswig, Guido Vrankar, and Claudio Morana. Interactive animation of visual program execution. In *IEEE Symposium on Visual Languages*, 1993.

[171] Jörg Poswig, Guido Vrankar, and Claudio Morara. Visavis: a higher-order functional visual programming environment. *Journal of Visual Languages and Computing*, 5(1), March 1994.

[172] Blaine A. Price and Ronald M. Baecker. The automatic animation of concurrent programs. In *Proc. First Moscow International HCI Workshop*. The International Centre for Scientific and Technical Information, Moscow, USSR., August 1991.

[173] Blaine A. Price, Ronald M. Baecker, and Ian S. Small. A principled taxonomy of software visualization. *Journal of Visual Languages and Computing*, 4(3), September 1993.

[174] Alan Purdy, Bruce Schuchardt, and David Maier. Integrating an object server with other worlds. *ACM Transactions on Information Systems*, 5(1), January 1987.

[175] George Raeder. A survey of current graphical programming techniques. *IEEE Computer*, 18(8), August 1985.

[176] Vaclav Rajlich. VIFOR: a tool for software maintenance. *Software—Practice and Experience*, 20(1), January 1990.

[177] Eric Raymond and Guy L. Steele. *The New Hacker's Dictionary*. MIT Press, second edition, 1993.

[178] Steven P. Reiss. Graphical program development with the PECAN program development system. In *ACM SIGSOFT/SIGPLAN Software Engineering Symposium*, 1984.

[179] Steven P. Reiss. Pecan: Program development systems that support multiple views. *IEEE Transactions in Software Engineering*, 11(3), March 1985.

[180] Steven P. Reiss. Connecting tools using message passing in the Field environment. *IEEE Software*, 4(7), July 1990.

[181] Steven P. Reiss. Interacting with the FIELD environment. *Software—Practice and Experience*, 20(1), June 1990.

[182] Steven P. Reiss. *The FIELD Programming Environment*. Kluwer Academic, 1995.

[183] Steven P. Reiss, Scott Meyers, and Carolyn Duby. Using GELO to visualize software systems. In *Proc. ACM Symposium on User Interface Software and Technology (UIST)*, November 1989.

[184] Steven P. Reiss and Scott Myers. FIELD support for C++. In *USENIX C++ Conference Proceedings*, Spring 1990.

[185] Gruia-Catalin Roman and Kenneth C. Cox. A declarative approach to visualizing concurrent computation. *IEEE Computer*, 22(10), 1989.

[186] Gruia-Catalin Roman and Kenneth C. Cox. A taxonomy of program visualization systems. *IEEE Computer*, 26(12), December 1993.

[187] Gruia-Catalin Roman, Kenneth C. Cox, Donald Wilcox, and Jerome Y. Plun. Pavane: a system for declarative visualization of concurrent computation. *Journal of Visual Languages and Computing*, 3(2), June 1992.

[188] Leo J. Scanlon. *The 68000: Principles and Programming*. Howard W. Sams, 1981.

[189] Robert W. Scheifler and Jim Gettys. The X Window System. *ACM Transactions on Graphics*, 5(2), April 1986.

[190] Kurt Schmucker. MacApp: an application framework. *Byte*, 11(8), 1986.

[191] Kurt Schmucker. Prograph CPX. In Ted Lewis, editor, *Object Oriented Application Frameworks*. Prentice-Hall, 1995.

[192] Robert Sedgewick. *Algorithms*. Addison-Wesley, 1988.

[193] John J. Shilling and John T. Stasko. Using animation to design object-oriented systems. *Journal of Object Oriented Systems*, 1(1), September 1994.

[194] Takao Shimomura and Sadahiro Isoda. Linked-list visualization in debugging. *IEEE Software*, 8(3), May 1991.

[195] Grant Slade and Neville Parker. SEE-Ada: Software evaluation environment for Ada. *Australian Computer Journal*, 26(4), November 1994.

[196] David C. Smith. *Pygmalion — a Computer Program to Model and Stimulate Creative Thought*. Birkhauser, Basel, 1977.

[197] Randall B. Smith, John Maloney, and David Ungar. The Self-4.0 user interface: Manifesting a system-wide vision of concreteness, uniformity, and flexibility. In *OOPSLA Proceedings*, 1995.

[198] Randall B. Smith and David Ungar. Programming as an experience: The inspiration for Self. In *ECOOP Proceedings*, 1995.

[199] John T. Stasko. *Tango: A Framework and System for Algorithm Animation*. PhD thesis, Brown University, May 1989.

[200] John T. Stasko. The path-transition paradigm: a practical methodology for adding animation to program interfaces. *Journal of Visual Languages and Computing*, 1(3), September 1990.

[201] John T. Stasko. A practical animation language for software development. In *IEEE Intl. Conf. on Computer Languages*, 1990.

[202] John T. Stasko. Simplifying algorithm animation with TANGO. In *IEEE Workshop on Visual Languages*, 1990.

[203] John T. Stasko. Using direct manipulation to build algorithm animations by demonstration. In *Human Factors in Computing Systems (ACM CHI Conference Proceedings)*, April 1991.

[204] John T. Stasko. Animating algorithms with XTANGO. *ACM SIGACT News*, 23(2), Spring 1992.

[205] John T. Stasko, William F. Appelbe, and Eileen Kraemer. Utilizing program visualization techniques to aid parallel and distributed program development. Technical Report GIT-GVU-91-08, Georgia Institute of Technology, June 1991.

[206] John T. Stasko, Albert Badre, and Clayton Lewis. Do algorithm animations assist learning? An empirical study and analysis. In *Proceedings of the INTERCHI '93 Conference on Human Factors in Computing Systems, Amsterdam, Netherlands,* April 1993.

[207] John T. Stasko and Charles Patterson. Understanding and characterising software visualisation systems. In *IEEE Workshop on Visual Languages,* 1992.

[208] John T. Stasko and Carlton Reid Turner. Tidy animations of tree algorithms. In *IEEE Workshop on Visual Languages,* September 1992.

[209] Lynn Andrea Stein. Delegation is inheritance. In *OOPSLA Proceedings,* December 1987.

[210] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1986.

[211] Bjarne Stroustrup. Sixteen ways to stack a cat. Technical Report CSTR-161, AT&T Bell Laboratories, October 1991.

[212] Ivan E. Sutherland. Sketchpad: A man-machine graphical communication system. In *Proceedings AFIPS Spring Joint Computer Conference,* volume 23, pages 329–346, Detroit, Michigan, May 1963.

[213] Antero Taivalsaari. *A Critical View of Inheritance and Reusability in Object-oriented Programming.* PhD thesis, University of Jyväskylä, 1993.

[214] Shin Takahashi, Satoshi Matsuoka, Akinori Yonezawa, and Tomihisa Kamada. A general framework for bi-directional translation between abstract and pictorial data. In *Proc. ACM Symposium on User Interface Software and Technology (UIST),* 1991.

[215] The Self Group. *Self Programmer's Reference Manual.* Sun Microsystems and Stanford University, 2.0alpha edition, June 1992.

[216] Gerald Tomas and Chrisoph W. Ueberhuber. *Visualization of Scientific Parallel Programs,* volume 771 of *Lecture Notes in Computer Science.* Springer-Verlag, 1994.

[217] David Ungar, Craig Chambers, Bay-Wei Chang, and Urs Hölzle. Organizing programs without classes. *Lisp And Symbolic Computation,* 4(3), June 1991.

[218] David Ungar and Randall B. Smith. SELF: the Power of Simplicity. *Lisp And Symbolic Computation,* 4(3), June 1991.

[219] Bradley T. Vander Zanden, Brad A. Myers, Dario A. Guise, and Pedro Szekely. The importance of pointer variables in constraint models. In *Proc. ACM Symposium on User Interface Software and Technology (UIST),* November 1991.

[220] Jean-Yves Vion-Dury and Miguel Santana. Virtual images: Interactive visualization of distributed object-oriented systems. In *OOPSLA Proceedings,* October 1994.

[221] John M. Vlissides and Mark A. Linton. Unidraw: A framework for building domain-specific graphical editors. *ACM Transactions on Information Systems,* 8(3), 1990.

[222] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering,* 10(4):352–357, July 1984.

[223] Alan West. Making a case for animating C++ programs. *Dr. Dobb's Journal,* October 1994.

[224] Geoff Williams. The Apple Macintosh computer. *Byte,* 9(2), 1984.

[225] Alan Wills. Capsules and types in Fresco. In *Proceedings ECOOP '91.* Springer-Verlag, July 15-19 1991.

[226] Phil Winterbottom. ACID: a debugger built from a language. In *Winter USENIX,* January 1994.

[227] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object-Oriented Software*. Prentice-Hall, 1990.

[228] Niklaus Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4), April 1971.

[229] Niklaus Wirth. *Algorithms + Data Structures = Programs*. Prentice-Hall, 1976.

[230] Niklaus Wirth. *Programming in Modula-2*. Springer-Verlag, 1983.

[231] Niklaus Wirth. History and goals of Modula-2. *BYTE Magazine*, August 1984.

[232] Kenny Wong, Scott R. Tilley, Hausi A. Müller, and Margaret-Anne D. Storey. Structural redocumentation: A case study. *IEEE Software*, 12(1), January 1995.

[233] Ed Yourdon. RE-3, part 1: Re-engineering, restructuring, and reverse engineering. *American Programmer*, April 1989.

Of making many books there is no end.

Ecclesiastes 12:12