# Visualisations of Execution Traces (VET): An Interactive Plugin-Based Visualisation Tool

**Mike McGavin** **Tim Wright** **Stuart Marshall**

School of Mathematics, Statistics and Computer Science
Victoria University of Wellington,
New Zealand,
Email: {mike.mcgavin,tim,stuart}@mcs.vuw.ac.nz

**Abstract**

An execution trace contains a description of everything that happened during an execution of a program. Execution traces are useful, because they can help software engineers understand code, resulting in a variety of applications such as debugging software, or more effective software reuse. Unfortunately, execution traces are also complex, typically containing hundreds of thousands of events for medium size computer programs, and more for large scale programs. We have developed an execution trace visualisation tool, called VET, that helps programmers manage the complexity of execution traces. VET is also plugin based. Expert users of VET can add new visualisations and new filters, without changing VET's main code base.

*Keywords:* software visualisation, execution traces.

## 1 Introduction

Developers have always been interested in understanding software behaviour. The traditional approach is to read source code or documentation written by the software's author. This approach has several problems: source code is static, complex, and hard to understand; and documentation might be out of date or incomplete. An alternative approach is to capture, log, and visualise runtime information during the software's execution. This approach is now garnering much interest in the research community

Capturing, logging, and visualising run-time information requires significant tool support to automatically extract and present useful information. Any such tool support needs to overcome several hurdles. First, no-one has found a widely-agreed-upon single visualisation template that is suitable for showing all behavioural information. Second, different developers have different information requirements—affecting what a tool needs to display as well as how that information should be displayed. Third, the amount of information extracted from any non-trivial execution of a moderately complex software application is huge, causing information overload.

We believe that a solution to the information overload is to let developers configure their tools. Some existing tools have capability to filter out irrelevant information while the information is being extracted from the runtime environment. However, as developers typically use these tools because they *do not* understand the software, it can be difficult for the developer to identify *what* is useful information before actually seeing a visualisation of the information.

Copyright ©2006, Australian Computer Society, Inc. This paper appeared at the Seventh Australasian User Interface Conference (AUIC2006), Hobart, Australia. Conferences in Research and Practice in Information Technology, Vol. 50. Wayne Piekarski, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

In this paper we present and discuss VET. VET is a software visualisation tool that allows a developer to filter out uninteresting aspects of a software's behaviour while viewing the visualisation. A sample screenshot of VET can be seen in figure 1, and this will be discussed in more depth later on. We used principles outlined by Card, Mackinlay & Shneiderman (1999) as design heuristics:

**Show the user everything up-front.** The system should show the user everything at once when they start the program. This way a user can get an overview of all information and remove information they are not interested in.

**Let the user filter out unwanted information.** In this way, users are aware of what information is not being shown and know the limitations of the current visualisation.

**Show details about a data-point on demand.** Users should be able to easily get detailed information about any particular data point on demand.

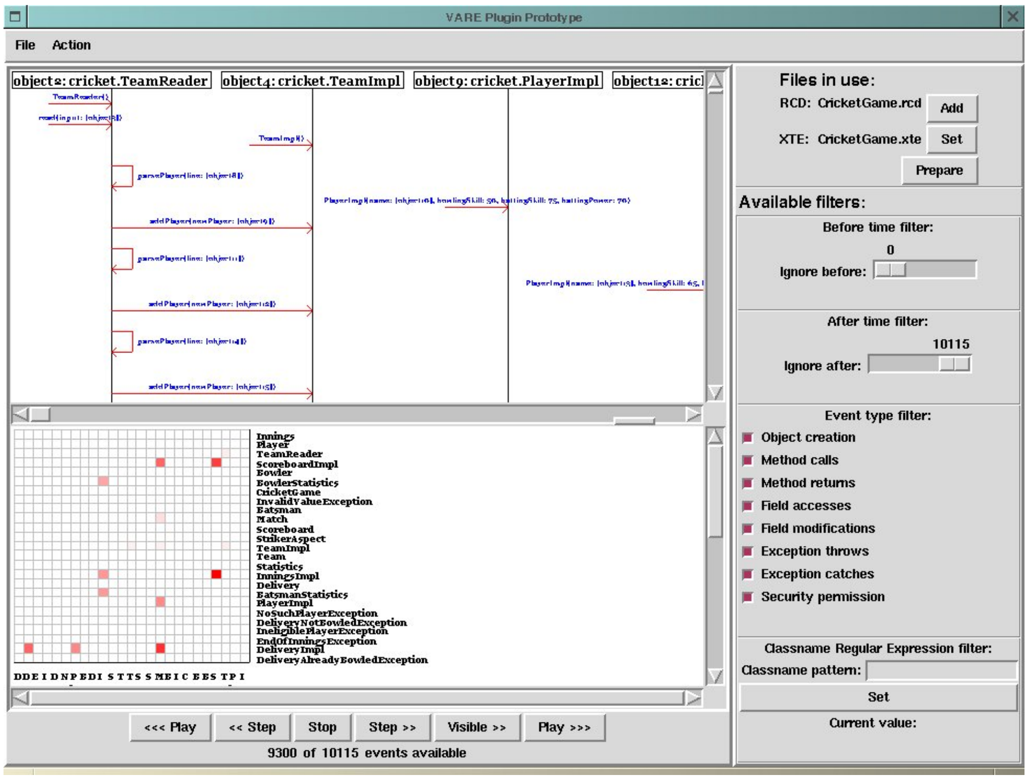
While Card et al.'s (1999) principles are not new, and many researchers have already created software visualisations of software behaviour, we believe that the application of Card et al.'s principles to software visualisation has the potential of developing a viable mechanism to better help developers understand software.

The VET tool is implemented as part of Marshall, Jackson, McGavin, Duignan, Biddle & Tempero's (2001) larger *VARE* architecture for producing software visualisations of reusable software components. However, this approach — as with general software visualisation — is applicable to such areas as teaching computer programming; debugging faulty software visually; and profiling applications.

In this paper we introduce our tool, VET. Section two contains an examination of execution traces and recap their capabilities, limitations, and usage in existing software visualisation research. We then describe a methodology for building visualisations from huge data sets in section 3. Section 4 describes the major contribution of this paper: the VET tool. VET was built by translating the methodology for huge data sets described in section 3 into the domain of understanding software behaviour. Section 5 then discusses future work, before we summarise the direction and findings of this paper in section 6.

## 2 Visualising Execution Traces

Traditionally, if developers have wanted to understand some software's behaviour, they have resorted to using several data sources. These include reading the code, reading any associated documentation written by the software's author, using a debugger to step through an execution, or manually annotating the software with print statements to output state or location at different points during execution. All of these methods have their limitations,



Files in use:
RCD: CricketGame.rcd
XTE: CricketGame.xte

Available filters:

Before time filter:
0
Ignore before:

After time filter:
10115
Ignore after:

Event type filter:
- Object creation
- Method calls
- Method returns
- Field accesses
- Field modifications
- Exception throws
- Exception catches
- Security permission

Classname Regular Expression filter:
Classname pattern:

Current value:

9300 of 10115 events available

Figure 1: A typical view of VET in action. The upper visualisation is a generated Sequence Diagram, and the lower visualisation is a generated Class Association Diagram.

and researchers are now investigating the use of automatically extracted execution traces as a viable data source (Marshall et al. 2001).

In this section we briefly discuss some of the issues with existing approaches to understanding software. We will then introduce the idea of execution traces in more depth, and then briefly discuss existing work in visualising execution traces.

## 2.1 Difficulties with Understanding Behaviour

Code can be difficult to read either because of developers' differing writing styles, or because the code isn't available for legal reasons, or because control flow jumps from one block of code to some other distant block of code, or because (in the case of languages that support polymorphism) it can be difficult to say exactly what one statement will do until the software is actually executing.

Associated documentation may or may not exist, and may not cover the particular facet of the behaviour that the developer is interested in.

Debuggers can be useful, but often require the developer to know where to insert interesting breakpoints prior to understand the software.

Finally, manually inserting `print` statements again requires that code is legally modified, and requires the developer knows where useful data and events are likely to occur.

## 2.2 Execution Traces

Execution traces (or program traces as they are also known) can instead be created by "spying" on the runtime environment in which the software is executing. This approach requires the ability to modify the runtime environment and to access an executable version of the software. The former requirement is fairly trivial for many languages, and especially for those that require virtual machines such as Java and .NET. The latter requirement does have some issues associated, such as securely executing inherently untrusted software, but these issues are being addressed separately (Marshall 2005). In this paper we are interested in execution traces of object-oriented languages, although many of the ideas could be translated to other language paradigms.

Executions traces are a description of events that occur during the execution of a software application. An execution trace typically records such events as object creation, method calling, method returns, field access, field modifications, exception throwing and handling, class loading, and threading.

We have developed our own prototype execution trace format that utilises XML to store execution traces in an architecture- and language-independent manner. This prototype execution trace format is XTE (Marshall, Jackson, Anslow & Biddle 2003). XTE documents information for all the events listed in the preceding paragraph, and also stores the values and references for any objects that are created during the execution of the software. A snippet of XTE can be seen in Figure 2. XTE purely stores behavioural information, and has been designed to work with a second XML format — Reusable Component Descriptions (RCD) — when more extensive structural information is required (Marshall et al. 2003).

As we mentioned earlier, the number of events that can occur during any non-trivial execution is huge. Execution traces in XML can be up to 100MB in size for a typical run of a medium-sized piece of software, and this is obviously outside the ability of any developer to read and comprehend. To compensate for this problem, researchers have developed visualisation tools.

## 2.3 Visualisation Tools

Examples of these visualisation tools are Jive (Reiss 2003) (that visualises Java software as it executes), Bloom (Reiss 2001) (that creates 3D visualisations of Java), and Gammatella (Orso, Jones & Harrold 2003) (that creates visualisations of remote Java software).

One type of visualisation that can already be created using existing tools (such as Bloom) is a class association

```
<objectcreation objectid="object1">
  <complextype>
    <typename>
      test_classes.TestClass1
    </typename>
  </complextype>
</objectcreation>
<methodcall receiverid="object1"
              senderid="object2">
  <methodname>method2</methodname>
  <typename>
    test_classes.TestClass1
  </typename>
  <parameter>
    <objectvalue objectid="object4"/>
    <complextype>
      <typename>
        java.lang.String
      </typename>
    </complextype>
  </parameter>
</methodcall>
```

Figure 2: A sample execution trace taken from the VARE architecture. This execution trace contains two events: an object being created and a method being invoked. Program traces for medium to large scale applications may contain millions of events, making it infeasible to examine them without processing in some way. Do not be concerned if you cannot understand this trace. Neither can we: that's the point.

diagram (otherwise known as a call graph). Class association diagrams show an overview of how often one class calls methods in another class. These diagrams contain a scatter-graph with all objects listed on both the $X$ and $Y$ axis. The point $(x, y)$ is colour coded to represent how often object $x$ has invoked a method on object $y$. A sample class association diagram, produced by VET, is shown in Figure 4.

While class association diagrams provide a great overview of which classes are affiliated, they do not provide any detail about the exact interactions. For example, they do not clearly express which methods are being called or in which time-frame most of the method calls occur. Visualisation tools also can utilise diagrammatic notations that are already familiar to developers, such as the Unified Modelling Language (UML) used for documenting analysis and design documents during software development. One example of a useful behavioural diagram in UML is the sequence diagram.

Sequence diagrams show a visualisation of method calls between objects. Typically, a sequence diagram displays a horizontal list of objects, and a vertical time line. Method calls are represented on the timeline as an arrow from the object that makes the method call to the object that receives the method call. A sample sequence diagram, produced by VET, is shown in Figure 3.

While sequence diagrams are useful for understanding what has happened when software executes, they quickly become difficult to view as the execution traces become large. In an execution trace with millions of events, a user must either scroll far up and down an execution trace to find the information they want or they must zoom out to such a level that the execution trace is unintelligible.

### 2.4 Filtering Execution Traces

Execution traces can be extremely large, and a developer may only be interested in one particular facet of an execution. For example, the developer may only be interested in how one component was used in a larger application, or they may only be interested in the object instantiations during execution.

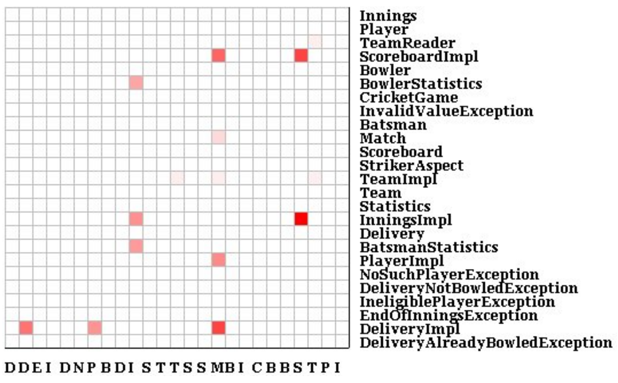


Figure 4: An example of a class association diagram in VET. The darkness of the shade in each part of the grid indicates the amount of messages that have been passed from the class on the $x$ axis to the class on the $y$ axis. These diagrams provide a useful overview of how objects interact during the execution of an application but do not let software engineers examine a particular time-period in the application's runtime.

Irrelevant information can be filtered out during the extraction phase or during the visualisation phase. The former approach is problematic since it requires the developer to make an informed filtering decision with incomplete knowledge. The latter approach is the approach that we utilise in this paper. Few existing visualisation systems support easy filtering of execution trace data. One of the few to support this approach is Bloom (Reiss 2001). Bloom includes a restriction control to filter information in numeric and string fields. Jive can also filter information based on class selection, but does not seem to support other forms of grouping or filtering.

We will expand on this approach by developing a generic filtering framework that opens up the possibility for new types of filters, and by applying Shneiderman's principles to how the developer would then configure and interact with these filters.

### 2.5 Plugging in Visualisations

Several visualisation tools have already been developed that support the plugging in of new visualisations. Some tools, such as Bloom, support the specification of new visualisation strategies via a visualisation backend. We will now briefly discuss a few tools that are representative of the wider collective.

Lintern et al. (Lintern, Michaud, Storey & Wu 2003) discuss their experiences in integrating their software visualisation tool — SHriMP — into the Eclipse IDE. While Eclipse is not solely a visualisation tool, Lintern et al. demonstrate how new visualisations can be added to a framework that already supports users in other development activities. SHriMP supports the navigation through — and the understanding of — static source code, combined with the views already supported by Eclipse. Lintern et al.'s experiences show how other visualisation tools can also be plugged into Eclipse, using the Eclipse framework's plug-in capability.

Wang et al. (Wang, Wang, Brown, Driesen, Dufour, Hendren & Verbrugge 2003) present the EVolve software visualisation framework. EVolve (like VET) has been designed to visualise the dynamic execution of software. EVolve supports the addition of new types of visualisations, and supports linking these new visualisation types to different types of underlying data.

When a new visualisation type is added to EVolve, the submitter must specify dimensions for the new visualisation. Each dimension is tied to a data property in a data source, and the visualisation code can constrain the types of data properties that are valid for any given dimension. VET and EVolve are targetted at the same basic problem

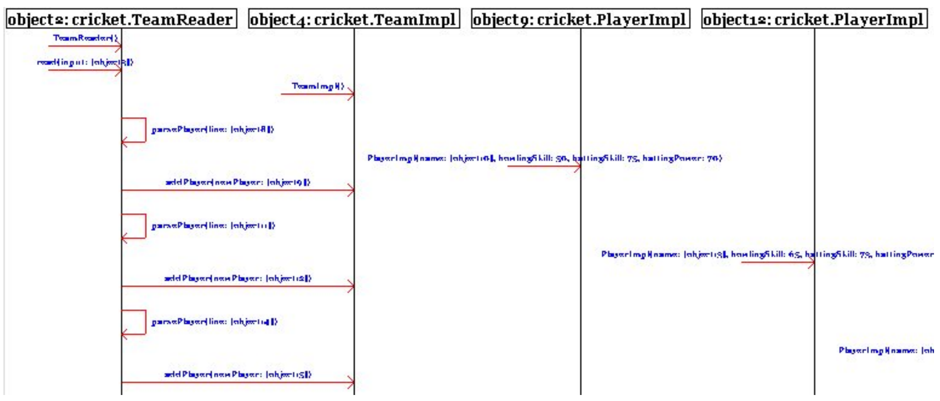

object2: cricket.TeamReader
object4: cricket.TeamImpl
object9: cricket.PlayerImpl
object12: cricket.PlayerImpl
TeamReader()
read(input: (object3))
TeamImpl()
parsePlayer(line: (object8))
PlayerImpl(name: (object10), bowlingSkill: 50, battingSkill: 75, battingPower: 70)
addPlayer(newPlayer: (object9))
parsePlayer(line: (object11))
PlayerImpl(name: (object13), bowlingSkill: 65, battingSkill: 75, battingPower
addPlayer(newPlayer: (object12))
parsePlayer(line: (object14))
PlayerImpl(name: (ob
addPlayer(newPlayer: (object15))

Figure 3: An example of a sequence diagram visualisation for a tiny program consisting of seven events. While sequence diagrams are useful, they become unwieldy when the execution trace contains many thousands of events.

(i.e. flexible visualisation of execution traces). However VET uses Card et al.'s principles to allow the user to dynamically identify what is useful information to be extracted from the execution traces.

## 3 Visualising and Browsing Huge Data Sets

We identified two problems with existing visualisations of execution traces. First, users can not easily browse and navigate to the data they are interested in. Second, users can not filter out data that they are not interested in — users must view all data. Researchers in the information visualisation community have done much work supporting this task. Unfortunately, they have not applied this research to execution traces.

One approach taken by the information visualisation community is to show all information, then let users dynamically filter out unwanted information to drill down to data items where they can get details on demand (Card et al. 1999). The filtering and recreation of the visualisation is done in real-time. In this way, users got an overview of everything and could specify real-time dynamic queries to drill down. The information visualisation community has used this approach to to visualise huge amounts of structured or semi-structured data; an area clearly similar to execution traces.

To better explain this approach, this section introduces and describes two applications. The first, developed by Bederson (2001), is Photomesa. It lets users browse and find photos in large photo collections. The second, developed by Zhao, Smith, Norman, Plaisant & Shneiderman (2004), is Dynamaps. It lets people browse and understand the U.S. Census data. In both of these domains, the data is structured or semi-structured, and the interaction with the user follows the same pattern: users are shown all the data and then the users filter out unwanted data to focus on the information the user wants.

**PhotoMesa** is an application that lets people visualise large sets of photos (Bederson (2001), Figure 5). Photomesa uses the directory structure the photos are stored in to structure the photos. When Photomesa is started, it loads all photos in a user-specified directory (and recursively scans all subdirectories of that directory for photos as well). This levers the structure implicit in directories of photos. After all the photos are loaded they are displayed, grouped hierarchically by the directory or subdirectory in which the photos are stored. Users can click on a directory (or a group of photos) and Photomesa will zoom that group of photos to fill the screen. Users can then scroll left or right, zoom in further, or zoom out using simple mouse-based interactions. At any stage, users can pause the mouse over a photo and a larger version will appear as a tooltip (see Figure 5).

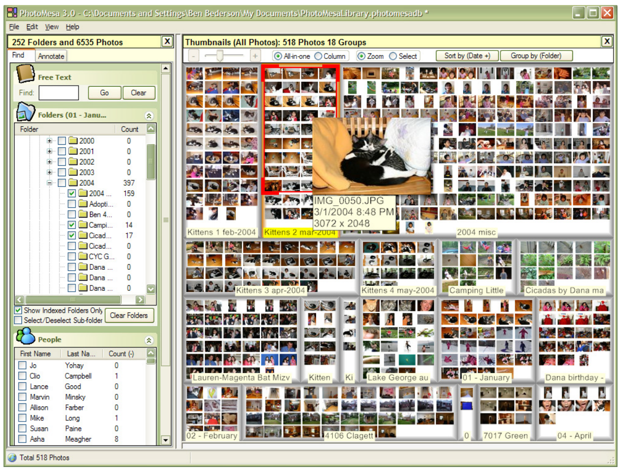

PhotoMesa 3.0 - C:\Documents and Settings\Ben Bederson\My Documents\PhotoMesaLibrary.photomesadb *
File Edit View Help
252 Folders and 6535 Photos
Thumbnails (All Photos): 518 Photos 18 Groups
Find
Annotate
Free Text
Go
Clear
Folders (01 - Janu...
Sort by (Date +)
Group by (Folder)
IMG_0050.JPG
3/1/2004 8:48 PM
3072 x 2048
Kittens 1 feb-2004
Kittens 2 mar-2004
2004 misc
Show Indexed Folders Only
Select/Deselect Sub-folder
Clear Folders
People
Total 518 Photos

Figure 5: PhotoMesa was designed to let people view and navigate around a huge database of photos. When PhotoMesa is started it shows all photos. Users then select a subset of the photos. At all times, PhotoMesa can show details of a particular photo—is it is doing in the screen snapshot above. Image used with permission from Windsor Interfaces, Inc. — www.photomesa.com.

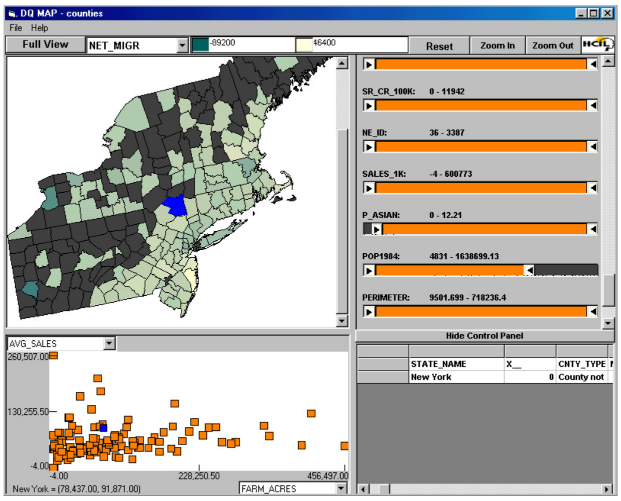


Figure 6: Dynamaps: a user interface for the U.S. Bureau of Census Online Survey Interfaces and Data Visualisation. Users can select what data they want using the sliders on the right of the window and the results of their queries are immediately shown on the visualisations on the left. The redisplay is done in real time as the user moves the slider. Used with permission, Human-Computer Interaction lab, University of Maryland

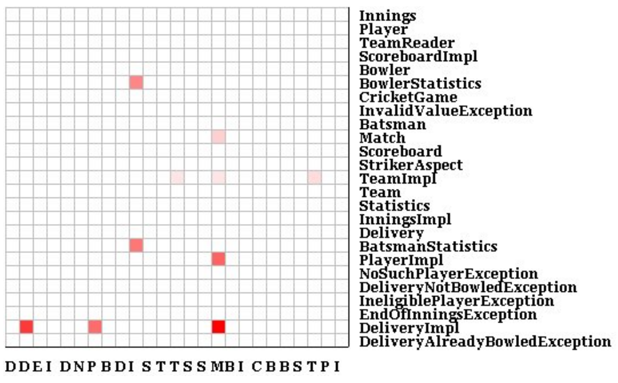


Figure 7: A class association diagram for the same execution trace as in Figure 4, but with events related to several classes having been filtered out.

**Dynamaps** is an application designed to help employees of the US Census Bureau analyse and visualise data from the U.S. census (Zhao et al. (2004), Figure 6). As there is much structure present in the data, Dynamaps provides multiple visualisations of the data. Users select only the visualisations that they want to see, showing only the fields they are interested in examining. Users filter out unwanted data by using sliders on the right hand side of the screen - users can select maximum and minimum values of any field in the data, ranging from standard fields including age or income to esoteric fields like percentage of people owning mobile homes. As users change a filter Dynamaps updates all visualisations in real time.

Dynamaps differs from PhotoMesa in a couple of ways. First, PhotoMesa combines the visualisation and filtering mechanisms. PhotoMesa can combine the two mechanisms because there is only one of each. Second, Dynamaps lets users customise an arrangement of visualisations for their task.

The general approach used by PhotoMesa and Dynamaps is to: show everything; let users filter based on structure in real-time; provide details about a data-point on demand. While we have used two sample applications to show this approach, interested readers should read Card et al. (1999) to get more details.

We believe this approach is applicable to execution traces. Like the census data, execution traces have much structure and there are many ways to view the structure. There is also much varied information in execution traces that we can use to design different filters. The next section describes our tool, VET, which applies this methodology to execution traces. An interesting feature of VET is that it has a plugin-based architecture—users can build their own filters and visualisations.

## 4 VET

VET (Visualisation of Execution Traces) is a prototype visualisation application that lets users interact with and understand execution traces. It is designed to visualise the information stored in an execution trace, which is retrieved from other parts of the VARE architecture. VET allows the user to manipulate multiple views of the execution trace, so that they control how (and what) they learn about the software that the execution trace documents.

### 4.1 Special features of VET

VET uses the methodology we described in the previous section: show everything; let users filter out unwanted data; provide details on demand. In accordance with this methodology, VET displays large amounts of information, and then provides filters to let the user hide what isn't wanted. By manipulating the parameters of the filters, users can directly adjust the information being displayed by the visualisations.

A novel feature of VET is its plugin architecture: VET can be highly customised by expert users. Well defined APIs provided by VET allow expert users to:

1. Design their own visualisation plugins, so that a user can decide what to draw when particular events occur.

2. Design their own filter plugins, so that a user can decide which events to display and which to filter.

**Separation of execution trace from display** VET parses an execution trace and converts it to an event-driven interface for visualisation and filter plugins. It makes the events of an execution trace available through an abstract API. An expert user of VET can define their own visualisations that draw pictures using a provided API, and their own filters. At appropriate times, VET will send messages to the user-designed visualisation components, instructing them to update their displays. This means that expert users who develop visualisations and filters can primarily focus on the task of visualising or filtering, without needing to be as concerned about the functional aspects of parsing and interpreting the execution trace itself.

**Customisable visualisations** VET lets expert users develop visualisation plugins to define visualisations that are displayed for a particular execution trace. These visualisation plugins can later be used by non-expert users. VET provides an abstract programming interface to instruct visualisation plugins of which events to draw, and relevant details. It also provides an abstract interface for drawing pictures, and associated text.

**Customisable criteria for event filtering** VET lets expert users develop filter plugins to define criteria by which events are displayed. It also allows for non-expert users to alter parameters of these filters before, during and after the processing of an execution trace. Visualisations are updated in real time as filtering criteria are adjusted. For example, an expert user might write a time-based filter to block all events that occur before or after certain points in the program. The actual parameters of that filter, however, could be manipulated by a regular user, using a slider widget (such as those shown in Figure 9), before or during the visualisation.

**Synchronisation of visualisations** VET can display multiple visualisations in parallel as an execution trace is processed. At any given time, all visualisations will be synchronised to display different views of the *same* categories of information. The available information depends on the current filter settings.

VET processes an execution trace on an event-by-event basis. For every event, each active filter is queried to determine if the event should be displayed by visualisations. After checking the event, VET passes the event to every visualisation plugin. Visualisation plugins are given the event *even* if the filters do not allow the event, but the visualisation plugins are also instructed not to display the event. Visualisation plugins are still notified of events that are filtered out because some types of visualisations may still require information about hidden events to draw visible events more effectively. For example, when a *Method Call* event is processed, visualisation plugins might want to be aware of the associated *Method Return* event, even if the Method Return event is not displayed. If a user changes the value of any parameters of active filters during or after the execution trace's execution, VET creates a list of all events that have been affected by the change and sends this list to each of the visualisation plugins with requests that the visualisation plugins redraw their display based on the change.

## 4.2 VET in action

This section describes how VET works from a user perspective, including brief descriptions of several example plugins we have developed. We include the main notes about its user interface with summarised details about how the parts interact, descriptions of the visualisation and filter plugins we have developed so far, and a brief explanation of the technologies on which it is built.

### 4.2.1 The user interface

Figure 1 shows VET during the visualisation of a simple execution trace.

**Visualisations** On the left, two visualisations are being displayed. The top visualisation is a sequence diagram and the bottom visualisation is a class association diagram.

Through its internal mechanism of passing instructions about what to display to the Visualisation plugins, VET keeps the two visualisations synchronised at the same location within the execution trace, and also in the information that they are instructed to display.

**Filters** Various filters and their user-modifiable parameters are displayed in the right margin. Figure 10 contains a larger view of the interface for event type filters. The user has set one of the filters to *hide* method return events, VET has instructed both Visualisation plugins to hide them from view. Each visualisation has done so according to its own rules.

**Playback controls** Along the bottom of the program's window are several buttons to allow the user to control the state of the execution trace. The user can "play" the execution trace in forward or reverse, or step through it by one event at a time, to simulate being at different points in the execution process. In Figure 1, the visualisations are showing what has happened up to event 45 out of 74 in total. The button marked "Visible" allows the user to jump forwards to the next event that is *not* hidden from view by any filter.

We have found similar controls to be useful in past visualisation tools that we have developed. They allow the user to move through the execution trace as if the program is actually running and watch events occur in a sequential order. As we will explain in our ideas for future work, however (in section 5), we are considering removing these controls in future versions and replacing them with for more powerful filters.

### 4.2.2 A sequence diagram plugin

In Figure 3, we demonstrate VET's Sequence Diagram visualisation plugin. We have also implemented a feature so that a user can hover the mouse pointer over a class box, and view a popup list of methods within that class. This feature is related to the "details on demand" part of the visualisation methodology we used (see section 1 and the book by Card et al. (1999)).

Although the entire sequence diagram does not fit within the available window in this case, and therefore has scroll bars, the VET API for visualisations provides a "virtual canvas" interface. This means that a visualisation plugin can "draw" visual items on as much of the canvas as is convenient, and VET will optionally scale the graphic automatically, to allow the entire visualisation to be seen. In Figure 8, we show how the same visualisation appears with VET's dynamic scaling in use. Text in the latter diagram is automatically shown as rectangles when it is reduced below a size beyond human readability. We did this to make it more visible, and also because we found that the Tk graphical toolkit was uncomfortably slow at rendering small fonts.

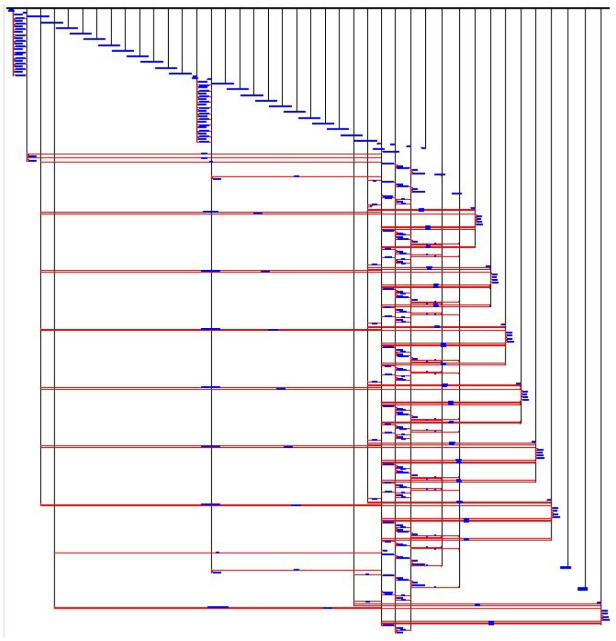


Figure 8: When visualisation plugins make use of VET's dynamic scaling, text may become too small to read. If this occurs, it is displayed as rectangles to reduce the time required for rendering text.

### 4.2.3 A class association diagram plugin

Figure 4 shows a Class Association Diagram visualisation plugin. This visualisation lays classes out in a grid where the frequency of method call events from any class to another class is represented by the darkness in the shade of each square relative to other squares in the grid.

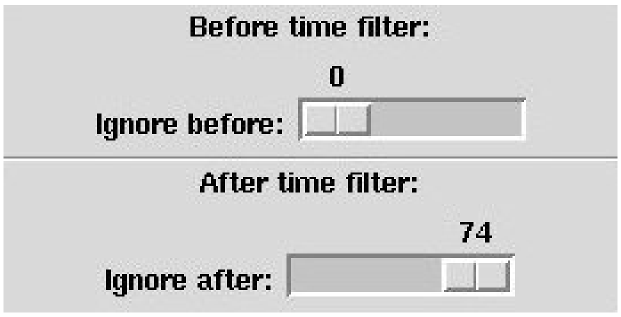
Using the filter plugins with VET, a user can manipulate what is displayed. For example, if the user is primarily interested in activity that relates to certain classes, a *Class name regular expression* filter can be adjusted. In Figure 7, events that relate to the most popular classes of Figure 4 have been filtered. By removing the extreme cases, the shades for other class relationships are separated further, and it becomes easier to discern the difference.

### 4.2.4 Example filter plugins

VET contains several sample *filter* plugins.

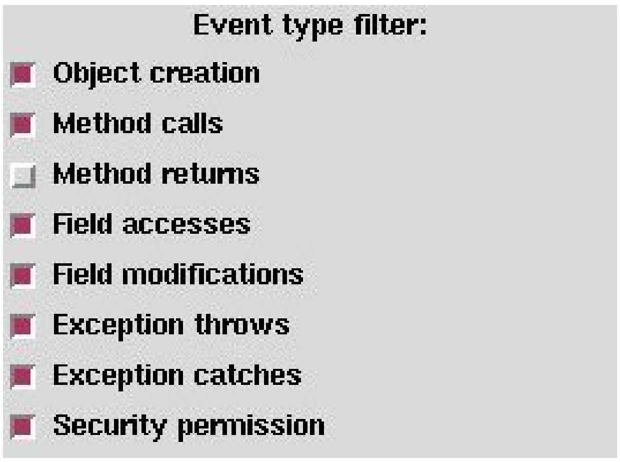
**Event sequence time filters (Figure 9)** These two filters are simple time-based filters (see Figure 9). They allow a user to block all events before or after certain points in time of the execution trace. The visible effect of these filters is similar to the advancing or reversing of the execution trace by using the *step* and *play* buttons. We have implemented them regardless, due to their simplicity and as a proof of concept.

**Event type filter (Figure 10)** The next filter lets a user block events based on their event type. For example, a developer who uses VET might find that there are so many *method call* events being displayed that it is difficult to identify the creation of objects. By using this filter, the developer could quickly remove the display of all *method call* events within visible visualisations.



Before time filter:
0
Ignore before:
After time filter:
74
Ignore after:

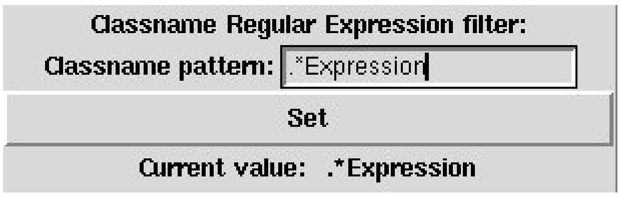
Figure 9: Before Time and After Time filters.



Event type filter:
- 🟥 Object creation
- 🟥 Method calls
- ☐ Method returns
- 🟥 Field accesses
- 🟥 Field modifications
- 🟥 Exception throws
- 🟥 Exception catches
- 🟥 Security permission

Figure 10: An Event Type filter.

**Class name filter (Figure 11)** The final filter that we have designed allows a VET user to filter out events associated with certain classes, by specifying a Regular Expression that identifies the classes. For example, a developer could be particularly interested in the workings of a particular class, and then filter out all other events.



Classname Regular Expression filter:
Classname pattern: .*Expression
Set
Current value: .*Expression

Figure 11: A ClassName filter. This filter is set so that the only events displayed in visualisations will be those associated with classes whose names fit the regular expression.

## 4.3 Implementation notes

Figure 12 shows a simplified diagram of information flows between the plugin architecture of filters and visualisations within VET. Sequential display of an execution trace is controlled by the *Driver Interface* module, which implements the "run", "step", and related commands. For each step, the Driver interface instructs an abstract *Visualisation Box* module to have all active visualisations step to a new position in the trace (arrow labelled A). The Visualisation Box module then instructs each active visualisation plugin to display the visualisation for that particular point in the execution trace (arrow labelled B).

During each update step, any visualisation plugin may request information about *any* event in the process from the *Event Cache* module (arrow labelled C), which is an interface for accessing information in the execution trace.

The Event Cache module acquires necessary information about the requested event from the external XML

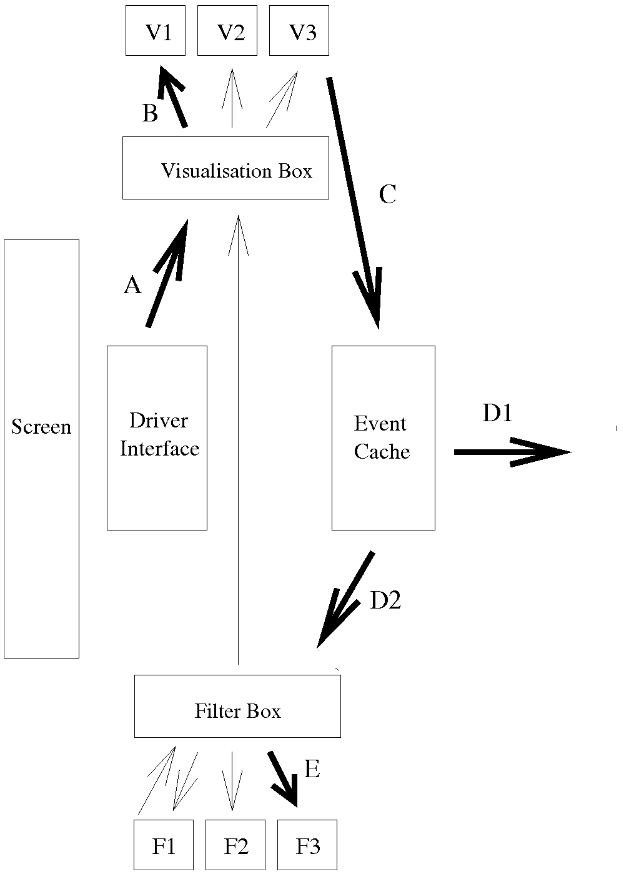


Figure 12: A diagram of the information flow within VET.

source (arrow labelled D1), and wraps it in abstract classes that hide the XML, and provide methods for a visualisation plugin to locate related information. At approximately the same time, the Event Cache module requests information from the abstract *Filter Box* module, about whether the event in question should have a "hidden" status (arrow labelled D2).

The Filter Box module requests the same information from each of the active filter plugins (arrow labelled E). If any filter module states that the event should be hidden, the event is flagged as a hidden event. Otherwise it is flagged as a visible event.

The Event Cache module then returns both the wrapped event, and information about its visible state, to the visualisation plugin that requested it. We have designed the system such that the Event Cache module may cache information about events for a certain amount of time, in anticipation that multiple active visualisations are likely to request information about similar events at similar times.

The green arrows on the diagram indicate alternative flows that take place if the user adjusts the settings of one of the active filters. After an adjustment, the filter plugin notifies the *Filter Box* that a change has been made, and suggests a list of events that may need updating as a result, based on records kept by each filter plugin as the events have first been seen. The Filter Box then notifies the *Visualisation Box*, which instructs each of the active visualisations to update their display of the relevant events. The previously described flows are followed again, with each visualisation plugin requesting relevant information from the *Event Cache* module.

VET is built using the Python scripting language, and the Tk toolkit via Python's Tkinter module. A Python-based API is provided for VET plugins, providing abstract wrappers around all graphical operations and execution trace information. Therefore, an expert user who authors a plugin does not need to interact with Tk widget libraries, or XML parsing libraries. XML data used by VET is currently read into XML DOM objects, which loads an entire execution trace into memory. Although some execution traces would be too large to efficiently fit into memory, we made this design decision to speed up the development of our prototype.

The application has been primarily built and tested in both NetBSD and Linux Operating System environments, however we expect it will port to other operating systems without serious issues, as both Python and Tk are portable.

## 5 Future work

We intend to do a formal usability study of VET. So far we have developed a prototype application that allows for users to filter the information displayed when examining an execution trace visualisation. We would like to find out if potential users find our interface is useful and usable. As we mentioned earlier, we believe that the nature of the filters used in VET might render the "playback" interface obsolete. This is a particular idea to investigate in a usability study.

Another improvement would be to improve the directness of the user's interaction even further. VET does not let a user directly interact with the visualisations themselves: users must manipulate filters to show and hide information. A future application could let the user perform more direct actions like interacting with particular events or objects in one visualisation and having the related part in other visualisations be highlighted in some way.

A further interesting development would be to consider removal of the playback controls. Although we have found similar controls to be useful in past visualisation tools, our personal experience with VET have suggested that the controls are not as meaningful when the filter system is available. This is because the end result of adjusting the "current position" of the execution trace is very similar to the effect of a filter that hides all events that occur after that particular position. For instance, the same effect as shown in Figure 1 could be achieved by adjusting the "after time" filter to hide the results of events that occur after event 45. Presently, we do not have a reliable mechanism for filters to simulate a "play" style of effect, where event would automatically become visible over a passage of time. This is something we could add in future versions of VET, however, and we may therefore experiment with removing the play, step and stop controls altogether.

Finally, we would also like to create more visualisations and more filters for VET as our current library of visualisations and filters is limited. By developing more filters and visualisations there will be a wider range of ideas to experiment with during user testing.

## 6 Conclusion

We have introduced and demonstrated the functionality of VET, a front-end application that we have designed to work as part of the VARE software visualisation architecture. VET is notable because it includes an interactive interface for filtering information. The design is such that all information in an execution trace is displayed, and the user is then given control over how to filter the information. Expert users of VET can design their own visualisations and filtering criteria, via a plugin architecture.

## References

Bederson, B. (2001), 'Photomesa: A zoomable image browser using quantum treemaps and bubblemaps', *CHI Letters* **3**(2), 71–80.

Card, S. K., Mackinlay, J. D. & Shneiderman, B., eds (1999), *Readings in information visualization : using vision to think*, Morgan Kaufmann Publishers.

Lintern, R., Michaud, J., Storey, M.-A. & Wu, X. (2003), Plugging-in visualization: Experiences integrating a visualization tool with eclipse, *in* 'Proceedings of the ACM Symposium on Software Visualization'.

Marshall, S. (2005), Test Driving Reusable Components, PhD thesis, Victoria University of Wellington.

Marshall, S., Jackson, K., Anslow, C. & Biddle, R. (2003), Aspects to visualising reusable components, *in* T. Pattison & B. Thomas, eds, 'Proc. Australian Symposium on Information Visualisation (invis.au 2003)', Vol. 24 of *Conferences in Research and Practice in Information Technology*, Australian Computer Society Inc., pp. 81–88.

Marshall, S., Jackson, K., McGavin, M., Duignan, M., Biddle, R. & Tempero, E. (2001), Visualising reusable software over the web, *in* T. Pattison & P. Eades, eds, 'Proc. Australian Symposium on Information Visualisation (invis.au 2001)', Vol. 9 of *Conferences in Research and Practice in Information Technology*, Australian Computer Society Inc., pp. 103–111.

Orso, A., Jones, J. & Harrold, M. J. (2003), Visualization of program-execution data for deployed software, *in* 'Proceedings of the ACM Symposium on Software Visualization'.

Reiss, S. (2003), Visualising java in action, *in* 'Proceedings of the IEEE Conference on Software Visualisation'.

Reiss, S. P. (2001), An overview of bloom, *in* 'PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering', ACM Press, New York, NY, USA, pp. 2–5.

Wang, Q., Wang, W., Brown, R., Driesen, K., Dufour, B., Hendren, L. & Verbrugge, C. (2003), Evolve: An open extensible software visualization framework, *in* 'Proceedings of the ACM Symposium on Software Visualization'.

Zhao, H., Smith, B. K., Norman, K., Plaisant, C. & Shneiderman, B. (2004), 'Interactive sonification of choropleth maps: Design and evaluation', *IEEE multimedia, Special issue on Interactive Sonification* **12**(2), 26–35.