Dynamic Visualisation of Software State

James Ashford **Neville Churcher** **Warwick Irwin**

Department of Computer Science and Software Engineering
University of Canterbury,
Private Bag 4800, Christchurch 8140, New Zealand,
Email: jra82@uclive.ac.nz, neville.churcher@canterbury.ac.nz
warwick.irwin@canterbury.ac.nz

Abstract

The size and complexity of software systems presents many challenges to developers. Software visualisation techniques can help make more manageable tasks across the development cycle. In this paper we focus on the dynamic visualisation of software state—an important element in supporting activities such as debugging. We propose Pseudo-Breakpoints as a means of collecting data and making it available to specialised visualisation components. We describe the implementation of this concept in Eclipse and present some example visualisations.

Keywords: Software visualisation, information visualisation, debugging.

1 Introduction

Software engineering, described by Parnas as the "multi-person construction of multi-version programs," (Parnas 1975) is a dynamic and challenging discipline.

Despite numerous advances in areas such as programming languages, design techniques, tool support and process models developers have struggled to keep pace with the ever-increasing size and complexity of software systems.

The consequences are reflected in the figures for the number of software projects which either fail or are significantly compromised.

Although software engineering is a collaborative undertaking in which individuals play a variety of rôles, the bulk of the "real" work continues to be carried out by individual developers. These people will typically be using individual tools, with IDEs such as Eclipse (`http://www.eclipse.org`) being the norm, and collaboration being managed at the resource level by version control tools such as subversion (`http://subversion.apache.org`).

Development ultimately consists of individuals pitting themselves against a range of tasks including elements of design, comprehension, implementation, fault diagnosis and repair, and testing.

Many of the techniques used by individual software engineers (such as UML, OO design patterns, code smells, complexity metrics, ...) are essentially static. They are applied in the context of a snapshot of the evolving software artifacts. Such techniques are among the most effective available.

However, there are a number of vital development activities where dynamic techniques have much to offer—either on their own or in combination with static techniques. Examples include:

algorithm visualisation: Understanding and communicating algorithms by enacting them in a way which highlights visually the relevant elements has a long history, dating from systems such as Balsa (Brown & Sedgewick 1984) and Tango (Stasko 1990).

profiling: Implementing a "clean" design and subsequently observing and gathering data about performance issues is likely to direct refactoring effort to productive areas.

debugging: The ability to suspend a running program at "interesting" places and examine its evolving state is a powerful tool for identifying faults.

testing: Intrinsically dynamic activities such as enacting scenarios, use cases or performing acceptance tests occur in software processes.

Information visualisation (Spence 2001, Ware 2004) involves the display of information via computed geometry (a simple example is the representation of a class by a rectangular glyph in UML). Unlike scientific visualisation (electric field strength, temperature profiles, ...), there is no "real" geometry. The computed geometry provides an anchor or metaphor for the user. Software visualisation is the application of information visualisation techniques to problems in the software engineering domain and has been used to address a wide range of issues in software engineering (Eades & Zhang 1996, Stasko et al. 1998).

Effective software visualisation design involves the identification of tasks in the problem domain, the selection of relevant domain data items, and the processing and presentation of derived *information* (Churcher & Irwin 2005). In our previous work, we have applied this approach primarily to visualisations of static software properties (Irwin & Churcher 2003, Churcher et al. 2003, 2007, Irwin et al. 2005, Neate et al. 2006, Harward et al. 2010).

In this paper, we explore the extension of our ideas to the visualisation of dynamic software state. There are two main contributions. Firstly, we propose Pseudo-Breakpoints. The breakpoint is a familiar concept in debuggers: a user-specified point at which execution may be suspended while the user gathers data by exploring the current program state. The Pseudo-Breakpoint is essentially similar, with the

exception that execution need not be interrupted—instead the specified information is included in a range of visualisations. Secondly, we propose a range of visualisations which will assist developers with tasks requiring comprehension of dynamic software state. We have implemented Pseudo-Breakpoints and a number of visualisations in Eclipse.

The remainder of the paper is structured as follows. In the next section, we discuss tasks which developers must perform and their information needs. In section 3 we present the Pseudo-Breakpoint concept and describe its implementation in Eclipse. Section 4 includes a number of visualisations and examples of their application. Finally, we present our conclusions and discuss ongoing work.

2 Tasks and issues

For simplicity, we will use the term "debugging" to refer to a wide range of activities which require a developer to acquire, comprehend, maintain, analyse and react to information about the state and behaviour of a running program.

Debugging is challenging, and remains so despite the advent of modern OO languages (Java, C++, C#, ...) and IDEs (Eclipse, Visual Studio, ...). Multi-threaded software, distributed applications and huge APIs are just three of a range of complicating factors which were not commonly faced by C programmers in the 1980s but which have become major issues today.

Carrying out activities such as those mentioned above requires a developer to cope with potentially very large data sets. There are likely not only to be a large number of data items involved but also to be complex interrelationships. This leads to the problem of information overload—which information visualisation aims to address by employing mechanisms for structuring and exploring related data items.

The tools we use are themselves very complex and have the potential to contribute to the information overload experienced during debugging. Figure 1 shows the debug perspective of Eclipse in use. Like its analogues in other IDEs, the display features numerous components, each with a specific purpose. Some relate to threads, some to the location of breakpoints, some to the state at the current execution point, some to corresponding source code and so on. It is common for holophrasting mechanisms to be employed to allow data structures to be expanded as desired. While this is an effective technique, one can have too much of a good thing—the state of the interface in Figure 1 shows several structures expanded in this way. The user can be given the feeling of "You are in a maze of twisty little passages, all alike" in such cases.

In order to solve a problem or complete a task, the developer needs access to a (potentially very large) set of data. However, at the same time, it is also necessary to have a "holistic" overview of the larger-scale situation. This is known as the *focus+detail* problem in information visualisation (Spence 2001, Ware 2004).

Ways of addressing this problem include distortion-oriented (fisheye view) techniques (Furnas 1986, Sarkar & Brown 1994, Leung & Apperley 1994) in which the degree of interest (DOI) of each data item is determined and used to determine quantities such as the amount of the display or degree of highlighting with which to is presented to the user. We explore the application of this idea in our present work.

The mechanics of debugging have not changed substantially for decades, though the tools themselves are much more sophisticated. Users set breakpoints at "interesting" locations. When program execution is suspended at a breakpoint, information about the program state is gathered: this may involve examining the contents of heap and stack in varying degrees of detail.

Effective debugging involves developers selecting appropriate strategies for isolating problems, identifying their causes and taking corrective action. The process is often initiated by a specific symptom such as a NullPointerException. Isolating this may involve activities such as stepping through nested control structures (in case the cause is an initialisation error), following a call chain or considering concurrent thread issues.

In order to support people engaged in such tasks we need an understanding of the various activities and sequences of steps together with their information needs. To do this exhaustively would require extensive empirical work (such as that of Vans et al (Vans et al. 1999)) and the results would reflect a range of individual styles and available technologies. There is a considerable amount of experience-based advice (e.g. `http://home.earthlink.net/~patricia_shanahan/debug`) about debugging strategies. There are also interesting overlaps with more general areas such as software comprehension as well as more specific ones such as debugging strategies—an example is the work of Lawrance et al. (2007).

However, it is sufficient for our purposes to consider the following (incomplete and unordered) list:

Location: Where am I? This question may be answered in varying degrees of detail—including at this breakpoint, in this scope/block, in this method, at this node, in this collection. Location also has an aspect of "*When* am I?" which might be conveyed by an iteration number or recursion depth.

History: Related questions include "How did I get here?" and "Have I been here before?" Answering these may involve referring to branching (for invocations or jumps) or to iteration controls. A question like "Where have I been?" can be difficult for a conventional debugger to answer fully.

Activity: What's happening? This might involve the most recent change to a variable in scope, relative frequency of changes to individual variables, activity within collections and so on.

Analysis: Where is the time going? What patterns are evident in the updates to this collection? Addressing questions such as these may require multiple information sources and involve exploration of various parts of the software state and history.

Conventional breakpoints answer some of these (such as corresponding source code location) precisely and associated debugger features such as watch lists indicate aspects of activity. Others are much less accessible via conventional displays.

Additionally, using conventional breakpoints can lead to a somewhat "stop-start" mode of operation. The user is required to make significant cognitive jumps as the context changes from one breakpoint to another, or even to another occurrence of the same breakpoint.

Our approach is based on the concept of using visualisation to obtain a more holistic view of the software state and then to support a focus on more specific elements or information. Our system complements the conventional breakpoint approach and can be used alongside it as desired.

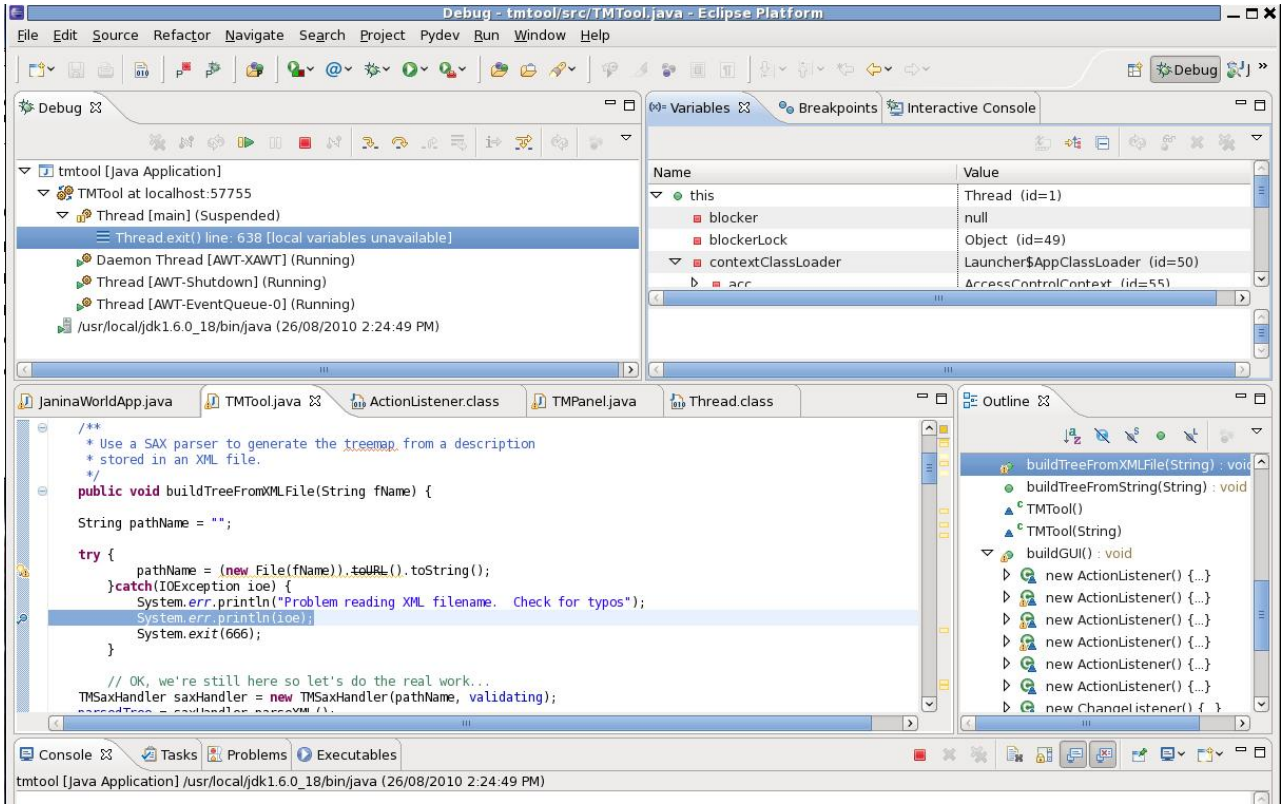


Figure 1: Eclipse debugger perspective

3 PseudoBreakpoints

Much work has been done on support for debugging and other activities requiring access to dynamic software state information and there is a considerable body of research literature on adding visual elements. However, some of these are stand-alone research prototypes, teaching tools or constrained in some way. We have chosen to implement our system as an Eclipse plug-in because we believe it is essential to be able to study the resulting tools in "real" user environments. Similarly, visualisation sometimes appears to be an after-thought—being provided as something to do with the data rather than being an integral element in the overall system design. Our approach involves identifying tasks, together their information needs, and designing visualisations, together with appropriate data capture facilities, to support them.

Mehner (2002) developed a UML-based visualisation similar to the augmented UML class diagrams we have created. It runs as a separate application outside of any IDE and appears to be designed more as a teaching tool than a tools for practitioners. Another stand-alone application sitting outside an IDE is an extensible sequence diagram generator based on execution traces (McGavin et al. 2006).

Lnnberg et al. (2004) developed a prototype debugging tool (MVT) which allows for the visual debugging of software. MVT uses the JDI (Java Debug Interface) — again as a stand-alone application.

Jacobs & Musial (2003) make use of the DOI concept in order to increase significantly the number of classes which can be displayed in their UML-based visual debugging application.

Czyz & Jayaraman (2007) is another system which uses the Eclipse interface to generate UML diagrams within the IDE. While their work involves recording some state information, it does not address the range of aggregation and integrated visualisation techniques that we attempt.

In order to fulfil our visualisation goals we need a means of obtaining relevant data and directing it to appropriate GUI elements. Our initial vision was essentially a "watch list" including the entire system but this would be impracticable on systems of realistic size.

The Pseudo-Breakpoint concept represents what seems to us to be the best of both worlds: it allows specific locations to be selected and de-selected in the same way as conventional breakpoints but does not require execution to be suspended in order to obtain information.

Using Eclipse extensions, we have created a new breakpoint type, PseudoBreakpoint, which extends IBreakpoint in the Eclipse breakpoint model. The standard Eclipse editor has been extended to include a new annotation (the red dot to indicate where a pseudo breakpoint is located) and menu item to allow the user to toggle the new breakpoint on and off. In this way, Pseudo-Breakpoints appear to the user in a natural manner for Eclipse users.

When a new Pseudo-Breakpoint is added, the PseudoBreakpointCreator class determines which breakpoint should be created. Two types, PseudoBreakpointJava and PseudoBreakpointPython, are defined in the example shown in Figure 2. Each class contains language-specific breakpoint code (PseudoBreakpointJava actually extends the normal JavaBreakpoint to provide this functionality).

Data collection occurs when the program is run in debug mode. Our Pseudo-Breakpoints are inserted into the virtual machine in exactly the same way as ordinary breakpoints.

We have created an event handler to capture every Eclipse Debug event — including whenever a break-

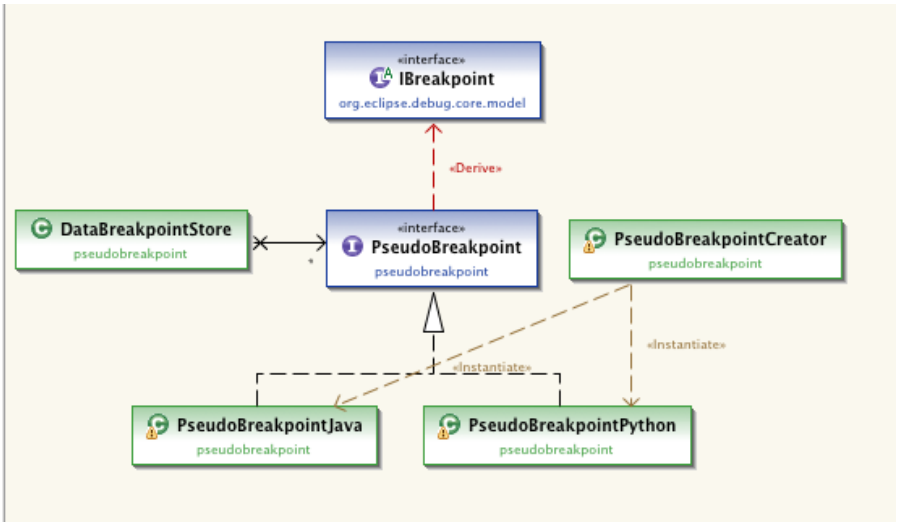


Figure 2: Pseudo-Breakpoint structure

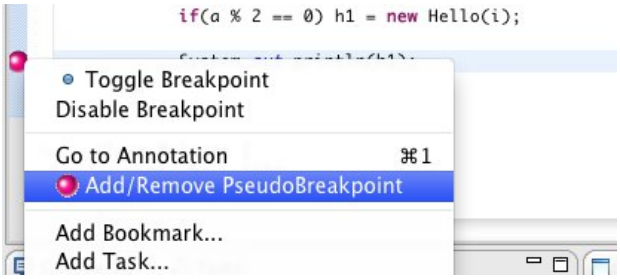


Figure 3: Managing Pseudo-Breakpoints

point is hit (i.e. the corresponding statement is about to be executed). The process is shown schematically in Figure 4. Whenever a breakpoint has been hit, the handler first determines if it was a pseudo breakpoint. If so, then it will walk the object tree, taking a copy of all variables it encounters. It will also record the time the breakpoint was hit and the name of the executing thread. Each variable has a unique object ID allowing individual objects, and the relationships between them, to be identified.

Any necessary comparisons between objects (to determine whether/how the variables have changed) occurs during the generation of the visualisation rather than during the program runtime.

The model design is very simple, and contains two main elements: BreakpointEvent and Variable. Each execution of a program will result in a list of BreakpointEvents (one for each time a PseudoBreakpoint is hit), and each BreakpointEvent will contain a set of Variables — all the variables that were in-scope at the breakpoint event (including local, global, and instance variables). Each variable contains a set of all references it contains (if it is an object), or the value (if it's a primitive). This is done recursively to ensure that all variables and sub-variables are recorded.

Our system is intended to be extensible, so that new visualisations may be added as required. Adding a new visualisation requires development of two main components: a VisualisationDisplayType and a Visualisation (see Figure 5). A VisualisationDisplayType contains all the code to display an Eclipse ViewType. A ViewType is an Eclipse-specific view which is used within the IDE to create a new panel (similar to AWT Panel or Swing JPanel). To create your own custom visualisation, you must extend the VisualisationDis-

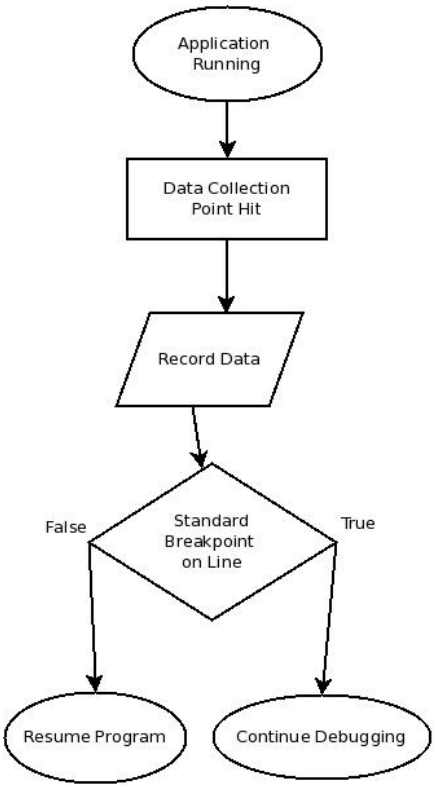
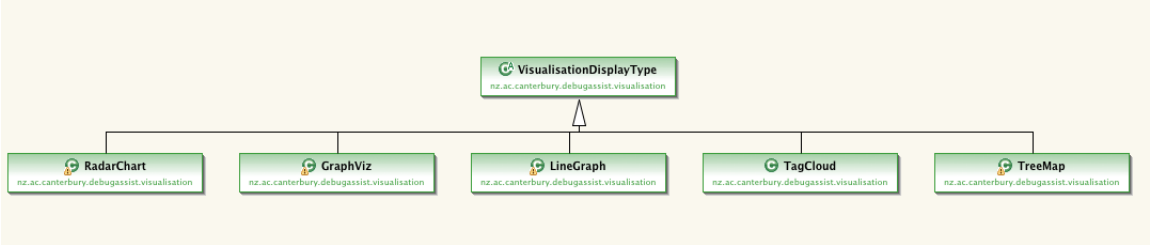
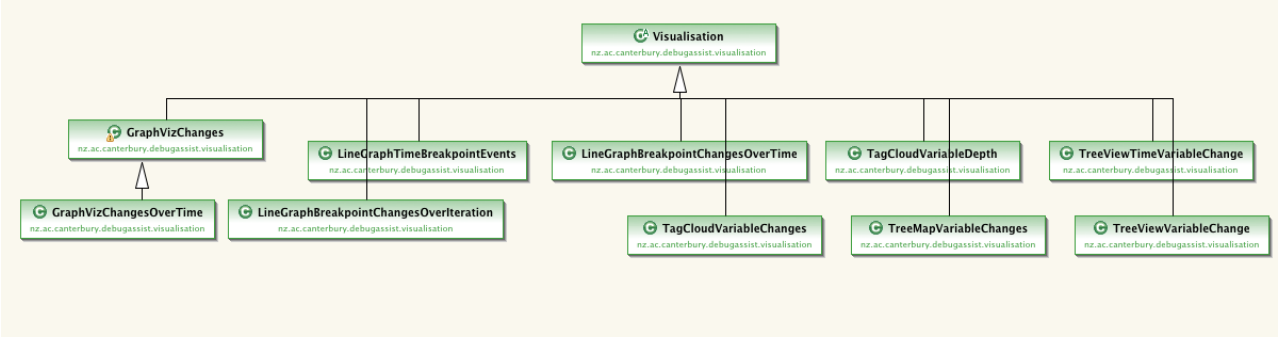


Figure 4: Handling Pseudo-Breakpoint



(a)



(b)

Figure 5: Visualisation structure

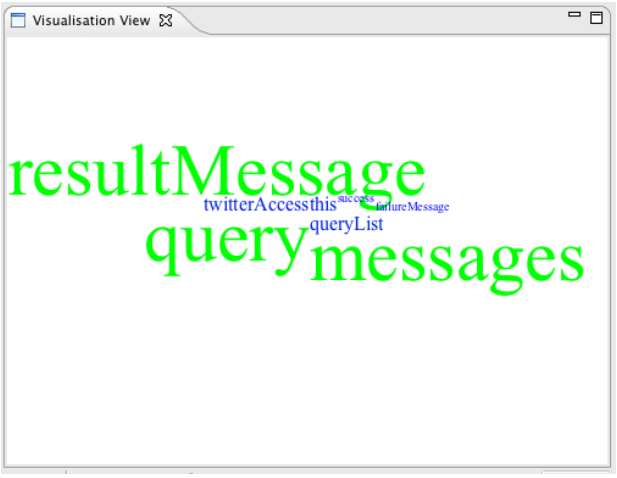


Figure 6: Tag cloud corresponding to state shown in Figure 7

playType class, and implement the createPartControl method to generate the ViewType's content.

A Visualisation class contains the code to process the breakpoint and variable information for a VisualisationDisplayType to use.

This means that a given VisualisationDisplayType can have multiple Visualisations, and multiple Visualisations can use different VisualisationDisplayTypes.

Currently, each Visualisation and VisualisationDisplay must be integrated manually within the plug-in via the VisualisationHandler class. It is intended that future versions will support the use of an XML configuration file to allow new visualisations to be added without the need to manually alter the code.

To date, we have created a number of visualisations using this framework. An example is our LineGraph (see Section 4.6. We created a generic LineGraph DisplayType using JFreeChart (`http://www.jfree.org/jfreechart`) that was able to generate a ViewPoint to display a number of points on a Line Graph. A number of visualisation variants are then able to supply data to be plotted on the LineGraphs (such as number of variable changes per breakpoint and time between breakpoint hits). The ability to make use of existing visualisation tools such as JFreeChart, while supporting the development of specialised extensions, makes it relatively straightforward to generate simple, yet useful, visualisations.

Performance has not been the primary focus of our work to date but our experience thus far has been encouraging. Any debugging or monitoring environment will inevitably experience some overheads when compared to the production environment.

In our case, there are two potential sources. Firstly, the number of Pseudo-Breakpoints, the number of times each is activated, and the number of variables to be monitored will contribute to the data collection costs. The cost of processing the data for visualisation is the other potential source of performance issues. We have not found either to be problematic. We ran our system on a typical Java project with 200 Pseudo-Breakpoint activations we found $\mu = 125ms, \sigma = 26ms$ for the time cost of data collection — well within acceptable limits,

This cost depends on the number and complexity of in-scope variables, since these are copied when a Pseudo-Breakpoint is hit, so having sufficient client memory available will be important as scope size and activation count increase.

Ongoing and future work includes improving the overhead required with the data collection process and the potential for serialising large data sets for subsequent analysis and replay.

One way to manage the volume of information is to filter out "uninteresting" data. Basic filtering has been implemented. Users can remove primitives from data collections and limit the variable depth. String objects are converted to a primitive type (rather than having all their internals shown) to save space on any visualisations.

Future work would include more extensive and finely-controllable (class and variable based) filtering to help suppress irrelevant information and decrease

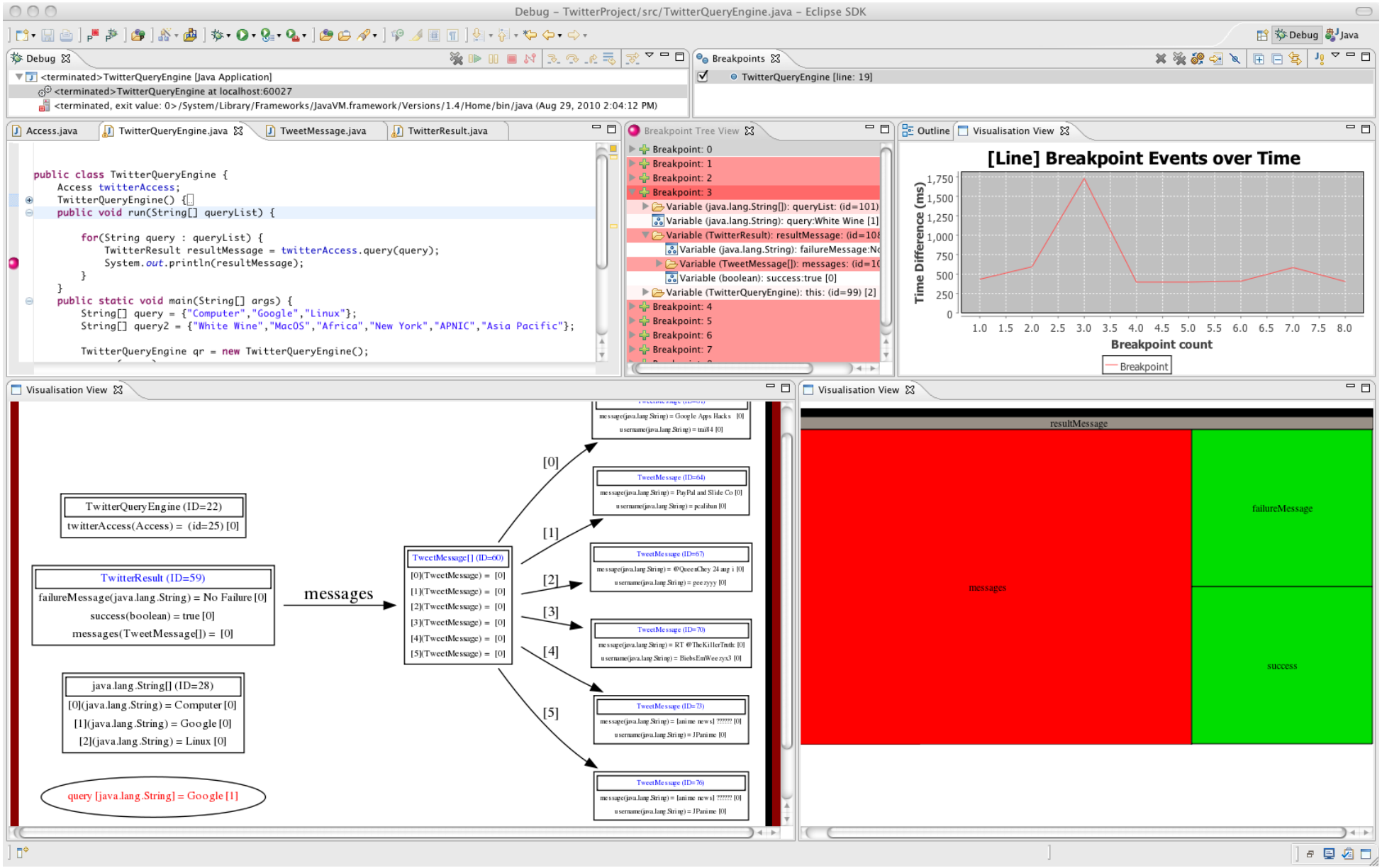


Figure 7: Eclipse debug perspective with visualisations

memory usage.

4 Visualisations

In this section we present a range of visualisations which illustrate the application of our approach. Having considered the range of tasks to be supported, we have identified a number of common themes and designed individual visualisations around them. Examples include drilling down through hierarchical structures to reveal more detail and determining the differences between the state at successive breakpoint visits.

Figure 7 shows Eclipse in action with a selection of our visualisations in use. Note the significant contrast with Figure 1. Figures 1 and 7 represent two extremes and the user is free to customise the interface as her information needs evolve.

4.1 Tag clouds for context

Tag clouds have become commonplace on blogs and in information retrieval contexts where the holistic contextual overview of the content of a document or document collection is of interest. A recent example is the analysis of Barack Obama's inauguration speech (`http://www.telegraph.co.uk/news/worldnews/northamerica/usa/barackobama/4299886/Barack-Obamas-inauguration-speech-as-a-tag-cloud.html`) In their simplest form, tag clouds consist of a number of words which are terms in a document. The font size used to display each word indicates the frequency with which it occurs in the document. We have extended the concept for use in software visualisation (Deaker et al. 2010). In our approach, visual attributes such as the size, font family, colour and transparency of individual words may each be used to display a variable of interest.

This approach has potential benefits in comprehension of the evolving state of a running program. The tag text is directly mapped to identifiers in the program so no separate translation is required. Properties such as size and colour can be mapped to quantities such as number of invocations or time in scope.

Figure 6 shows an example: it corresponds to the state shown in Figure 7 and indicated by a number of other visualisations there. The font size for each identifier indicates the number of assignments to the corresponding variable. This allows the user to identify rapidly the most active data items which can then be scrutinised more closely.

4.2 Treemaps for hierarchy

Hierarchical structures are common in software engineering. Static examples in Java include the inheritance relationships of classes and the nesting of packages.

In the dynamic context, activities such as following nested scopes, call graphs or drilling into data structures involve navigation and comprehension of hierarchical structures.

In practice, "nearly hierarchical" structures are also common: adding interface implementation to pure inheritance breaks the strict hierarchy and real call graphs aren't necessarily trees. However, representing such situations with visualisations designed primarily for hierarchical contexts is often satisfactory.

Displaying large trees can be awkward. One appealing solution is treemaps, a space-saving representation which can fit neatly within the constraints of a pane in a GUI component (Johnson & Shneiderman 1991).

We have made use of treemaps in previous work (Churcher et al. 1999, Irwin & Churcher 2002) and they are widely-used in information visualisation applications.

A treemap is visible at the lower right of Figure 7. The rectangular nodes correspond to data items in the scope. The rectangle size indicates the number of children: in this case the larger node (messages) at left is an array while the other two are primitive types. The colour of the rectangle is currently configured to indicate the number of updates which have occurred to the children. Expanding and collapsing the nodes allows the user to explore potentially complex structures without encroaching onto the screen real estate for other concurrent visualisations.

4.3 Augmentation

A common strategy is to implement individual visualisations separately: each has its own primary purpose and is located in a well-defined dedicated part of the IDEs interface. Rather than providing a separate visualisation is it sometimes helpful to "piggyback" by overlaying additional information onto existing display elements.

This approach can be effective when it can take advantage of a diagram or other structure which is familiar to users. UML is arguably the *lingua franca* of software engineering and our users are likely to be familiar with object diagrams, class and sequence diagrams.

Visual attributes such as border thickness and colour can be used to show the quantities such as the number of updates at a particular breakpoint.

The lower left pane visible in Figure 7 shows an augmented UML object diagram. Colour is used to indicate "freshness" (e.g. blue for an object seen for the first time)

4.4 DOI & focus+context

The Degree of Interest of a point (i.e. data item) x, given a focus at x_f is given by

$$DOI(x|. = x_f) = API(x) - D(x, x_f) \qquad (1)$$

API is the *a priori* interest of x. For example we might consider that methods are more important than fields, and that concrete elements are more important than abstract ones. D represents a (conceptual) distance between x and x_f. Thus we might consider length of a call chain or number of generations of inheritance to indicate relative distance between elements.

The Functional form is application-dependent and we are free to allow this to be configurable by users to better support specific tasks.

The augmented object diagram at the lower left of Figure 7 also illustrates the use of DOI-based visualisation. The font size indicates the "distance" of the corresponding identifier from the neighbourhood of the visualisation's focus.

4.5 Augmented tree view

Another effective visualisation technique involves overlaying additional information *in situ* to augment an existing display without diverting the user's focus of attention (Harward et al. 2010).

Figure 8 shows colour being used to augment a breakpoint tree view with information about activity

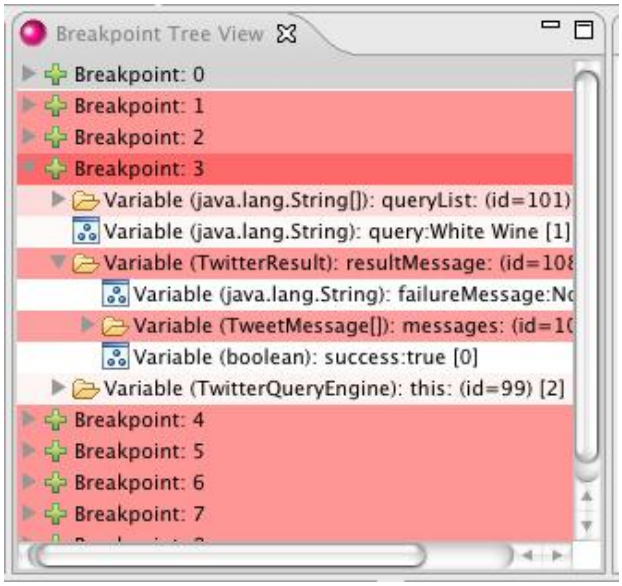


Figure 8: *In situ* augmentation

in the scopes corresponding to children. Expanding the nodes in the usual way reveals finer grained detail.

4.6 Conventional elements

Effective visualisation need not involve "fancy" graphics or 3D effects. Our system supports such conventional diagrams as line graphs. The line graph at the upper right of Figure 7 shows the time intervals between successive activations of a Pseudo-Breakpoint. In this case, the peak corresponds to the creation of an object which caused network authentication to occur, leading to a slight delay.

5 Conclusions

We have proposed Pseudo-Breakpoints as a mechanism for supporting visualisation-driven debugging an other activities requiring understanding of dynamic software state. We have implemented Pseudo-Breakpoints and a range of visualisations in Eclipse and these are available in the form of a plug-in.

Our approach allows visualisations to be integrated into the GUI of an industrial strength IDE and for users to use them alongside conventional breakpoints and their interface in a natural way.

We are encouraged by our results thus far and this work is continuing.

One important activity is the evaluation of the effectiveness of our approach. To date, this has consisted of some anecdotal feedback and "eating our own dog food." Our intention is to conduct a heuristic evaluation (Nielsen 1992) to guide the ongoing development of the system. A parallel project is gathering usage data and we hope that this will shed light on debugging practices among our target users.

Another possible direction is the ability to record and play back dynamic state information as this is useful in a range of contexts.

References

Brown, M. H. & Sedgewick, R. (1984), 'A system for algorithm animation', *SIGGRAPH Comput. Graph.* **18**(3), 177–186.

Churcher, N. & Irwin, W. (2005), Informing the design of pipeline-based software visualisations, *in* S.-H. Hong, ed., 'APVIS2005: Asia-Pacific Symposium on Information Visualisation', Vol. 45 of *Conferences in Research and Practice in Information Technology*, ACS, Sydney, Australia, pp. 59–68.

Churcher, N., Frater, S., Huynh, C. P. & Irwin, W. (2007), Supporting OO design heuristics, *in* J. Grundy & J. Han, eds, 'ASWEC2007: Australian Software Engineering Conference', IEEE, Melbourne, Australia, pp. 101–110.

Churcher, N., Irwin, W. & Kriz, R. (2003), Visualising class cohesion with virtual worlds, *in* T. Pattison & B. Thomas, eds, 'Australasian Symposium on Information Visualisation, (invis.au'03)', Vol. 24 of *Conferences in Research and Practice in Information Technology*, ACS, Adelaide, Australia, pp. 89–97.

Churcher, N., Keown, L. & Irwin, W. (1999), Virtual worlds for software visualisation, *in* A. Quigley, ed., 'SoftVis99 Software Visualisation Workshop', University of Technology, Sydney, Australia, pp. 9–16.

Czyz, J. & Jayaraman, B. (2007), Declarative and visual debugging in Eclipse, *in* 'Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange', ACM, pp. 31–35.

Deaker, C., Pettigrew, L., Churcher, N. & Irwin, W. (2010), Software visualisation with tag clouds, *in* J. Hosking & B. Long, eds, 'ASWEC 2010 Industry Track Proceedings', Auckland, New Zealand, pp. 129–133.

Eades, P. & Zhang, K., eds (1996), *Software Visualisation*, Vol. 7 of *Series on Software Engineering and Knowledge Engineering*, World Scientific.

Furnas, G. (1986), Generalised fisheye views, *in* 'Proc ACM SIGCHI '86 Conference on Human Factors in Computing Systems', pp. 16–23.

Harward, M., Irwin, W. & Churcher, N. (2010), In situ software visualisation, *in* J. Noble & C. Fidge, eds, 'ASWEC 2010', IEEE, Auckland, New Zealand, pp. 171–180.

Irwin, W. & Churcher, N. (2002), XML in the visualisation pipeline, *in* D. D. Feng, J. Jin, P. Eades & H. Yan, eds, 'Visualisation 2001', Vol. 11 of *Conferences in Research and Practice in Information Technology*, ACS, Sydney, Australia, pp. 59–68. Selected papers from 2001 Pan-Sydney Workshop on Visual Information Processing.

Irwin, W. & Churcher, N. (2003), Object oriented metrics: Precision tools and configurable visualisations, *in* 'METRICS2003: 9th IEEE Symposium on Software Metrics', IEEE Press, Sydney, Australia, pp. 112–123.

Irwin, W., Cook, C. & Churcher, N. (2005), Parsing and semantic modelling for software engineering applications, *in* P. Strooper, ed., 'Australian Software Engineering Conference', IEEE Press, Brisbane, Australia, pp. 180–189.

Jacobs, T. & Musial, B. (2003), Interactive visual debugging with uml, *in* 'Proceedings of the 2003 ACM symposium on Software visualization', ACM, San Diego, California, pp. 115–122.

Johnson, B. & Shneiderman, B. (1991), Tree-maps: A space-filling approach to the visualization of hierarchical information structures, *in* G. Nielson & L. Rosenblum, eds, 'proc. Visialization '91', IEEE Computer Society Press, Los Alamitos, CA, pp. 284–291.

Lawrance, J., Bellamy, R. & Burnett, M. (2007), Scents in programs:does information foraging theory apply to program maintenance?, *in* 'Visual Languages and Human-Centric Computing, 2007. VL/HCC 2007. IEEE Symposium on', pp. 15 –22.

Leung, Y. K. & Apperley, M. D. (1994), 'A review and taxonomy of distortion-oriented presentation techniques', *ACM Transactions on Computer-Human Interaction* **1**(2), 126–160.

Lnnberg, J., Korhonen, A. & Malmi, L. (2004), Mvt: a system for visual testing of software, *in* 'Proceedings of the working conference on Advanced visual interfaces', ACM, Gallipoli, Italy, pp. 385–388.

McGavin, M., Wright, T. & Marshall, S. (2006), Visualisations of execution traces (vet): an interactive plugin-based visualisation tool, *in* 'Proceedings of the 7th Australasian User interface conference - Volume 50', Australian Computer Society, Inc., Hobart, Australia, pp. 153–160.

Mehner, K. (2002), Javis: A uml-based visualization and debugging environment for concurrent java programs, *in* 'Revised Lectures on Software Visualization, International Seminar', Springer-Verlag, pp. 163–175–.

Neate, B., Irwin, W. & Churcher, N. (2006), Coderank: A new family of software metrics, *in* J. Han & M. Staples, eds, 'ASWEC2006: Australian Software Engineering Conference', IEEE, Sydney, pp. 369–378.

Nielsen, J. (1992), Finding usability problems through heuristic evaluation, *in* 'Proceedings of the SIGCHI conference on Human factors in computing systems', ACM Press, pp. 373–380.

Parnas, D. L. (1975), Software engineering or methods for the multi-person construction of multiversion programs, *in* C. E. Hackl, ed., 'Programming Methodology, 4th Informatik Symposium', Vol. 23 of *Lecture Notes in Computer Science*, Springer-Verlag, Wildbad, Germany, pp. 225–235.

Sarkar, M. & Brown, M. (1994), 'Graphical fisheye views', *Communications of the ACM* **37**(12), 73–84.

Spence, R. (2001), *Information Visualisation*, Addison-Wesley.

Stasko, J. (1990), 'Tango: a framework and system for algorithm animation', *IEEE Computer* **23**(9), 27 – 39.

Stasko, J., Domingue, J., Brown, M. & Price, B., eds (1998), *Software Visualization: Programming as a Multimedia Experience*, MIT Press.

Vans, A. M., von Mayrhauser, A. & Somlo, G. (1999), 'Program understanding behaviour during corrective maintenance of large-scale software', *Int. J. Human-Computer Studies* **51**(1), 37–70.

Ware, C. (2004), *Information Visualization: Perception for Design*, 2nd edn, Morgan Kaufman.