



# Durham E-Theses

---

## *Runtime visualisation of object-oriented software*

Smith, Michael Philip

### How to cite:

---

Smith, Michael Philip (2003) *Runtime visualisation of object-oriented software*, Durham theses, Durham University. Available at Durham E-Theses Online: <http://etheses.dur.ac.uk/3732/>

### Use policy

---

The full-text may be used and/or reproduced, and given to third parties in any format or medium, without prior permission or charge, for personal research or study, educational, or not-for-profit purposes provided that:

- a full bibliographic reference is made to the original source
- a [link](#) is made to the metadata record in Durham E-Theses
- the full-text is not changed in any way

The full-text must not be sold in any format or medium without the formal permission of the copyright holders.

Please consult the [full Durham E-Theses policy](#) for further details.

# **Runtime Visualisation of Object-Oriented Software**

**Michael Philip Smith**

Department of Computer Science  
University of Durham

1999-2003

June 2003

**PhD Thesis**

**A copyright of this thesis rests with the author. No quotation from it should be published without his prior written consent and information derived from it should be acknowledged.**



19 JAN 2004

Thesis  
2003/  
SM1

# Abstract

Software is a complex and invisible entity, yet one which is core to modern life. The development and maintenance of such software includes one staple task, the need to understand the software at the implementation level. This process of program comprehension is difficult and time consuming. Yet, despite its importance, there remains very limited tool support for program comprehension activities.

The results of this research show the role that runtime visualisation can play in aiding the comprehension of object-oriented software by highlighting both its static and dynamic structure. Previous work in this area is discussed, both in terms of the representations used and the methods of extracting runtime information. Building on this previous work, this thesis develops new representations of object-oriented software at runtime, which are then implemented in a proof of concept tool. This tool allowed the representations to be investigated on real software systems. The representations are evaluated against two feature-based evaluation frameworks. The evaluation focuses on generic software visualisation criteria, due to the lack of any specific frameworks for visualising dynamic information. The evaluation also includes lessons learnt in the implementation of a prototype visualisation tool.

The object-oriented paradigm continues to grow in popularity and provides advantages to program comprehension activities. However, it also brings a number of new challenges to program comprehension due to the discrepancies between its static definition and its runtime structure. Therefore, techniques that highlight both the static definition and the runtime behaviour of object-oriented systems offer benefits to their comprehension.

Software visualisation offers an approach to aid program comprehension activities through providing a means to deal with the size and complexity of the software and its invisible nature.

This thesis highlights the generic issues that software visualisation faces, before focusing on how the visualisation of runtime information affects these issues. Many of the issues are compounded by the dynamic nature of the information to be visualised and the explosive growth in the volume of information that this dynamism can bring.

Wider results of this research have allowed the proposal of the necessary concepts that should be considered in the design and evaluation of runtime visualisations.

Software visualisation at runtime is still a relatively unexplored area and there remains many research challenges within it. This thesis aims to act as a first step to addressing these challenges and aims to promote interest and future development within this area.

# Acknowledgements

First and foremost, a special thanks goes to my wife, Julie, for all her support and encouragement throughout my time studying for my PhD. Also for all the time she spent reading various documents and thesis drafts.

I would also like to thank my family, and in particular, my parents, Anne and Roger, for their continual support throughout my entire education. Without their help and support I would not have got this far.

Thanks also to Malcolm Munro who has been an excellent supervisor and provided much encouragement and feedback.

Many thanks go to Julie, Chris Taylor, Jill Munro and Claire Knight for proof reading this thesis. I greatly appreciate all the time they spent and the helpful feedback that I received.

I wish to thank all the members of the VRG and the Computer Science department with whom I have shared many good times.

Finally, I would like to thank the EPSRC for funding this research.

## Copyright

The copyright of this thesis rests with the author. No quotation from this thesis should be published without prior written consent. Information derived from this thesis should also be acknowledged.

## Declaration

No part of the material provided has previously been submitted by the author for a higher degree in the University of Durham or in any other University. All the work presented here is the sole work of the author and no-one else.

This research has been documented, in part, within the following publications:

- **M. Smith** and **M. Munro**, *Runtime Visualisation of Object Oriented Software*, Proceedings of the IEEE 1<sup>st</sup> International Workshop on Visualizing Software for Understanding and Analysis, Paris, pages 81-89, June 2002.
- **A. S. Hatch**, **M. P. Smith**, **C. M. B. Taylor** and **M. Munro**, *No Silver Bullet for Software Visualisation Evaluation*, Proceedings of the Workshop on Fundamental Issues of Visualization, Proceedings of The International Conference on Imaging Science, Systems and Technology (CISST), Las Vegas, USA, pages 651-657, June 2001.

# Contents

<b>Abstract</b> .....	<b>i</b>
<b>Acknowledgements</b> .....	<b>ii</b>
<b>Copyright</b> .....	<b>iii</b>
<b>Declaration</b> .....	<b>iii</b>
<b>Contents</b> .....	<b>iv</b>
<b>List of Figures</b> .....	<b>viii</b>
<b>List of Tables</b> .....	<b>x</b>
<b>Chapter 1 Introduction</b> .....	<b>1</b>
1.1 Introduction.....	2
1.2 Objectives .....	4
1.3 Criteria for Success .....	5
1.4 Thesis Overview.....	5
<b>Chapter 2 Program Comprehension</b> .....	<b>7</b>
2.1 Introduction.....	8
2.2 Key terminology.....	8
2.2.1 Cognitive model.....	8
2.2.2 Mental model .....	8
2.2.3 Knowledge base .....	9
2.2.4 Assimilation process .....	9
2.3 Key Theories.....	10
2.3.1 Shneiderman and Mayer.....	10
2.3.2 Brooks .....	10
2.3.3 Wiedenbeck.....	11
2.3.4 Soloway and Ehrlich .....	11
2.3.5 Pennington .....	12
2.3.6 Littman et al. ....	13
2.3.7 Letovsky.....	13
2.3.8 Von Mayrhauser and Vans.....	14
2.4 Summary .....	15
<b>Chapter 3 Software Visualisation</b> .....	<b>16</b>
3.1 Introduction.....	17
3.2 Definition .....	17
3.3 The Need for Visualisation .....	18
3.4 Existing Taxonomies.....	19
3.4.1 Myers .....	19
3.4.2 Price et al.....	20
3.4.3 Roman and Cox.....	21
3.4.4 Summary of Taxonomies .....	22

3.5	An Overview of Current Software Visualisation Systems .....	23
3.6	Issues and Challenges .....	31
3.7	Conclusions .....	32
<b>Chapter 4 Software Visualisation at Runtime .....</b>		<b>34</b>
4.1	Introduction .....	35
4.2	Debugging Tools .....	35
4.3	Extracting runtime information .....	36
4.4	Online Vs. Offline Approaches .....	37
4.5	Dynamic Software Visualisation Tools .....	38
4.5.1	NestedVision3D (NV3D) .....	38
4.5.2	Virtual Images: Interactive Visualisation of Distributed Object-Oriented Systems .....	40
4.5.3	VizBug++ .....	41
4.5.4	Look! .....	42
4.5.5	Jinsight .....	44
4.5.6	Program Explorer .....	45
4.5.7	HotWire .....	46
4.5.8	VisiVue™ .....	47
4.6	The Benefits for Object-Oriented Software .....	48
4.7	Current Trends and Issues and Challenges .....	49
4.8	Conclusions .....	52
<b>Chapter 5 The DJVis Approach .....</b>		<b>53</b>
5.1	Introduction .....	54
5.2	Terms .....	54
5.3	Language Choice .....	54
5.3.1	Language Constructs .....	55
5.4	DJVis .....	58
5.4.1	Overview .....	58
5.4.2	DJVis Views .....	61
5.4.2.1	The Runtime View .....	61
5.4.2.2	The Class View .....	66
5.4.2.3	The Variable Watch View .....	75
5.4.2.4	The Method Pixel View .....	80
5.4.2.5	The Query View .....	82
5.4.3	General Features of DJVis .....	87
5.5	Conclusions .....	89
<b>Chapter 6 Implementation .....</b>		<b>90</b>
6.1	Introduction .....	91
6.2	Information Extraction Method .....	91
6.3	DJVis Prototype Implementation .....	91
6.4	Limitations .....	93
6.5	Use of the Prototype Tool .....	93



6.6	Conclusions .....	95
<b>Chapter 7 Evaluation Approach .....</b>		<b>96</b>
7.1	Introduction .....	97
7.2	Evaluation Techniques .....	97
7.3	Chosen Evaluation Approach.....	100
7.4	Scenarios .....	104
7.5	Conclusions .....	105
<b>Chapter 8 DJVis Evaluation .....</b>		<b>106</b>
8.1	Introduction .....	107
8.2	Informal Evaluation of DJVis .....	107
8.2.1	The Views .....	107
8.2.2	The Visualisation and implementation approach .....	114
8.3	Application of the Frameworks to DJVis.....	118
8.4	Scenarios .....	130
8.4.1	Scenario 1: Corrective Maintenance .....	130
8.4.1.1	Task.....	130
8.4.1.2	Information Requirements.....	130
8.4.1.3	Application of DJVis.....	130
8.4.2	Scenario 2: Code Familiarisation .....	132
8.4.2.1	Task.....	132
8.4.2.2	Information Requirements.....	132
8.4.2.3	Application of DJVis.....	132
8.4.3	Scenario 3: Preventative Maintenance .....	134
8.4.3.1	Task.....	134
8.4.3.2	Information Requirements.....	134
8.4.3.3	Application of DJVis.....	134
8.4.4	Scenario 4: Impact Analysis.....	137
8.4.4.1	Task.....	137
8.4.4.2	Information Requirements.....	137
8.4.4.3	Application of DJVis.....	137
8.4.5	Scenario 5: Test Case Validation .....	138
8.4.5.1	Task.....	138
8.4.5.2	Information Requirements.....	138
8.4.5.3	Application of DJVis.....	139
8.5	Case Study: Understanding GraphTool.....	140
8.6	Conclusions .....	151
<b>Chapter 9 Conclusions .....</b>		<b>154</b>
9.1	Introduction .....	155
9.2	Summary of Research .....	155
9.3	Criteria for Success .....	156
9.4	Future Work .....	159

9.5 Conclusion .....	160
<b>Abbreviations .....</b>	<b>162</b>
<b>References.....</b>	<b>163</b>

# List of Figures

Figure 3-1 The SHriMP System.....	24
Figure 3-2 SeeSoft [Eick92] showing 4000 lines of code. The oldest lines are in blue and the newest lines in red. Image from Stephen Eick's web page <a href="http://www.bell-labs.com/user/eick/SoftwareVis.html">http://www.bell-labs.com/user/eick/SoftwareVis.html</a>	25
Figure 3-3 The Execution Mural of an object oriented programs message trace [Jerd96a]. Image from: <a href="http://www.cc.gatech.edu/gvu/softviz/infviz/information_mural.html">http://www.cc.gatech.edu/gvu/softviz/infviz/information_mural.html</a> .....	26
Figure 3-4 Revision Tower representation of source code version information [Tayl02] .....	27
Figure 3-5 3D representation of UML [Dwye01]. Image available at: <a href="http://www.wilmascope.org/">http://www.wilmascope.org/</a> .....	28
Figure 3-6 FileViz showing an overview of files in a software system. ....	29
Figure 3-7 A view of part of a city district in Software World [Knig00].....	30
Figure 4-1 Objects name used to create representation [Vion94] .....	40
Figure 4-2 An overview of the system [Vion94] .....	40
Figure 4-3 VizBug++ showing class and instance relations [Jerd94].....	41
Figure 4-4 Look! showing object references.....	43
Figure 4-5 The cluster and class views respectively .....	43
Figure 4-6 Jinsight showing execution view.....	44
Figure 4-7 Histogram View showing object relations.....	44
Figure 4-8 Reference Pattern View.....	45
Figure 4-9 Program Explorer showing multiple views [Lang95] .....	46
Figure 4-10 Hotwire [Laff94] .....	47
Figure 4-11 VisiVue™ .....	48
Figure 5-1 An overview of the information shown by each view. ....	60
Figure 5-2 The Thread Group Hierarchy .....	62
Figure 5-3 A single thread .....	62
Figure 5-4 A possible scaling of methods on the stack.....	63
Figure 5-5 Methods on a call stack .....	63
Figure 5-6 Details of a method on the stack.....	64
Figure 5-7 Presenting an overview of control flow information.....	65
Figure 5-8 Viewing the GraphDesktop class in Class View .....	67
Figure 5-9 Class View showing static and dynamic references .....	68
Figure 5-10 Showing field variables in the Class View.....	69
Figure 5-12 Showing data encapsulation using the Class View.....	69
Figure 5-13 Displaying User Abstractions.....	70
Figure 5-14 Embedded Abstractions.....	71
Figure 5-15 Showing Package Inclusion in the Class View.....	71
Figure 5-16 Displaying the method name and source code in the Class View .....	72
Figure 5-17 Creating a user filter.....	73
Figure 5-18 Length Mapping Modes .....	74
Figure 5-19 The Variable Watch View showing field access by type and frequency.....	76
Figure 5-20 Multiple patterns in the Variable Watch View .....	77
Figure 5-21 Displaying class and method accesses through the use of method lines .....	79

Figure 5-22 Methods of the GraphDesktop class shown in the Method Pixel View.....	80
Figure 5-23 Showing two thread groups in the Query View.....	83
Figure 5-24 Displaying loaded classes.....	84
Figure 5-25 Visualising the structure of a class in the Query View.....	84
Figure 5-26 Showing an overview of the methods and fields.....	85
Figure 5-27 Detailed view of a class in the Query View .....	86
Figure 5-28 Showing a class has user annotated documents in the Class View.....	87
Figure 5-29 Showing annotation levels.....	87
Figure 5-30 Displaying annotation details for the class MessageCatalog.....	88
Figure 6-1 Generating a visualisation within the prototype tool.....	94
Figure 7-1 Cognitive Design Elements for Software Exploration [Stor97a] .....	101
Figure 8-1 Navigation options within the Runtime and Query Views .....	108
Figure 8-2 Game based navigation aid for showing the position of objects relative to the user [Frontier]. .....	109
Figure 8-3 Method Calling shown in the Class View .....	131
Figure 8-4 Focusing on three threads of interest in the Query View .....	133
Figure 8-5 Showing the class structure of a web server.....	135
Figure 8-6 Inspecting for inheritance candidates .....	136
Figure 8-7 Inspecting the inheritance/implements structure of four classes in the Query View.....	136
Figure 8-8 Gaining an overview of the method calling relationships for the Lexer class using the Method Pixel View.....	138
Figure 8-9 Custom mapping for the easy identification of methods which are not called. ....	139
Figure 8-10 Highlighting uncalled methods.....	140
Figure 8-11 GraphTool classes after initialisation as shown in the Class View .....	141
Figure 8-12 Identification of interesting features in the main sub graph. ....	142
Figure 8-13 Classes loaded as a result of displaying the Preferences Dialog .....	143
Figure 8-14 Displaying the field information for the Graph class .....	145
Figure 8-15 Investigating field names and types using mouse over information.....	145
Figure 8-16 Investigating the threading structure of GraphTool.....	147
Figure 8-17 Showing the methods involved in the horizontal layout functionality of GraphTool .....	148
Figure 8-18 Investigating the fields of the Node and Edge classes.....	149
Figure 8-19 Close up of Edge class showing field types .....	149
Figure 8-20 Watching the from_edges vector of the Node class in the Variable Watch View.....	150

## List of Tables

Table 3-1 Software World mapping [Knig00] .....	30
Table 5-1 Dynamic relationships between Java items in an executing Java program.....	58
Table 5-2 Object level relationships .....	58
Table 5-3 Viewing the different relationships in the different views.....	60
Table 5-4 Class View Representations .....	67
Table 5-5 Possible interpretations of patterns from Figure 5-18.....	78
Table 7-1 Shneiderman's seven tasks [Shne96] .....	97
Table 8-1 Information that is unavailable using the JPDA .....	115
Table 8-2 Application of Storey et al. [Stor97a] framework to DJVis.....	121
Table 8-3 Application of Knight's [Knig00] framework to DJVis .....	126

# Chapter 1 Introduction



## 1.1 Introduction

The objective of this thesis is to investigate the issues involved in understanding object-oriented code. This takes the form of showing the execution behaviour of object-oriented code using visualisation techniques.

The task of comprehending software is central to the majority, if not all, of software engineering tasks. This is particularly true for understanding the software's implementation details. Program comprehension is not an easy task and it can be time-consuming and problematic, even for experts. Software is highly complex and program comprehension requires gaining an understanding of the complex and varied relationships between its constructs. This understanding of the software is typically based on its source code and documentation. However, documentation and designs can often be out-of-date and differ from the actual structure of the software. The pressures of deadlines, poor coding standards and unrecorded changes can often mean that software is significantly different in its implementation and structure compared to that defined in the documentation on the software. However, it is difficult to understand a program from the source code alone, due to the complexity and size of source code that defines real world software and the low-level nature of its description. Relationships between source code constructs are not obvious and it can be difficult to find the particular piece of source code of interest that implements a particular feature, or follow the code in a logical manner, such as through the calling structure. Good coding practices, structuring, encapsulation and cross-referencing and searching facilities in source code editors can all help, but they cannot solve the problem due to the inherently complex and abstract nature of software.

Software development is a team activity and therefore each team member will often need to understand the code developed by other members of the team, for example, if it interacts with their code, or there is a bug that needs fixing. Thus there is a need for program comprehension of unfamiliar code as well as their own. High staff turnover can compound this problem, especially if additional programmers join a project in the middle of its development stage as they will have no existing knowledge of the software. These new programmers will need to gain an understanding of the current structure of the software, before they can be most effective. Loss of existing staff also means that a large amount of typically undocumented knowledge and experience is lost. It will take a new employee a significant amount of time to reach this level of knowledge and experience. These issues also affect the maintenance of the software.

Software maintenance is:

*“The process of modifying a software system or component after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment.” [IEEE]*

Software maintenance can be categorised into four subsections [Benn91]:

- Perfective maintenance: Enhancing the original program by adding new functionality.
- Corrective maintenance: The fixing of errors and bugs in the original program.
- Adaptive maintenance: The adaptation of the original program to a changed

environment e.g. Operating system.

- Preventative maintenance: Reengineering the original program to improve its structure and quality in order to allow for easier future maintenance.

All of these maintenance activities require some level of program comprehension and they can also offer increased challenges compared to development activities. The people maintaining a piece of software will normally not have developed it. Therefore, they have no initial experience or knowledge of its structure or design to aid them in their initial maintenance activities. Existing documentation will often be out-of-date, especially if the software has already undergone maintenance by another team. Software maintenance consumes a large amount of resources in the software lifecycle and much of the time involved in software maintenance is spent on program comprehension activities. Program comprehension is needed in order to carry out maintenance tasks, for instance in a corrective maintenance task it may be necessary to understand at the code level when an unhandled exception is thrown. Therefore, any improvements offered to program comprehension tasks will be of substantial benefit to the development and maintenance of software.

The Object-Oriented (OO) paradigm has grown increasingly popular in recent years and it has brought a number of advantages. Its supporters claim that it allows increased reusability and understanding by maintainers, due to its ability to encapsulate and provide inheritance of existing classes. Such facilities allow developers to focus their program comprehension activities and reduce the understanding they need of other classes, often to just the class' interface and the functionality it provides. However, the object-oriented paradigm is certainly not a silver bullet and program comprehension of OO systems is still a difficult and time-consuming task. With its benefits to program understanding come new challenges. OO programs are comprised of a network of communicating objects at runtime and this results in a large discrepancy when compared to the static class descriptions of the program. This discrepancy is increased by the use of inheritance and dynamic binding. The latter make it impossible to see the actual calling structure based on the static source code alone, because the actual call is only resolved at runtime. When there are a large number of objects inheriting from a class, it can be difficult to ascertain which actual object type is referenced by variables of that type. For instance, is it an object of the variable type, or an object of one of the classes, which inherit from it? The aforementioned techniques are beneficial to a developer, as for example, the same data structure can be used for all objects by simply storing references to a base class and this reduces the need to understand and debug multiple variations of the same data structure. However, this makes it increasingly difficult to see how each instance of the data structure is used.

Visualisation is defined as [OED]:

*"The process of forming a mental picture or vision of something not actually present to the sight"*

This is a generic dictionary definition of visualisation, however it captures the goal of building a mental picture. Visualisation involves the representation of data in order to aid the user's understanding and this representation is typically based on graphical representations. Sophisticated visualisation techniques are increasingly available on desktop PCs, as graphical hardware has become very powerful and affordable due to its growth in areas such as games and entertainment. Visualisation has been applied to a number of



areas, such as information and scientific visualisation and has proved successful in aiding such tasks, for example, in visualising airflow in aerodynamic studies. Software visualisation is the application of visualisation techniques to aid in software understanding tasks, and provides an approach to assist with program comprehension tasks. One of the problems in program comprehension is dealing with the vast amount of information and trying to understand the complex and often hidden relationships between software constructs. Software visualisation can aid such tasks by presenting information on the program in a different form to its source code. This may be achieved by using techniques such as abstraction, filtering and pattern recognition to help the user understand the software's structure and behaviour. However, software offers a number of challenges to visualisation techniques. Software is an abstract entity and massively complex, encompassing a vast number of varied relationships between different constructs. Therefore, the application of visualisation techniques to software is not an easy task and designing a suitable representation is difficult and problematic. Software visualisation cannot rely on intuitive real world representations, as is often the case in scientific visualisation, where the data typically represents some physical item. Much research is therefore needed into software visualisation representations.

Runtime visualisation is defined, for the purposes of this thesis, to be the visualisation of the runtime behaviour of software. Software visualisation can be applied to both the statically available details of software and the dynamic details available as the program executes. Runtime visualisation offers a way to aid the understanding of maintainers and developers by showing the actual behaviour and structure of the software as it executes, using dynamically extracted facts. Runtime visualisation provides an approach to deal with the discrepancies between the static description of the software and its runtime behaviour. It is therefore particularly useful when applied to object-oriented software where these discrepancies are especially prevalent. Runtime visualisation faces the generic software visualisation challenges, however, the dynamic aspects together with the vast volume of information available on an executing program result in new intricacies in these challenges, such as those of scale and representation.

## **1.2 Objectives**

Program comprehension is an area that offers significant potential for improvements through tool support. It is a major task in many software engineering activities and it is currently time-consuming and therefore costly. This research aims to focus on the application of visualisation techniques in order to aid program comprehension tasks. This will specifically focus on the visualisation of object-oriented systems, due to the growth of this paradigm. Object-oriented software also offers additional challenges due to the discrepancies between its static specification and its dynamic structure. Therefore, it is the dynamic structure that will be visualised in order to highlight the dynamic nature of the paradigm. Effective aids to program comprehension will help to maximise the benefits of object-oriented development.

This research will investigate the current state of runtime visualisation research and highlight the research issues and challenges that the field still faces. These issues will then drive the development of new representations of object-oriented programs, whose aim is to improve program comprehension tasks on object-oriented software. These representations will need to take into account the issues and challenges for runtime visualisation, as well as generic software visualisation challenges. Issues, such as scale and

the abstract nature of software, will need to be considered in the design of the new representations. The developed representations will need to be able to show the structure of the software as it executes and at various levels of abstraction, in order to allow the user to deal with the large volume of information.

An important aspect of a software visualisation, and in particular, a runtime visualisation, is the ability to generate the visualisation automatically without the need to modify the original source code of the program. This allows the visualisation to be applied to real world programs. This research discusses the possible information extraction techniques for gaining access to details of the software's execution. The visualisations developed as part of this research will be incorporated into a proof of concept tool, which will allow both the visualisations and the information extraction technique to be evaluated. This will focus on the investigation of debugging techniques for runtime visualisation.

The visualisations produced by this research will aim to aid program comprehension activities. However, they will only be one step towards addressing the issues involved. Therefore, this research also aims to discuss issues that should be considered in the design and evaluation of runtime visualisations as a means of driving future research. Areas for future development both in terms of the ideas presented in this thesis, and in runtime visualisation generally, will also be discussed at the end of the thesis.

### **1.3 Criteria for Success**

This research aims to investigate the applicability of visualisation to the runtime behaviour of software in order to aid in maintenance tasks. The success of the research will be judged against the following criteria:

- a) Address the visualisation issues of representing an object-oriented language such as Java at runtime.
- b) Develop new visual representations of object-oriented system at runtime.
- c) Provide various levels of abstractions in visualisations of runtime information.
- d) Develop a proof of concept prototype tool to demonstrate visualisations.
- e) Show the applicability of these concepts to maintenance tasks using this proof of concept tool.
- f) Demonstrate that the visualisations can be generated automatically with the programmer needing no knowledge of the structure of the software under study.

These criteria are revisited in chapter 9, where they are compared against the research achieved.

### **1.4 Thesis Overview**

This thesis is structured into a number of chapters, each of which addresses a different aspect of the research and its background. The remaining 8 chapters can be summarised as follows:

Chapter 2 introduces the program comprehension field and presents an overview of the main concepts involved. Key work within the field is presented, thus allowing an understanding of the current state of research to be gained.

Chapter 3 introduces software visualisation. The different definitions and sub areas of software visualisation are presented, followed by the benefits that software visualisation can offer. This chapter also provides the definition of software visualisation as used by this research. The chapter provides an overview of what the field entails by summarising three of the major taxonomies on software visualisation and thus showing where this research fits into the larger field. A number of existing visualisations are also presented to show the current state of the field and demonstrate some of the representations that exist. Finally, software visualisation faces a number of generic challenges and issues and these are outlined in this chapter.

Chapter 4 goes on to focus on software visualisation at runtime. This chapter presents an overview of the existing work that has been performed in this area through the presentation of a number of runtime visualisations. A summary of the different techniques for information extraction is also presented which outlines the respective benefits and drawbacks of each. The chapter outlines why this research focuses on visualising object-oriented software and the particular benefits that can be gained for aiding the understanding of object-oriented software. The chapter concludes with a discussion of the specific issues and challenges that runtime visualisation presents.

Chapter 5 introduces the DJVis approach and defines a number of visualisations of runtime information that aim to address the issues identified in the previous chapter. Each of these visualisations forms a view that shows some particular aspect of the executing software. Each of these views is discussed along with issues that affect all views and the interoperation of the views.

Chapter 6 discusses the implementation of the prototype tool version of DJVis. The techniques and technologies used are briefly described, as well as, a guide on how the prototype tool can be used to visualise Java programs without the need for programmer intervention.

Chapter 7 introduces the evaluation of DJVis. The chapter outlines the different evaluation techniques that could be applied, providing details of the advantages and disadvantages of each. From this, the chosen approach of two feature-based frameworks and multiple usage scenarios are described in more detail.

Chapter 8 goes on to show the application of the evaluation approach to DJVis. This is preceded by an informal discussion of the merits and issues of the main aspects of DJVis and the implementation techniques used for the prototype tool. The feature-based frameworks are then applied followed by five usage scenarios and one in-depth case study.

Chapter 9 draws the thesis to a conclusion by providing an overview of the research and discussing its contributions. The criteria for success defined in chapter 1 are also re-examined against the achieved results. Finally, areas for future research are outlined for both DJVis and runtime visualisation in general.

# Chapter 2 Program Co mprehension

## 2.1 Introduction

Program comprehension is the task of understanding how a program is constructed and how it operates. Therefore, program comprehension is essential to the task of modifying or maintaining a program. It is a major component of any software maintenance task, occupying 50-90% of the maintenance time according to some estimates [Stan84]. It is also present in the initial software development process, through tasks such as code reviews, debugging and some testing strategies. There are a number of theories on the methods used in program comprehension, varying between top-down and bottom-up techniques. An introduction to the main terminology and ideas will be presented, followed by summaries of the main work in the field.

## 2.2 Key terminology

A number of terms are used in the various program comprehension theories, however Von Mayrhauser and Vans [Mayr95] provide a summary of the key theories and draw similarities between them. All the approaches use existing knowledge, combined with a comprehension strategy in order to acquire new knowledge and understanding of the program. A brief summary of the general terminology is presented below, this includes the cognitive model, mental model, knowledge base and assimilation process. The key theories are then summarised to show the individual approaches.

### 2.2.1 Cognitive model

A key term is that of the *cognitive model*, which refers to the complete set of processes, mental models, knowledge and heuristics used in program comprehension. This is composed of three main components: a *mental model*, a *knowledge base* and an *assimilation process*.

### 2.2.2 Mental model

The *mental model* is the programmer's internal, working representation of the software and is continuously updated as the comprehension process proceeds. It is made up of semantic constructs and a number of definitions for these exist within the different theories, which are summarised by Von Mayrhauser and Vans [Mayr95].

*Text structure* knowledge is made up from the program text and its structure. Examples of this include conditional constructs, such as IF constructs, variables and function definitions and looping constructs.

*Chunks* are constructed from various levels of text structure abstractions. Macrostructures are text structure chunks identifiable by a label. For example, a sort routine macrostructure is simply its label sort, representing the abstracted code microstructures (individual code statements) that make up the sort code block. *Chunking* is the process of creating new higher level abstractions from existing lower level abstractions.

*Plans* are “*program fragments that represent stereotypic action sequences in programs*” [Solo84]. Von Mayrhauser and Vans [Mayr95] define plans as “*knowledge elements for developing and validating expectations, interpretations and inferences*” and define them as either slot types or slot fillers. Slot types describe generic objects, for example data structures such as trees, whilst slot fillers are specialised for a specific task, with specific program fragments being an example of these. Von Mayrhauser and Vans further classify plans as programming plans or domain plans. Programming plans relate to programming knowledge and can vary in abstraction from high (e.g. abstract program functionality), to intermediate (e.g. data structures and algorithms), to low level (e.g. individual control statements). Domain plans contain knowledge of the problem area, such as knowledge on the real world operations of the software. This excludes low-level details of the code and algorithms.

*Hypotheses* [Broo83], is the method by which Brooks suggests that maintainers build a mapping between the problem domain (top level) and the programming domain (bottom level). They drive the direction of future investigations as they are refined, rejected or accepted.

*Conjectures* are how Letovsky [Leto86] refers to hypotheses. Letovsky classifies these, identifying three main types:

- Why conjectures, e.g. why a certain design choice?
- How conjectures, e.g. how a program goal is achieved?
- What conjectures, e.g. what does a variable do?

Associated with each conjecture is a degree of certainty, ranging from almost certain to uncertain guesses.

## 2.2.3 Knowledge base

The *knowledge base* is the maintainer’s understanding of the domain. It consists of both general knowledge and task specific knowledge and this can be existing knowledge or that newly acquired in the comprehension processes. General knowledge is things that do not relate directly to the task. An example of this is general knowledge about software engineering, such as general data structures and programming language knowledge. Task specific knowledge is knowledge that relates directly to the software under study, such as system goals or implicit business rules used by the system. If the maintainer has worked on the software before, they will have some (partial) mental model. Variations exist across the key theories over how this knowledge base is structured and the different levels of abstraction used.

## 2.2.4 Assimilation process

The *assimilation process* is the glue that binds the knowledge base to the mental model. This is the process by which the maintainer refines and updates their mental model using their knowledge base. A number of aids and strategies exist for knowledge acquisition.

*Rules of programming discourse* are rules of conventions in programming [Solo84] and these rules produce expectations with the maintainer on what should be in the program.

*Beacons* are idioms in programming or stereotypical code, which are typically associated with some functionality or operation [Broo83], for example, a function name. They act as cues that index into knowledge and allow a high-level understanding to be gained.

*Cross-referencing* is the process by which different abstraction levels are related allowing mappings from program parts to functional descriptions.

## **2.3 Key Theories**

This section presents some of the main theories on program comprehension and shows how some of the concepts in the terminology section were devised.

### **2.3.1 Shneiderman and Mayer**

Shneiderman and Mayer [Shne79] believe that comprehension relies on semantic and syntactic knowledge. This is stored in long term memory, each with differing levels of abstraction. Syntactic knowledge is knowledge of the programming language syntax and any specific issues of that programming language, such as library functions etc. It is more specific and detailed than semantic knowledge, and therefore, more easily forgotten. However, they suggest that it is easier for humans to learn a new syntactic representation for an existing semantic structure, than to learn a new semantic structure. As an example of this, they highlight learning a programming language. The first is difficult, because it requires both semantic and syntactic learning, whilst learning a subsequent language (of the same semantic structure e.g. imperative) is easier as only the syntax needs learning. Semantic knowledge consists of general programming concepts that are independent of a specific language. This can vary in abstraction from low level details, such as what data types are, to higher level concepts, such as searching techniques. Higher than this there may be domain knowledge, for example, knowledge to solve problems in an application area such as airline reservation systems.

Shneiderman [Shne80] then goes on to suggest that programmers abstract program information into *chunks*, which are used to build an internal semantic structure that represents the program. Chunks are syntactic or semantic abstractions of text structures within the source code. Sections of the source code are abstracted into chunks by the maintainer and then these chunks can be abstracted into higher level chunks. This represents a bottom-up approach to comprehension.

### **2.3.2 Brooks**

Brooks [Broo83] suggests a top-down approach to comprehension based on the hypothesis of a mapping between the problem domain (top level) and the programming domain (bottom level). The theory suggests that maintainers form a number of increasingly refined hypotheses about the program function instead of reading the program line by line. The initial hypothesis is formed from the first information on the program known by the maintainer. This can be a brief description of its purpose, or simply just its name. This initial hypothesis then sets up expectations with the maintainer of objects and operations to see in the program. Through comprehension the maintainer verifies the hypotheses from information on the program and rejects or modifies any which are not supported. This occurs until the maintainer has

sufficient knowledge to perform the maintenance task required. Brooks [Broo83] believes that a notion of *beacons* is used in the process of hypothesis verification by the maintainers, rather than them studying individual lines of code. *Beacons* are idioms in programming or stereotypical code, which are typically associated with some functionality or operation. For example, the common example is the swapping of two variables, which could be a beacon for a sorting routine.

### 2.3.3 Wiedenbeck

Wiedenbeck expanded on Brooks' notion of beacons and investigated them empirically in a number of studies [Wied86a, Wied86b, Wied91]. The results of these suggest an association between comprehension of programs, programmer expertise and beacon recognition [Wied91]. One study [Wied86a] found a significant difference in the number of lines containing beacons that could be recalled depending on the programmer's experience. Novices recalled only 14%, compared to 79% for experienced programmers, whereas there was not a significant difference for non-beacon lines. However, Wiedenbeck [Wied91] found a much smaller percentage of recall of a standard, non-disguised swap beacon by advanced programmers (33%), compared to the 78% in previous work [Wied86b]. The results must be viewed with caution as pointed out in the paper, due to design difficulties in the experiment. For example, the shellsort code used [Wied91] may have been familiar to some of the subject population. Despite this, Wiedenbeck believes that the accumulation of the results supports the role of beacons in program comprehension by stating "*the meaning of these findings is that the idiomatic or stereotypical code did appear to play a large role in the initial high level comprehension of programs*" [Wied91]. However, the studies also showed that a strong beacon can lead to miscomprehension, as discrepancies between the beacon and surrounding context information are not often noticed at the initial stage of comprehension.

### 2.3.4 Soloway and Ehrlich

Soloway and Ehrlich [Solo84] suggest that expert programmers use two types of programming knowledge, which novice programmers typically do not have.

- 1) *Programming plans.*
- 2) *Rules of programming discourse.*

They define programming plans as "*program fragments that represent stereotypic action sequences in programs*" [Solo84], for example, an item search loop plan. They take inspiration from work in the field of text comprehension where there is a notion of *schemas* which "*are generic knowledge structures that guide the comprehender's interpretations, inferences, expectations and attention when passages are comprehended*" [Grae81]. They view programming plans as corresponding directly to schema. Rules of programming discourse are the rules of convention in programming. These rules produce expectations with the maintainer on what should be in the program. They identify the following rules of discourse: (Fig. 2 from Soloway and Ehrlich [Solo84])

- (1) Variable name should reflect function.
- (2) Do not include code that will not be used.



- (2a) If there is a test for a condition, then the condition must have the potential to be true.
- (3) A variable that is initialized via an assignment statement should be updated via an assignment statement.
- (4) Do not do double duty with code in a non-obvious way.
- (5) An IF should be used when a statement body is guaranteed to be executed only once and a WHILE when a statement body may need to be repeatedly executed.

Using this, they view programs as being composed from programming plans altered to suit a specific problem, with the rules of programming discourse specifying how the plans are composed. They believe that programs that conform to these rules of discourse (plan-like programs) are easier to understand for expert programmers (who possess knowledge of plans and rules of discourse) than those that do not conform to the rules (unplan-like programs). They investigated this empirically, proposing that expert programmers are much better at understanding plan-like programs than novice programmers. However, when programs are unplan-like, the performance of the expert programmers reduces to that of novice programmers, due to the confusion caused by their strong expectations being broken by rule violations. Novice programmers are less sensitive as their lack of knowledge means they have fewer expectations. This theory was supported by the study's results showing that programming plans and rules of programming discourse have an impact on program comprehension, with the strong expectations of expert programmers and the subsequent drop in performance when these expectations are broken by rule violations. Later work [Solo88] also supported the use of plans in comprehension, with shallow reasoning occurring in plan like programs. Here, plans are matched to code without significant reasoning about relationships within the code. In unplan-like programs they suggest deep reasoning is employed, which involves reasoning casually about the goals of the program and how they relate to the code of the program.

### 2.3.5 Pennington

Pennington [Penn87a, Penn87b] describes a bottom-up comprehension process, which develops two different mental representations.

- Program model. Pennington suggests that when new code is presented to a programmer they initially try to build a control flow model. This is built up using beacons in a bottom-up manner. Text structures e.g. control primes such as loops and plans are used in the program model development, as microstructures are chunked together to give macrostructures. Cross-referencing is used to link knowledge.
- Situation model. This is also built bottom-up and it uses real world knowledge of the domain to represent the code in terms of real world objects. This model is built using cross-referencing and chunking.

Cross-referencing is used to link the mental representations, allowing mappings from statement level representations, to a functional abstract program view.

### 2.3.6 Littman et al.

Littman et al. [Litt86] investigated program comprehension through experiments with expert programmers implementing an enhancement to a small existing system. Their studies suggest that there are two basic approaches to program comprehension: the *systematic strategy* and the *as-needed strategy*.

- **Systematic strategy.** Here the maintainer examines the program in depth, performing extensive symbolic execution on the control and data flow paths. Using this study of the dynamic and static aspects of the program, the casual interactions between components of the program are understood. This knowledge allows the maintainer to take these interactions into account when modifying the program. With the systematic strategy the aim is to understand the program before modifying it.
- **As-needed strategy.** With this approach, the maintainer attempts to minimise the study of the program. This is done by localising the parts of the program that need to be modified, in order to make the change. When the change is made the maintainer will then typically be required to investigate further on an as-needed basis, to gather additional information to make the change. This can be problematic as the maintainer may not understand sufficiently the casual interactions and are therefore unlikely to detect any unforeseen side effects of their changes.

The results of the study also suggest that the approach a maintainer uses to study a program heavily influences the knowledge they acquire about the program. This knowledge then directly determines if they can successfully make the change to the program. However, the applicability of each approach is dependent on the system size, as it is unfeasible to use the systematic strategy on larger programs although it may still be possible to subdivide the program into sections, which can then be tackled with the systematic strategy. Larger programs cause problems for both strategies, due to the vast amount of information involved.

### 2.3.7 Letovsky

Letovsky [Leto86] performed an empirical study of the cognitive processes in program comprehension. This was done by encouraging the maintainers to "think aloud" whilst trying to add an enhancement to an unfamiliar piece of code. These responses were classified into questions and conjectures. Letovsky [Leto86] suggests a cognitive model with three main components: a knowledge base, a mental model and an assimilation process. The knowledge base contains all the maintainer's programming expertise, domain knowledge, goals, plans and rules of programming discourse. The mental model is split into three layers: specification, implementation and annotation layers. The specification layer contains the program goals and is the highest level of abstraction. The implementation layer is the description of the program's actions and data structures and therefore is the lowest level of abstraction. The final annotation layer represents a mapping between corresponding parts of the code and goals. The assimilation process uses any available information from the program code or the maintainer's knowledge base to construct the mental model of the program. Letovsky believes that this assimilation process is opportunistic, with the maintainer using either a top-down or bottom-up approach, depending on the situation. They use whichever they believe will give them the highest knowledge gain.

## 2.3.8 Von Mayrhauser and Vans

Von Mayrhauser and Vans [Mayr95] offer a summary of these approaches and define an integrated metamodel which incorporates aspects of Soloway, Adelson and Enrich's [Solo88] top-down model with Pennington's [Penn87a, Penn87b] program and situation models. The integrated metamodel has four main components: the top-down (or domain) model; the situation model; the program model and the knowledge base. The knowledge base is used to build the other three model components, which are related to the comprehension process. Their experiments showed that programmers frequently switch between the three types of comprehension model components defined in the integrated model. When a programming language or code is familiar, a top-down (or domain) model approach may be used, for example if the programmer spots a beacon. This includes domain knowledge describing the program's functionality and this can be used for formulating hypotheses. An opportunistic or as-needed strategy is often used to develop the top-down model. When faced with unfamiliar code, the programmer may switch to developing a program model i.e. control flow abstractions. The situation model describes functional attractions and data flow within the program and unlike Pennington's model [Penn87a, Penn87b], they suggest that a situation model can be developed after only a partial program model has been formed, rather than the complete program model as suggested by Pennington. This was because they felt that developing a complete program model was unrealistic for large programs [Mayr93]. Structures built by a model component are also accessible to the other model components. Finally, a knowledge base stores the information needed to build up the other three model components.

Vans et al. [Vans99] also specifically investigated their integrated metamodel within program understanding behaviour during corrective maintenance of large scale software. A small study was undertaken observing four experienced professional programmers debugging software. They investigated how maintainers go about debugging software, looking at the work process and information needs. Their conclusions can be summarised as [Vans99]:

- **Actions:** Knowledge use and hypothesising are important actions at all levels of abstraction. Chunking and knowledge storage are common at lower levels.
- **Process:** Comprehension will occur at lower levels when there is little experience in the domain, until enough domain experience allows connections to be made at higher levels. When there is little knowledge of the software but the maintainer has domain knowledge, then comprehension will again occur at low levels but use direct connections to the domain model. Knowledge of both the domain and software allows connections between all levels of abstraction.
- **Information needs:** Connected program, situation and domain knowledge are important during corrective maintenance, along with domain concepts. Existing tools do not normally support domain concepts and connect model information, meaning that anything above the program model has to be searched for and connected manually.
- **Hypotheses** are made at code, algorithm and application domain levels, which suggests that the software must be understood at all levels of abstraction.

The small sample size of the study means that these conclusions are working hypotheses, however they do support the integrated model. They suggest the need for knowledge support at multiple levels of abstraction and mechanisms to allow tool support for working at different abstraction levels and ease of changing between abstraction levels.

## **2.4 Summary**

A number of program comprehension theories exist and commonalities exist between them, as highlighted by Von Mayrhauser and Vans [Mayr95]. All the theories define a cognitive model and they all use existing knowledge, combined with a comprehension strategy in order to acquire new knowledge and understanding of the program. The most pertinent approach will of course depend on the task at hand and the experience of the maintainer with the program, domain and implementation language. All maintenance activities require some understanding of the program and this is typically from the source code, due to the common problems of poor, inaccurate, or even missing documentation. Therefore, program comprehension is a major task and could thus benefit from tool support. This could take the form of tools for increased cross-referencing between source code and documentation, or improved methods and support for documentation maintenance. Visualisation offers a way to help maintainers construct their mental models by helping to abstract out the semantic constructs which they use in its construction. Visualisation allows them to easily explore the large amount of data that the source code contains, in order to aid in verification of hypotheses.

# Chapter 3 Software Visualisation

### 3.1 Introduction

This chapter introduces software visualisation. Various definitions of software visualisation are presented in order to show the variations within the field. The main taxonomies are then summarised to allow the context of this work to be seen. An overview of current visualisation systems is then presented, which provides examples of the different application areas of software visualisation, as well as examples of some of the representations that have been used within the field. Issues and challenges that are faced by all software visualisations are discussed.

### 3.2 Definition

Visualisation is defined as the "*process of forming a mental picture or vision of something not actually present to the sight*" [OED]. This generic English language definition specifies the goal of visualisation, that is the forming of a mental picture of some phenomenon. However, the application of visualisation techniques not only attempts to provide a mental image, but also improve the understanding of the item under study. The formation of a mental picture is the important feature and this does not have to occur purely through visual means, for example techniques using sound have been tried in order to aid program comprehension [Baec97]. However, most software visualisation systems focus entirely on using visual stimulus to present information on the software.

The visualisation field can be subdivided into a number of distinct areas, of which software visualisation is one. Software visualisation is therefore the application of visualisation techniques with software as the item under study. Many authors have provided their own definitions of software visualisation based on the idea of forming a mental image of software. For instance, Price et al. define software visualisation as "*the use of the crafts of typography, graphic design, animation and cinematography with modern human-computer interaction and computer graphics technology to facilitate both the human understanding and effective use of computer software*" [Pric93]. However, Knight goes further and includes the goal of reducing complexity, by defining software visualisation as "*a discipline that makes use of various forms of imagery to provide insight and understanding and to reduce complexity of the existing software system under consideration.*" [Knig99a]

To confuse matters there exist a number of related terms that are often used instead of software visualisation. Program visualisation is a commonly used term especially for the early taxonomies. For example: "*Program visualisation uses graphics to illustrate some aspect of the program or its run-time execution, where the program is specified in a conventional, textual manner*" [Myer90] or "*program visualisation as a mapping from programs to graphical representations*" [Roma93]. However, the term software visualisation shall be used for this work as it encompasses all aspect of a piece of software, rather than just the code and executable properties that the term program visualisation can suggest. The other terms used can be briefly summarised as:

- Computation visualisation is introduced by Stasko and "*is the use of computer graphics to explain, illustrate and show how computer hardware and software function*" [Stas92b].

- Algorithm animation or algorithm visualisation shows the behaviour and abstract operation of an algorithm.
- Code visualisation focuses on displaying the source code and its attributes.
- Data visualisation illustrates the data structures and values used in the program.

Visual programming is also often confused with software visualisation. However, these are distinct areas, with visual programming being the use of graphics to allow program code to be specified and developed. Whereas software visualisation is based upon aiding the understanding of programs that have already been written.

### **3.3 The Need for Visualisation**

Understanding existing programs is a significant overhead in the software maintenance process. The majority of time used by maintenance, debugging and code re-use processes is spent on understanding existing programs [Stor97a]. Here, visualisation can be beneficial by aiding maintainers, allowing them to interact with large volumes of data, in a fast and effective manner and in an attempt to discover hidden characteristics and patterns. Visualisation offers a way to cope with the massive information overload that can occur with traditional techniques, such as simple code browsing. Many authors see it as a way to interact with programs and indeed computers in general, in a more natural way. For example, Walker states *“the traditional interface of mouse, keyboard and screens of text allows us to work on computers, while techniques such as visualisation will truly enable us to work with computers”* [Walk95].

Program comprehension is a major component of any software maintenance task, occupying 50-90% of the maintenance time according to some estimates [Stan84]. Therefore, any improvements in comprehension activities, due to the use of software visualisation, will have a large impact on improving maintenance activities. This is combined with the growing size and complexity of software, as systems are required to perform more and more functionality and are increasingly interconnected. Modern software is also moving towards more rapid development and deployment and in this environment of high paced change, tools are needed to allow software to be understood quickly and reliably. Software visualisation offers a solution for this problem and can help developers keep pace with the rapid change as it is no longer possible for users to maintain a detailed mental model of the software they are working on, unless all their tasks are highly localised.

Software visualisation offers a chance for developers to see an otherwise invisible item. However, Brooks defines software as *“invisible and unvisualizable”* [Broo87]. He highlights the difficulty in visualising software and argues that one sees only one dimension of the software through the different views, yet superimposing these views together makes it difficult to extract any global overview. It may be true that each view of an aspect of the software, such as control flow or variable cross-referencing, presents only a one dimensional view and simply adding the views together results in a lack of global overview. However, software visualisation does not aim to present a single picture that allows an entire piece of software to be understood in all its intricacies. It instead aims to aid the understanding of software by humans by presenting details on the software in a more easily understood form. As Myers [Myer90]

highlights the “*The human visual system and human visual information processing are clearly optimized for multi-dimensional data. Computer programs, however, are conventionally presented in a one-dimensional textual form, not utilizing the full power of the brain*”. Software visualisation has moved on since Brooks’ statement. It still remains a major challenge for which there are no easy answers, however software is no longer represented as simple flow charts and graphs. Visualisation offers a way for humans to utilise their natural visual skills using techniques such as pattern matching and by allowing the greater depth of information that graphical representations can present over textual representations. Software visualisation offers views of the different facets of software and overviews can be presented for some information, such as higher-level design patterns or overview graphs. It must be remembered that understanding a piece of software from simply reading its source code presents a huge effort and is problematic. Therefore, any help that visualisation can provide is beneficial, even if this still requires significant effort in acquiring the understanding, provided that it is quicker or more reliable than traditional approaches. Software visualisation is often judged as failing, because the users do not suddenly understand the software under study when they use it. However, despite their desire of instant understanding from software visualisation systems, they are still willing to invest a significant amount of effort in studying the code using traditional methods. Software visualisation also offers the advantage of surprising users and making them think about the software, even if they have existing knowledge of it. This questioning of the users mental model can help them build a more detailed and reliable model and highlight anomalies in the data, such as redundant code or overly complex sections that they had not considered. These things can often be overlooked when simply reading the source code. For instance, De Pauw et al. state that “*animated visual displays let users assimilate information rapidly and help them identify trends and anomalies*” [DePa97].

### **3.4 Existing Taxonomies**

There have been a number of taxonomies on software visualisation. The most notable are those by Myers [Myer90], Price et al. [Pric93] and Roman and Cox [Roma93]. They aim to classify the types of software visualisation tools and a number of similarities exist between the taxonomies particularly between Price et al. and Roman and Cox. This section aims to give a summary of these taxonomies whilst comparing and contrasting their differences. This will allow some of the variations in approaches to software visualisation to be seen, as well as allowing this research to be placed in context.

#### **3.4.1 Myers**

Myers [Myer90] provides one of the earliest taxonomies on software visualisation and clearly separates visual programming from program visualisation. Stating that program visualisation systems “*try to make programs more understandable by using graphics to illustrate the programs after they have been created*” [Myer90]. Myers classifies software visualisation systems into:

- Static code visualisation
- Dynamic code visualisation
- Static data visualisation
- Dynamic data visualisation



- Static algorithm visualisation
- Dynamic algorithm visualisation

This can be represented as two axes: area of code being visualised (code, data, algorithm) and the program state during visualisation (static or dynamic). Systems can belong to multiple categories within this classification.

### 3.4.2 Price et al.

Price et al. [Pric93] define a taxonomy of software visualisation with six distinct categories. The six main categories are then sub divided in minor categories, which may themselves be sub divided. These categories are structured hierarchically, to allow the taxonomy to be extended and revised as software visualisation develops, with the easy addition of new categories or sub categories. The six main categories are:

- A. Scope: The range of programs that can be input and visualised by the program.
- B. Content: The subset of information from Scope that is actually used in the visualisation.
- C. Form: The parameters and limitations that govern the output.
- D. Method: How the visualisation is specified.
- E. Interaction: Characterises the system interaction methods.
- F. Effectiveness: Does the system meet its objectives?

Each of these categories can then be summarised:

#### **A: Scope**

The scope is the range of programs that can be input and visualised by the program. Price et al. see a division in this, into generality and scalability. The generality of the visualisation is the range of programs it can visualise based on hardware, operating system, language and application type. The scalability of the program is classified in terms of the largest program it can handle in terms of program and data size. This idea of scalability is purely the fundamental limit and not related to the effectiveness of the visualisation, which is assessed in section F of the taxonomy.

#### **B: Content**

The content is the subset of information from Scope that is actually used in the visualisation. The authors split this into a number of categories, namely: Program, Algorithm, Fidelity and Completeness and Data Gathering Time. The program subsection classifies on the amount of the implemented program that is visualised in terms of its code and data and their flows. The algorithm subsection looks at the amount of the “higher level” algorithm(s) that is visualised, again in terms of its instructions and data. The fidelity and completeness category classifies on the extent to which a true and complete picture of the system is presented by the visualisation and whether the visualisation modifies the behaviour of the program under study. Finally, the data gathering time categorises visualisations on when the data on the program is

gathered. If this is at runtime then the details of the mapping of program time to visualisation time is also used in the classification.

### **C: Form**

The form is the parameters and limitations that govern the output of the visualisation system. This is divided into a number of sections classifying on: the medium used for the visualisation; the presentation styles used in terms of colour, dimensions, animation and sound; the granularity of the visualisation in terms of at what level is the program shown; the use of multiple views; and the abilities of the visualisation to synchronise and view multiple programs at the same time.

### **D: Method**

The method is how the visualisation is specified. This is divided into two areas, how the visualiser specifies the visualisation and how the visualisation system and the source code are connected.

### **E: Interaction**

The interaction section characterises the system interaction methods. Price et al. [Pric93] identified three main facets to this, which they say fundamentally affect the design of the visualisation system. These facets are: Style (how the user gives commands to the systems); Navigation (how can the user navigate the visualisation and hide information of no interest); and scripting facilities (does the visualisation allow the interactions to be recorded or scripted and viewed at a later date).

### **F: Effectiveness**

The effectiveness section investigates how well the system meets its objectives and how well it communicates information to the user. This section is highly subjective and the authors split it into the following sections; purpose (What purpose is the system suited for? This is needed to see how effective it is at achieving its intended purpose); appropriateness and clarity (How well do automatic visualisation communicate information?); empirical evaluation (To what extent has the system been evaluated experimentally?); and production use (To what extent are people using the system?).

## **3.4.3 Roman and Cox**

Roman and Cox see “*program visualisation as a mapping from programs to graphical representations*” [Roma93]. They classify program visualisation into five main categories:

1. Scope. What aspect of the program (code, data, control and execution behaviour) is to be visualised? Visualisation systems often limit their scope to a subset of these program aspects.
2. Abstraction. What level of information presentation is supported by the visualisation? The taxonomy distinguishes three levels of abstraction, though the boundaries between them are blurred.
  - Direct representation. Some aspect of the program is mapped directly to a picture giving the most basic graphical representation. For example, a flow chart may represent control flow, or an array can be colour coded to show the magnitude of the stored values. These direct representations have the advantage of being easy to produce automatically (without needing

- programmers intent) and are easy to relate to the program. However, the lack of any real abstraction can lead to excessive amounts of visual information for large data sets.
- Structural representation. Here greater abstraction is used by highlighting “important” information, or alternatively by encapsulating or concealing information and using a direct representation for the rest. For example, encapsulating the bodies of classes and their member functions in the representation.
  - Synthesised representation. These representations show information that is not explicitly represented in the program, but can be derived from it. For example, there may be higher level abstractions of an algorithm, which are not explicitly contained in the program.
3. Specification Method. How is the visualisation specified? This can be by a number of methods:
- Predefinition. The mapping is highly constrained or fixed. While this constrains the visualisation it has the advantage of speed and allows automatic generation.
  - Annotation. The input program is augmented with calls to the visualisation system typically at the point of “interesting events” which pass in the required program state and cause the visualisation to be updated. This method has the major disadvantage of having to modify the input code.
  - Declaration. A mapping is specified between the program state and the visualisation so that changes in state are immediately reflected in the image. For example a variable mapped to an attribute of a visual object.
  - Manipulation. Visualisations are specified through the use of examples, which the system tries to capture and link to a program event.
4. Interface. This category focus on what the user sees and how they can interact with the system. This is split into two sub-categories:
- Graphical Vocabulary. This specifies the types of graphical objects and their operations as supported by the system in the construction of the visualisation.
  - Interaction. How does the user control the system?
5. Presentation. How the system conveys information through the visualisation.
- Interpretation of graphics. How is the visualisation understood and explained?
  - Analytical presentation. How is the analytical reasoning of a program, rather than its mechanics presented. For example formal correctness properties.

### 3.4.4 Summary of Taxonomies

Price et al. [Pric93] believe that Myers' [Myer90] taxonomy, while being a good starting point, is not detailed enough due to the variety of systems, goals and techniques available. Myers' axes do define some of the most important aspects of a software visualisation system, however they miss other attributes that are important for distinguishing systems. Price et al. and Roman and Cox's taxonomies offer a more detailed set of attributes to distinguish systems and some parallels can be drawn between these two taxonomies. Many themes appear in both taxonomies, but under slightly different classifications, for

example visualisation specification (Specification method category in Roman and Cox and in the Method category in Price et al.). However, these taxonomies, while being broad, are not particularly well suited to some aspects of 3D visualisation due to the differences introduced by the extra dimension on areas such as navigation.

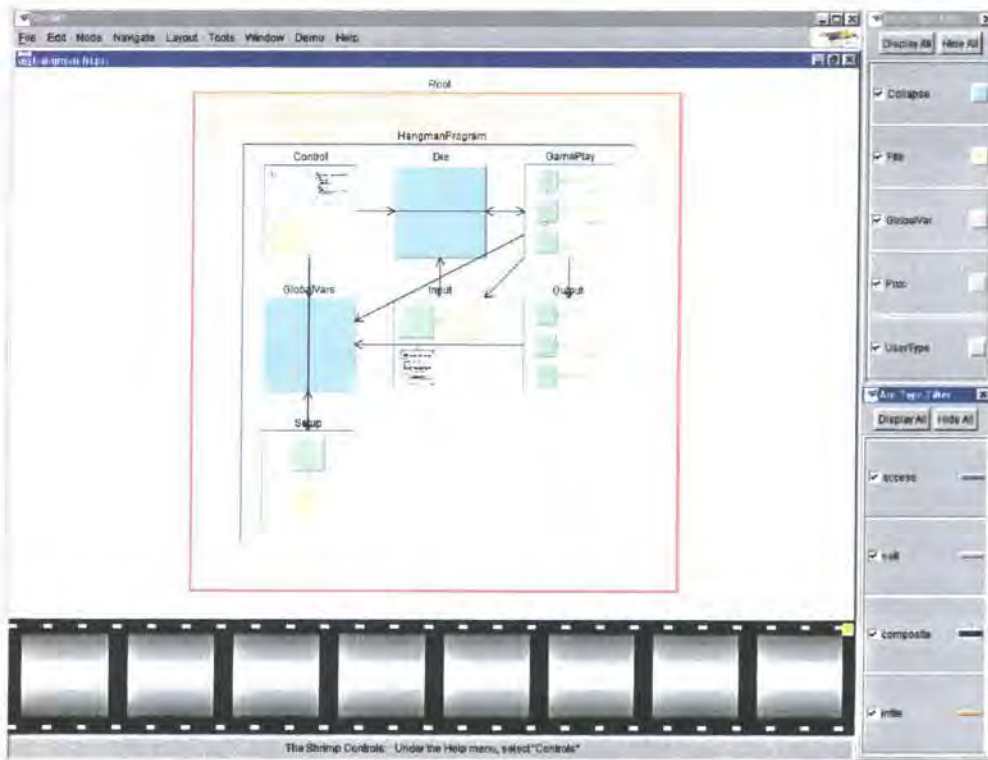
The taxonomies all use a number of example systems to illustrate their classification method and this also showed areas that many visualisation systems have failed to exploit. For example, Price et al. categorisation showed that relatively few systems made extensive use of colour in visualisations. Things have moved on since the taxonomy was completed with the increase in computer power, although colour is still not always used to its maximum effect. Price et al. also noted that intelligence is sorely lacking among automatic software visualisation systems and visualisation systems suffer from a lack of empirical evaluation. This is mainly due to the lack of methods to reliably compare techniques, due to the poor state of the art in software psychology. However, such evaluation is needed as it would show the effectiveness of a system and help to guide research efforts and system developments. Much of the evaluation done on systems at present, if any is done, is informal. Myers also identifies the need for experimental results for visualisation systems and mentions the problems with the scale of large programs that visualisation systems typically struggle with.

### ***3.5 An Overview of Current Software Visualisation Systems***

Software visualisation presents a breadth of different approaches, despite still being in its infancy. This section outlines a number of software visualisations, which are broadly categorised on the representations they use. The presented visualisations provide a glimpse of the different problem areas and representations that have been investigated within the field. However, the focus of this research is the visualisation of real software systems and not on algorithm animation systems, such as Tango [Stas90], Balsa [Brow85], Pavane [Roman92] and Eliot [Laht98] which all require some level of modification to the software and focus on small-scale sections of code. These systems focus on aiding a user's understanding by presenting them with an animation. This animation typically requires a specification to map program events to animation constructs. Therefore, these techniques do not allow a user to apply the visualisations to arbitrary programs of the supported programming language, and due to this, they are not presented in this summary.

The traditional focus of software visualisation has been the use of graphs to display information about programs, such as call graphs, control flow graphs and variable access graphs. These graphs can often be very closely mapped to the source code, for example, a control flow graph of the code. Therefore, they suffer from the same information overload issues as simply reading the source code itself. To alleviate these issues, there has been work on reducing the complexity of such graphs through filtering and clustering, as well as using hierarchical graphs and fish eye techniques. One such example is SHriMP (Simple Hierarchical Multi-Perspective) [Stor97b, Stor97c]. It uses a nested graph representation to present the structure of the software. The graph has composite nodes, which contain other nodes. This therefore provides the hierarchical structure. The composite nodes are typically used to represent software subsystems. The graph may be laid out in SHriMP using a grid, tree, spring or Sugiyama layout. The system uses fisheye and pan and zoom navigation techniques in order to try and retain context and reduce

user disorientation. The fish eye technique has a distorting optical effect so that objects in the centre of the view are larger than surrounding objects, thus providing focus and context. SHriMP's fisheye presentation can handle multiple focal points, in order to allow several subsystems to be examined at once. The source code of the program is represented in the graph by being embedded in the lowest level nodes. Figure 3-1 shows an example of the SHriMP system in use, with some nodes in the graph being expanded to show more detail.



**Figure 3-1 The SHriMP System**

Some systems focus on presenting queries about the software's structure, rather than presenting the whole system. For example, Richner and Ducasse [Rich99] present an approach that uses tailorable views of object-oriented software, based on both static and dynamic information. This approach allows users to specify the views of interest, which are then displayed as a graph using the dot framework [dot].

Some 2D representations use augmented graph representations, such as work by Lanza and Ducasse [Lanz01] on visualising classes using their Class Blueprint. This is a visualisation of one or many classes and it focuses on their internal structure. The class blueprint separates a class into a number of layers, namely initialisation, interface implementation, accessor and attribute. The first layer (constructor) contains methods that create and initialise objects of the class. The interface layer contains methods that provide access to the functionality of the class (e.g. declared public or protected in Java or C++), though accessor methods (provide access to attributes) are placed in their own layer. The implementation layer includes methods that are used by the class itself or are declared private. Finally, the attribute layer contains the attributes of the class. The items (methods or attributes) in each layer are represented as rectangles where the width and height can be mapped to a number of metrics, such as the number of lines of code in the method. The layers are positioned horizontally and the items in each layer are linked by edges that represent relationships between the methods and attributes, such as method A calls method B.

The invocation sequence is also laid out horizontally. The items and the edges representing their relationships are colour-coded to show types. This visualisation can be used to view multiple classes at once, for example, those in an inheritance hierarchy. Here, edges between the class blueprints represent inheritance relationships, whilst edges between items within the class blueprints represent relationships between methods and attributes in the class hierarchy. This separation of methods dependent on their role allows a class to be broken down, thus allowing the user to focus their search and compare classes more easily.

Whilst traditional software visualisations focus on the use of 2D representations, not all are based around graphs. Many other representations have been investigated and one notable example is that of SeeSoft. SeeSoft [Eick92] is a system which represents source code lines as rows of pixels. These rows are colour coded to display information such as modification date or scope level. This provides an overview of the source code and a source code browser shows the source code at a selected point in detail. For instance, Figure 3-2 shows a SeeSoft representation of 4000 lines of code, with the colouring of a line representing its age from old (blue) to new (red). The main ideas are:

- Reduces the representation by displaying files as columns and lines of code as thin pixel rows.
- Lines are coloured by statistic, for example, code age.
- Gives both an overview and detailed view with the capability to read the actual code

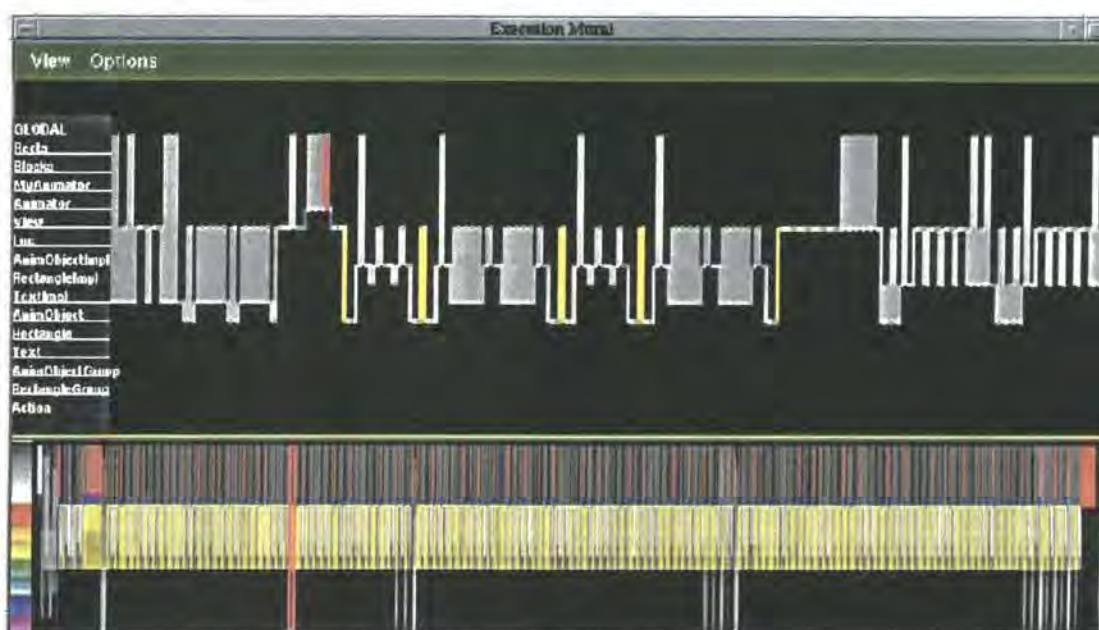
The method has been used to display a number of source code metrics, from code age to Y2K impact and changes to COBOL code [Burk98].



**Figure 3-2 SeeSoft [Eick92] showing 4000 lines of code. The oldest lines are in blue and the newest lines in red. Image from Stephen Eick's web page <http://www.bell-labs.com/user/eick/SoftwareVis.html>**

SeeSoft provides an intuitive visualisation due to the direct link between the representation and the underlying information. This is one software visualisation that can easily be applied to other text

documents and is not restricted to source code visualisation. Not all software visualisations are explicitly designed for software visualisation and there has been some cross over with the information visualisation community. For example, Jerding and Stasko [Jerd96a] define an Information Mural that allows a 2D reduced representation of a large information space to be presented at once. This representation is applied to numerous information types [Jerd96a], including object-oriented program method traces, sun spot records, river flow and automobile data. The aim of Information Mural is to allow large information spaces to be presented in their entirety, even when there is more information than available pixels on the display. It uses a pixel based technique that uses visual attributes such as greyscale shading, colour, intensity and the pixel size along with anti-alias compression techniques. Figure 3-3 demonstrates an application of the technique to visualising message patterns in object-oriented programs [Jerd96a] [Jerd96b]. In Figure 3-3 the bottom of the figure shows the entire execution trace using an Information Mural. The top of Figure 3-3 shows a zoomed in section of the trace. In both views the classes are assigned rows across the trace and a message between two classes is represented as a vertical line between the two class rows. The horizontal axis of the mural represents the sequence of the messages.

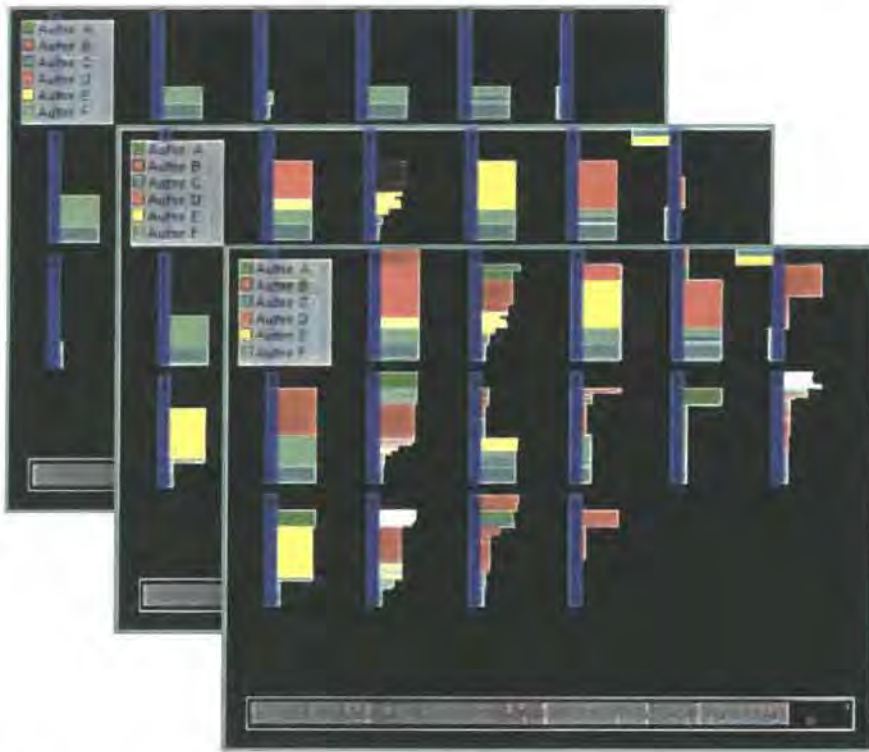


**Figure 3-3 The Execution Mural of an object oriented programs message trace [Jerd96a]. Image from: [http://www.cc.gatech.edu/gvu/softviz/infviz/information\\_mural.html](http://www.cc.gatech.edu/gvu/softviz/infviz/information_mural.html)**

This work is expanded upon for object-oriented traces through the addition of pattern extraction facilities for regular patterns in the message trace [Jerd96b]. The Information Mural approach is also used in the ISVis tool [ISVis], which allows a user to browse scenarios in a program's execution trace. Here, the mural is used to show the location of the scenario in the trace and to act as a navigation method.

Work by Taylor and Munro [Tayl02] has focused on the use of 2D representations for the visualisation of source code version data. Their Revision Tower approach [Tayl02] is based on tower structures where each level represents a version in the software. The left side of the tower structure shows header file changes whilst the right side of the tower represents changes in the implementation file (e.g. \*.c). Figure 3-4 demonstrates an example of the Revision Tower approach showing three stages of an animation. This

visualisation is designed to use animation as a core element in order to allow the changes over time to be seen. Therefore, the user can observe the picture being built up rather than being presented with the information all at once. This makes it easier to observe temporal patterns within the data.



**Figure 3-4 Revision Tower representation of source code version information [Tayl02]**

Moving away from the traditional 2D approaches has been experimentation with 3D representations. These approaches can be categorised into two separate strands; namely those based on using 3D graphs as the representation and those where more novel representations have been tried. Some of the first visualisations to use a 3D representation simply moved the traditional 2D graph structures into a 3D space. This was done with the rationale of being able to present more information and in a manner that can be understood more easily. This approach has been adopted by a number of systems such as Imagix 4D [Imag], NV3D [Park98] and Zebedee [Youn99]. Some of these systems allow graph abstraction mechanisms, such as sub graphs being contained within nodes, in order to simplify the graph structure. These 3D graphs have been used to represent a wide range of information on software, including method control flow [Imag], function calls [Youn99], metrics-based visualisation of large OO programs [Lewe02], UML diagrams [Imag][Gogo99][Dweye01] and software architecture [Feij98]. For example, Figure 3-5 shows a small UML class diagram in 3D [Dweye01], the purple spheres define package structures, whilst the nodes are classes.





**Figure 3-5 3D representation of UML [Dwyer01]. Image available at: <http://www.wilmascope.org/>**

Many systems support the use of 3D representations by claiming gains in understanding and in the ability to present greater amounts of information. For example, work by Ware and Franck [Ware96]. However, not all applications of 3D graphs have been claimed as a success. Young [Youn99] highlights issues in his Zebebee visualisation, commenting that the different view points result in an altered view of the graph, which could make it difficult to recognise, and that the crossing of the edge lines is also dependent on the viewpoint. Therefore, this area still offers potential, but as with any graph representation, the layout is a major challenge. The use of 3D also introduces navigation issues within the 3D space that the visualisation must address.

The use of novel and graph based 3D representations has been mixed in some visualisations. Work by Maletic et al. [Male01] defines a mixed approach where object-oriented systems are visualised. Their approach called Imsovision (IMmersive Software VISualizatION) uses a Virtual Environment and is designed to use a CAVE as the primary display. The CAVE is a virtual reality system, which is based on a room. Stereoscopic images are projected onto the room's walls and the user wears stereoscopic glasses, in order to see the images in 3D. The user is tracked within the room and the views are based on the user's current viewpoint allowing them walk around objects that appear in the centre of the room. However, Imsovision is based on VRML, so the visualisations can also be viewed on desktop PCs. Their visualisation represents object-oriented systems. The classes are represented as platforms, with the size depicting the number of methods and attributes of the class. The methods of a class are represented as columns that appear on the platforms. These columns are coloured according to the use of the method (constructor, accessor or modifier) and the size of the column represents the size of the method. Attributes are represented as spheres and they are also located on the class platforms. The interclass relationships (dependency and aggregation) are represented as edges between classes. Inheritance between classes is shown as the adjacency of the class platforms. This representation therefore uses a

graph as its underlying representation between the classes themselves. However, some 3D visualisations do not use a graph as the underlying representations. For instance, FileViz is a prototype system by Young for presenting a high level overview of a software system [Youn98,Youn99]. The visualisation is structured around the C source code files within a software system and the contents of those files. Files are represented as flat coloured pedestals as can be seen in Figure 3-6. The actual file type is shown by a glyph (an iconic representation in three dimensional space), for example a cylindrical glyph for a declaration file (\*.h). Colour is used to identify files with the same name e.g. test.c and test.h.



**Figure 3-6 FileViz showing an overview of files in a software system.**

The system uses the Viscape plugin<sup>1</sup> to provide the 3D visualisation within a WWW browser window. This is combined with two other HTML frames, one displaying information on the currently selected object in the 3D world and the other for showing the source code. Young's CallStax visualisation [Youn97][Youn99] is used to show dependencies between the various files. This allows file inclusion and library usage to be seen, and file details are also presented showing statistics, the functions defined and functions called within the file. The functions defined within a file are represented as blocks on the file pedestal and they are shown at two levels of detail depending on viewer's distance. Distance viewing shows low detail, with length and relative complexity presented, while the close up views shows greater detail. The system provides overviews of the files and functions defined. It uses Level Of Detail (LOD) to allow greater detail on the objects to be seen as the user moves closer to them, and so provides abstraction from overview to detail.

FileVis [Youn99] represents an abstract metaphor visualisation of the software. The use of abstract metaphors is the most common case within software visualisation. However, there has been some work done on the application of real world metaphors to software visualisation. For instance, Software World [Knig99b, Knig00] presents a 3D virtual environment approach to visualising Java source code. A real world metaphor is used in an attempt to deal with software of differing sizes in a coherent manner. The system aims to exploit the natural perception of users by providing a familiar environment. The following mapping (Table 3-1) is used from Java code to representation.

<sup>1</sup> Available at <http://www.superscape.com>

Visualisation Level	Code Level
World	Whole system, at very high level.
Country	Directory structure (packages in Java).
City	A single file from the current package.
District	A single class from the current file.
Building	Methods.
Inside Buildings	Lowest level allowing direct mapping to code.

**Table 3-1 Software World mapping [Knig00]**

The real world metaphor is further used, by having additional urban features such as parks to represent non-method file and class details. These can also act as navigation aids. Figure 3-7 provides an example of the visualisation at the district level with the building heights representing the method's length.



**Figure 3-7 A view of part of a city district in Software World [Knig00].**

The system uses the MAVERIK<sup>2</sup> (MANchester Virtual EnviRonment Interface Kernel) [Hubb99] to create the Software World virtual environment. This is a C toolkit for the development of single user Virtual Reality applications and can be compiled for multiple platforms. The visualisation is generated automatically through a four step process of:

1. Java code is parsed and extracted information stored in a database.
2. The database is queried to extract the information the user wants to visualise.
3. The extracted information is used to build the MAVERIK source files for the visualisation.
4. Generated C code is compiled to produce the executable visualisation.

The system demonstrates the use of real world metaphors, showing that they can be successfully used within the software visualisation field. However, Knight [Knig00] highlights that violating the logical framework of such metaphors will result in user disorientation.

<sup>2</sup> Developed at Manchester University and available for download at <http://aig.cs.man.ac.uk/>

FileVis and Software World demonstrate two novel approaches to software visualisation where the use of 3D representations has been investigated for none graph-based representations.

This section has attempted to provide an overview of software visualisation by highlighting some of the different approaches within the field. This is obviously not a definitive list and many other representations and visualisations exist.

### **3.6 Issues and Challenges**

Software visualisation offers a way to support program comprehension and therefore maintenance activities. However, this is not to say that this approach is not problematic and a number of challenges exist for software visualisation research. These general issues are faced by all software visualisation systems, regardless of which specific aspects of software they are representing. They can be summarised as.

- **Representation:** One of the main problems is that of representation. Software is made up of abstract constructs that have no geometric representation, therefore, there are no intuitive representations of such data. This contrasts to some other visualisation fields, where the data often maps to real world phenomena, such as flow visualisation. This abstract nature of software and the complexity of the relationships involved has lead some authors to suggest that software is unvisualisable [Broo87]. Representations need to be consistent and allow incremental change as the software is modified. Traditionally, node and arc based representations have been used to show all aspects of a piece of software. However, while it is an easily accessible representation to users, it suffers from problems of scale as the size and complexity of the program increases. Therefore, research into new representations is needed to enhance or replace the existing techniques.
- **Abstraction:** In order to handle the volume and complexity of software, abstraction is necessary. This offers challenges in finding visualisations that allow multiple levels of abstraction and provide coherent changes between them. This need for abstraction is supported by the program comprehension theories such as Von Mayrhauser and Vans integrated metamodel [Mayr95], where understanding occurs at different levels of abstraction. However, the extraction of meaningful abstractions can be difficult, and furthermore these abstractions must be supported within the representation which introduces effects on layout, presentation, navigation and interaction.
- **Scale:** Software is constantly growing in size and in order to be useful software visualisation systems must be able to handle the scale of real world problems. Typically “toy” programs are used to demonstrate visualisation systems, especially algorithm animation systems. Scale introduces many challenges, especially in terms of providing a representation that can scale up coherently and navigation methods to easily explore and locate information.
- **Interaction:** Not only must the data be represented in an understandable way, but it must also support ease of interaction. The user must be able to navigate the data easily and in a number of

forms dependent on their task. Support is needed for browsing and searching to allow rapid access to information. The interaction must be intuitive and consistent, so as not to increase the cognitive load on the user. The user should be thinking about the data and not having to think about how to use the interface to explore it. Research is needed into navigation techniques to aid user interaction.

- **Customisation:** Understanding is an individual process, dependent on person, task, time-scale and the information present. To support this, software visualisation tools need to provide customisation features to allow different users to tune the display to their information needs. For example, filtering information that is unnecessary for the current task and support for multiple views, as well as for the addition of new views.
- **Integration and Acceptance:** In order for software visualisation to be accepted and integrated into software lifecycle processes, software visualisation tools need to be designed specifically to support these processes. This means that the visualisations need to be generated automatically with very little or no intervention from the user. It must support the software as it occurs in its original form and not forcing it to use specific environments, libraries or language subsets. Many current systems have restrictions that prevent them from being used on real world programs. This is not to say that prototype systems should necessarily support all features. However, in order to show the usefulness of software visualisation to real world problems, systems are needed to demonstrate that the techniques are applicable to such problems and can be integrated into current environments.
- **Evaluation:** Currently, evaluation is a major problem in software visualisation with few real evaluation frameworks existing and very little evaluation occurring of prototype systems. Research into evaluation is needed to identify the contributions provided by systems and to direct future visualisation research. It would also benefit the acceptance of visualisation, by providing scientific results on its benefits allowing industry to make informed choices.

This is not a definitive list, but it highlights some of the main problems faced by the software visualisation field. These issues must be considered in the design of all software visualisations and are critical to the success of a visualisation. However, there may be trade-offs between different issues, for example the graph representation is widely accepted in the software engineering community and therefore a visualisation using this representation may gain more acceptance because of this, despite the scale issues of graph representations. Conversely, a visualisation based on more novel techniques such as an abstract three-dimensional representation may be better for large-scale problems, but face acceptance problems due to the unfamiliar nature of the representation.

### **3.7 Conclusions**

This chapter has presented a summary of software visualisation and discussed some of the definitions that exist for software visualisation, as well as highlighting related areas. The three main taxonomies on software visualisation were briefly summarised to give an overview of the field and to show how the subsequent work fits into the larger software visualisation arena. An overview of some of the existing

approaches and trends within software visualisation was then presented. Finally, some of the main issues to be faced by the software visualisation field were discussed, highlighting the need for further research within the field.

# **Chapter 4 Software Visualisation at Runtime**

## 4.1 Introduction

This chapter provides an overview of the visualisation of the runtime behaviour of software, and this is defined as ‘runtime visualisation’ for the purposes of this work. A number of approaches to runtime visualisation exist and these are presented here, some of the existing runtime visualisation tools are also summarised. Runtime visualisation introduces many issues and challenges and these are discussed in this chapter, the desirable features of runtime visualisation tool are then considered.

Runtime visualisation can show how a program executes in terms of both control flow and data flow. It can benefit program comprehension activities and therefore maintenance and debugging, by providing insights into how the software actually executes. Many program errors and attributes are only visible at runtime and as Lieberman and Fry indicate what “*makes programming cognitively difficult is that the programmer must imagine the dynamic process of execution while he or she is constructing the static description*” [Lieb95]. Traditionally, there has been little support for demonstrating to a programmer how the program executes. Few runtime visualisation tools exist and their acceptance is very low, therefore, traditional debugging techniques are often used to inspect the program. Nevertheless, despite developments in program development environments, debugging techniques are still fairly basic and debugging is often viewed as a second class activity. A common technique for observing how a program is executing is to simply add trace code to the existing program code to give an execution trace. For example, “*First, the bad news. Adding printf() calls to your code is still a state-of-the-art methodology*” [Geis94] is a common case. Even when debuggers are used, they have typically changed very little in overall design over the years. This makes understanding the execution, and therefore behaviour of a piece of software, a difficult task. The current state and shortcomings of runtime visualisation tools are discussed in this chapter.

## 4.2 Debugging Tools

A common approach by programmers when trying to understand the runtime behaviour of a piece of software is to use a debugging tool to inspect the program's structure at points in the execution. Java debuggers focus on the level of the virtual machine, which is in terms of threads and methods. This is, therefore, very similar to the information that is presented for a procedural language. Often the same debuggers are used across procedural and object-oriented languages, such as C and C++. Even for languages such as Java, the debugging interface has changed little. The debuggers show variables in terms of objects, and functions in terms of methods on those objects, however, it is difficult to get an impression of how the classes interact and the object-oriented nature of the program is often hidden. This view is also very different to that of the low level design views, such as UML class diagrams, which are commonly used by software engineers in the design and implementation of object-oriented software. Some debuggers have been expanded to have visual elements [Hans97][DDD]. The GNU Data Display Debugger is a prime example [DDD], with support for displaying data structures graphically and displaying values of arrays in a graph. However, there is no specific object-oriented support and as with all debuggers they are more suitable for investigating specific data structures or details, rather than for gaining an overview of the software.



Another common approach by some programmers is to place print statements in their code to allow them to see which parts of the code are being executed and to print out key values. Trace code such as this can be valuable when developing a system to record the progress of the program at key sections. However, in order to understand such trace code the programmer needs knowledge of their placement within the program and the relevance of the values being outputted. Also, such trace code output offers no support for aiding the programmer in understanding the structure of the program or its OO nature.

### **4.3 *Extracting runtime information***

The visualisation of the runtime behaviour of software requires runtime information to be extracted. Therefore, dynamic analysis of the program is needed. There are numerous methods of extracting dynamic information on a running program and these are summarised below.

One technique is to augment the code of the program to be analysed. The simplest approach to this is the hand coding of the visualisation by adding event notification or drawing code to the original source code. This is the method used by many of the Algorithm Animation systems, such as Tango[Stas90] and Balsa [Brow85]. These systems typically show the operation of particular algorithms, such as sorting routines. The hand coding of the visualisation allows the person doing the augmentation to encode higher level information and focus the representation to that specific algorithm. However, this approach is obviously unsuitable for real world application, due to the lack of automation and the huge investment needed to produce the visualisations. It is also very intrusive to the source code of the program under study, which can require significant additions.

An alternative technique is the declarative approach. Here, a set of graphic objects is supplied by the system. These objects are then used in the program to be visualised, for example, a binary tree data structure. As these structures change state, the visualisation is updated accordingly. Thus, there is mapping between computational objects and graphical objects. This has the advantage of separating the program code from the animation code, however, the original program may still need to be changed significantly. New graphical objects may also need to be defined, if there are no suitable ones present in the system. An example of this approach is the Eliot system [Laht98].

An approach to these restrictions is the automatic augmentation of the code. In this case, the source code is augmented using some automatic means, such as using a pre-processor to augment C/C++ code. The augmentation is not seen by the programmer, however, extra code is added and compiled into the final executable to capture events such as method calls. This additional code communicates these events to a visualisation tool, which then produces the visualisation. This approach has the advantage of being able to produce the visualisations with no user intervention. However, the source code must be recompiled in order for it to be visualised. A number of tools use this method, such as HotWire [Laff94] (pre-processor annotation) and Program Explorer [Lang95] (tool annotation). Standard compilers can also be modified to augment the code with tracing functionality. Languages such as Java do not have a pre-processing stage before compilation, however, Java offers another variation on this approach. Java loads the classes of the program dynamically using a class loader. This class loader can be modified to automatically augment the

byte-code as it is loaded. This has the advantage of not needing to modify the source code, or even have access to it.

The approaches discussed so far have all modified the code of the program in some way. However, this is not necessary in order to extract runtime information. Debuggers offer a way for software engineers to inspect the internal state of a program. Therefore, they can be used by visualisation tools to access runtime information. Some debuggers offer graphical representations of the program. A notable example is the GNU Data Display Debugger [DDD]. A visualisation tool can simply communicate through an existing debugger, such as `gdb` [`gdb`], by parsing the textual output and generating textual commands to request information. This approach has been used by systems such as DDD, however, it can suffer from performance issues of having to go through the textual conversion and it is highly dependent on consistency within the textual display. A more efficient solution is to use the underlying debugging mechanism, but to access it directly, either through a debugging interface, if one exists, or through the modification of an existing debugger. The use of debugging techniques means that the source code is not necessary and the source code does not need to be changed. It also allows information to be extracted on demand, unlike some augmentation methods where it must be decided which events to record, before the program is compiled. The only constraint with this method is that the code is compiled with debugging information included.

Code that is run on a virtual machine, such as Java, offers another avenue for extracting runtime information. In this case, the virtual machine can be used to record information on the program. This is the technique used by Jinsight [Jins], which uses a modified version of a virtual machine to record events to a trace file, this is then viewed in Jinsight.

#### **4.4 Online Vs. Offline Approaches**

Visualisation of runtime events can take either an online or offline approach. The offline approach is when information on runtime events, such as method entries, is extracted to some store, such as a trace file or database. Once the program has completed its execution, the stored information is then visualised. The online approach is when the visualisation is connected to the live program and is generated in real-time as events occur in the program under study. Each approach has its own advantages and disadvantages. The offline approach allows the trace file to be visualised many times for a single run of the program, which can be especially useful if the execution of the program is long, or places a large demand on resources. The use of a store of the execution also means that a complete record of the execution is present at the point of generating the visualisation, which can be used to provide a more optimal layout. The complete record of events also allows summary information to be extracted that could be used to give summary information and views within the visualisation. In addition, it allows the events to be easily replayed in reverse as well as forward. The separation between the extraction of the runtime information and the visualisation of that information allows the information to be processed in its whole, without the performance constraints of having to update the visualisation inline with the events. Thus, complex analysis can be performed without affecting the speed of interactivity of the visualisation. The offline approach can therefore have a performance advantage; when the program is running, events only need to be recorded and these can be processed before the visualisation is displayed. However, the offline

approach also has a number of disadvantages. The record of the execution can be substantial even for a small program, due to the large number of runtime events, such as method calls, that are present. The recording of events means that the runtime events of interest must be known before the program is executed. Problems can occur when a particular piece of information is desired at only one point in the program. For example, if the value of a field is wanted at a particular point, its value must be recorded for the whole trace. This can lead to a massive increase in the recorded information and a performance issue, if that value is modified frequently. In order to tackle this issue, some offline approaches allow the tracing to be switched on and off from within the program, however, this means that the source code for the program must be modified.

Online approaches are tightly coupled to the execution of the program under study. They allow the monitoring of different events to be changed on the fly as the user sees fit, for example, inspecting a field's value. This allows the visualisation to provide additional services, such as setting breakpoints on the use of a field for example. The visualisation acts on events as they occur which means that only events that make up the current state must be recorded and not necessarily all events to that point. However, this makes it more difficult to reverse some of the execution as the program is then not at the same point as the visualisation and items that are queried on demand, such as field values, will be unavailable. The tight coupling to the program under study also brings disadvantages. In order to view the visualisation, the program under study must be running which may not always be feasible or desirable, if the program is substantial or requires a particular set up that is unavailable. The visualisation can also be under greater performance constraints than offline approaches, as it must compete with the program under study for resources when they are running on the same machine. The visualisation must also try to "keep pace" with the program under study in terms of the events it shows. This makes the speed of generation of the visualisation an issue, especially as the complete record of the execution is never available as in offline approaches. This means that only the current state information can be used when generating the visualisation, so information on future states cannot be used, for example, to optimise the layout. Both approaches have been used by current runtime visualisation systems as the next section highlights.

## **4.5 Dynamic Software Visualisation Tools**

A number of runtime visualisation systems have been developed and this section aims to provide an overview of the different approaches tried within the field. The presented systems are not a definitive list but aim to demonstrate the main features that existing approaches display.

### **4.5.1 NestedVision3D (NV3D)**

NestedVision3D is a system by Parker et al. [Park98] for visualising large nested graphs in 3D. The paper [Park98] considers the problem of focus and context, where one wishes to provide detail on an area whilst providing the context of that detail, i.e. the bigger picture. The techniques presented for this are (for graphs):

- Distortion Techniques. The graph is distorted spatially to make the area of focus more prominent, for example, by giving more room in the graph and making other points less prominent. For example, the fish eye effects and the hyperbolic lens [Lamp95].
- Rapid Zooming Techniques. A large amount of information is present but only a small amount is visible at any one time. However, it is possible to quickly zoom in and out on a point of interest, giving smooth changes between context and focus, although both cannot be seen at the same time.
- Elision Techniques. Parts of the structure are hidden, typically using a collapse and expand idea, for example on sub-graphs. These elided structures then provide context whilst others can be expanded to provided detail.
- Multiple Windows. Separate windows are used to provide focus and context. However, visual cues are needed to show how the windows interrelate. E.g. which area of the overview is showed zoomed in another window.
- 3D Interactive Visualisation. Focus and context is provided through the use of perspective, with distant objects being less detailed than the foreground, especially if level of detail is used in the display. However, the context is very dependent on the layout and the orientation of the view leading to arbitrary context.

The authors argue the use of 3D over 2D, using Robertson et al. Cone Trees [Robe91] as an example where a 3D representation allows the viewing of larger structures than 2D.

NV3D is a 3D data visualisation tool for large relational information structures. The system uses the typical node and arc representation, with the nodes being colour coded or texture mapped according to type. The system uses elision with nodes able to represent sub-graphs that can be expanded or collapsed. This is key to the system and only through this is it able to deal with large data structures in real time. The graph can be explored using navigation widgets and rapid zooming and also non-spatially through queries and layout variations. The graph can be queried dynamically on relationship type from a node. A slider then controls the depth of this query from the start node. The layout can also be used to explore the graph by changing which relationship are used for layering the graph and relationships can be weighted according to importance.

The system allows the viewing of static data such as calling structures and object relations, however the system also supports dynamic behaviour, mainly to provide an outlet for viewing execution threads. The system uses a notion of snakes to visualise execution threads. These are animated heads and tails, where the head shape represents the type of action and the tail shows the recent history of the process as well as attracting the users attention. The viewpoint can also be attached to a snake to provide automatic tracing of the snake. In order to view the details of a snakes passage, for example the calling arguments, data probes can be attached to nodes which show the message when the snake arrives at the node. These work on a hierarchy so messages are passed up to any other probes allowing a sub hierarchy to be monitored by placing a probe at the top of the hierarchy. The notion of snakes provides a novel way to represent process execution. However, it would appear to suffer from the problem of scale, since it could be difficult to maintain a viewpoint that allows multiple snakes to be observed in a large graph.

## 4.5.2 Virtual Images: Interactive Visualisation of Distributed Object-Oriented Systems

Vion-Dury et al. [Vion94] present a system for the observation of objects in distributed object-oriented systems. They propose the notion of virtual images as a graphical representation whose main principles are a 3D spatial model where objects are represented by polyhedrons with significant orientation, size colour and shape. One of the more novel ideas presented is that of the shape of an object is representative of its name, in an attempt to allow object recognition without cluttering the display with names. An example of this is shown in Figure 4-1. This is combined with pop-up labels on mouse over to show the actual label of an object.

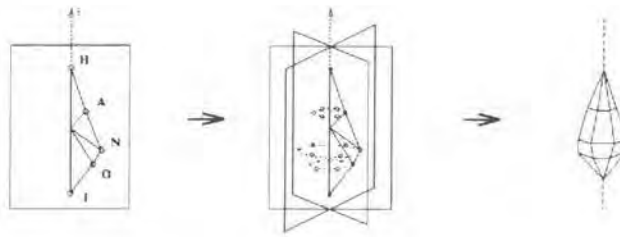


Figure 4-1 Objects name used to create representation [Vion94]

The system is implemented as a number of modules. Firstly, a debugging kernel allows the recording and replay of events and aims to solve the non-determinism of the parallelism and allow cyclic debugging. This kernel then passes information to a number of views using a bus architecture allowing for extensibility. These are then able to use a 3D graphical abstract machine to produce the virtual images. The system offers a number of views visualising the execution model and concurrency of an application.

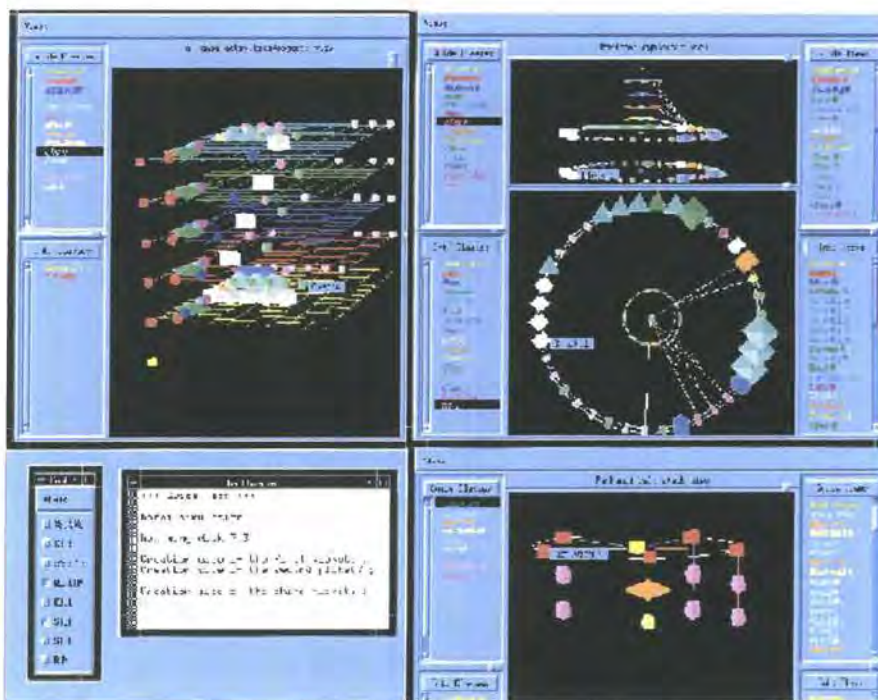


Figure 4-2 An overview of the system [Vion94]

The execution model (shown top right in Figure 4-2) is represented by showing the call graph of each activity and the memory mapping of objects. Activities are represented as pulsing spirals, which grow with each method call and have a segment pointing to the object being called. Thus recursion is easy to observe with segments going to the same object. The concurrent aspects are shown (shown top left in Figure 4-2) as a horizontal grid representing each activity and all objects linked in its context. Within this, the call graph is shown by an animated coloured line passing through objects on the grids. Object sharing is shown when the same instance is stored in multiple grids. The evolution of the stacks of program execution is shown in the activity stack view (shown bottom right in Figure 4-2). Here, the circle represents the application with the activities placed on the circle. Objects used by each activity are stacked above it in the order they are called.

### 4.5.3 VizBug++

VizBug++ [Jerd94] is a simple prototype for the visual debugging of C++ programs. It is based on previous work on GROOVE [Shil92], a visual design tool using the same visual paradigm. The view contains three basic entities: a tree structure representing class hierarchies, rectangular nodes representing global functions and circular nodes representing instances. Classes are represented as upside-down triangles, while arrows from the bottom of a class to the top of another class represent inheritance. The view shows the message passing that occurs during execution by drawing arrows between instances. From implementing this prototypical view, the authors found that the necessary information to construct a useful visualisation is difficult to gather and that view layout and information overload were major problems with the simple view. They suggest that multiple views with different levels of abstraction may be needed to present information in an organised and informative way [Jerd94].

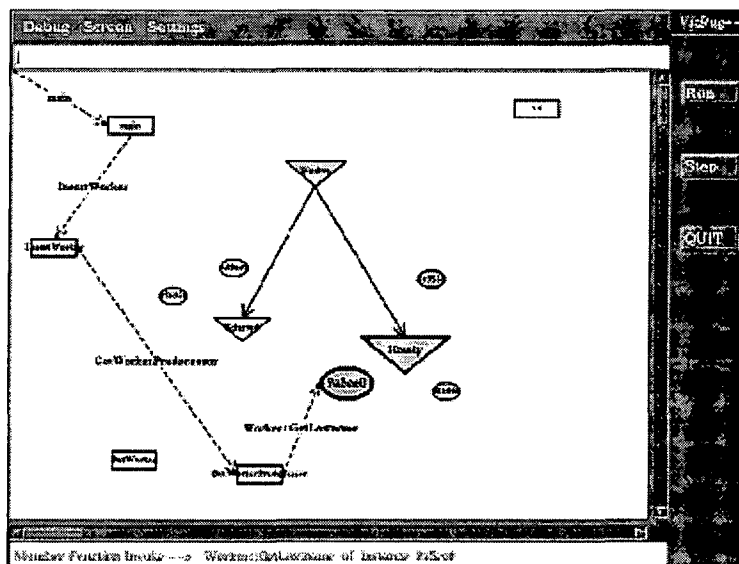


Figure 4-3 VizBug++ showing class and instance relations [Jerd94]

Through the development of VizBug++, Jerding and Stasko [Jerd94] define four main objectives that they suggest a visualisation of object-oriented software must fulfil.

- **Little or no programming intervention.** The visualisation should take no all little programmer overhead.

- **Present the "right" things.** The most important aspects of the software system should be presented.
- **Allow viewers to focus quickly.** The visualisation should support navigation so that users can focus on a particular concern.
- **Handle real world problems.** The visualisation must be applicable to large-scale systems.

These are presented in the context of visualising object-oriented systems, however they can be applied to the general software visualisation case. Oudshoorn et al. [Ouds96] refine the requirement of presenting the "right" things. Suggesting that:

- The system should provide feedback and allow the user to discover new information, not just confirming what they already know.
- The graphical representation must provide information and not just act as decoration.
- The visualisation should also allow users to focus quickly on areas of information and allow customisation over the views.

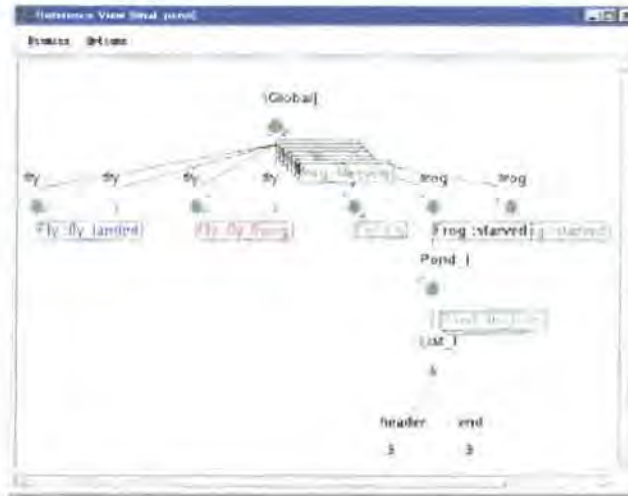
Oudshoorn et al. also state the need for:

- **Scalability.** The tool should be sufficiently scalable in terms of the problem sizes it can handle.
- **Extensibility.** The tool should be flexible to change. For example, allowing the modification or addition of views.
- **Portability.** The tool or its concepts implemented on one platform should not restrict porting to another platform.

#### 4.5.4 Look!

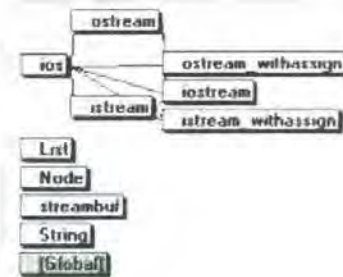
Look! is a runtime visualisation and debugging tool for C++ originally developed by Objectives Software Technology Ltd [Look]. It offers a number of views for displaying runtime information on C++ programs, including object creation and destruction, object relationships, inter-object communication and memory usage. The views allow graphical selection giving point and click access to source code, data and breakpoint setting.

The Reference View shows the reference relationships between objects in the system. It displays objects as they are created and destroyed, allowing pointer relationships and object interactions to be easily seen. All reference relationships to, or from, a selected object can be optionally shown to aid debugging. Threads in the application are visible and mapped to the objects that they use. The system also offers the ability to filter the view, for example by excluding objects of a particular class. Figure 4-4 shows an example of the reference view.



**Figure 4-4 Look! showing object references**

The Class View (Figure 4-5) provides information on the static structure of the program by displaying the class hierarchy. The view scrolls and illuminates the currently active class. The view also allows a quick way to navigate to the source code and to the instances that exist for the selected class. Clusters of classes can be created which allows the runtime operation of the program to be viewed at the architectural level, or to allow the user to focus on the specific application activity of these clusters. The Cluster View (Figure 4-5) enables the architectural level to be viewed by showing message interaction between clusters and the objects that exist in a particular cluster. This can be useful for design verification and behaviour checking at a higher level of abstraction.



**Figure 4-5 The cluster and class views respectively**

A Message View provides an active trace of function invocations, thus minimising the need for trace instructions (e.g. printf) in the code. Object creation is also shown in a Creation view and as with a standard debugger, a source code view is present.

The system offers a number of views and reporting facilities (coverage, hot spots, memory usage and leaks). As with a normal debugger, it operates on a version of the executable compiled with debug information, therefore having the advantage of no programmer intervention.



### 4.5.5 Jinsight

Jinsight [Jins] is a tool for visualising the execution of Java programs developed by IBM. A Java program to be analysed is run through a modified version of the Java Virtual Machine which is supplied with Jinsight and produces a trace file of the execution. User options allow the specification of which events to record. This trace file is then loaded into Jinsight to be analysed using an offline approach. The system offers a number of views and is designed with object-oriented and multithreaded programs in mind. An Execution View represents the programs threads as vertical "lanes" as shown in Figure 4-6. Time proceeds from top to bottom in the view and the execution stack for each thread is shown left to right down the lane. This is shown as coloured strips where the colour indicates the class of the invoked method. Relative timing between the thread activities can be seen by comparing events across the lanes.



Figure 4-6 Jinsight showing execution view.

A Histogram View shows the program resource usage in terms of classes.

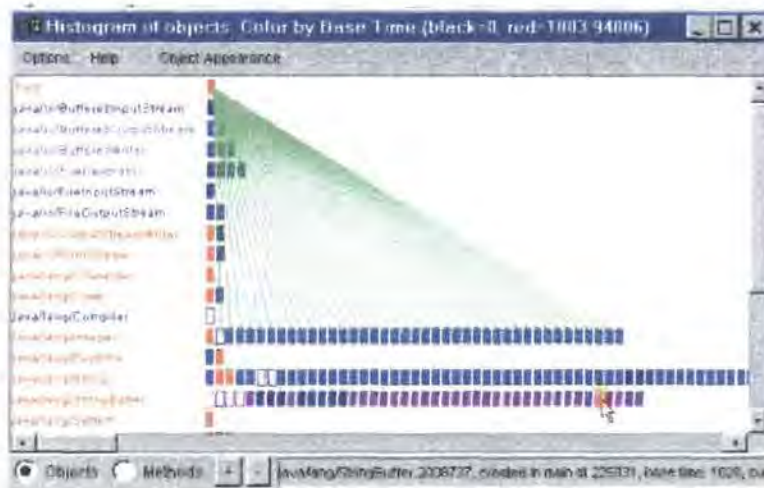


Figure 4-7 Histogram View showing object relations

Each row in Figure 4-7 shows the name of a class with the coloured rectangles representing the instances of the class. The colours used for the instances can represent various information depending on user choices, such as the number of calls, the amount of memory taken up and the time spent in methods of that instance or class. The rectangle turns into an outline when the object has been garbage-collected. Lines represent relationships among objects. For example, all of the method calls on objects of class

java.lang.Integer as in Figure 4-7. The lines can show how each object calls, creates, or refers to other objects. The view can also show the methods of each class rather than the instances.

The Invocation Browser displays all of the invocations of a selected method and uses the same graphical representation as the Execution View. This is expanded on by the Execution Pattern View, which automatically extracts patterns related to the given method, thus showing how the method typically executes and cases in which it diverges from the general pattern. This aims to overcome the massive numbers of innovations that may be present in the invocation browser.

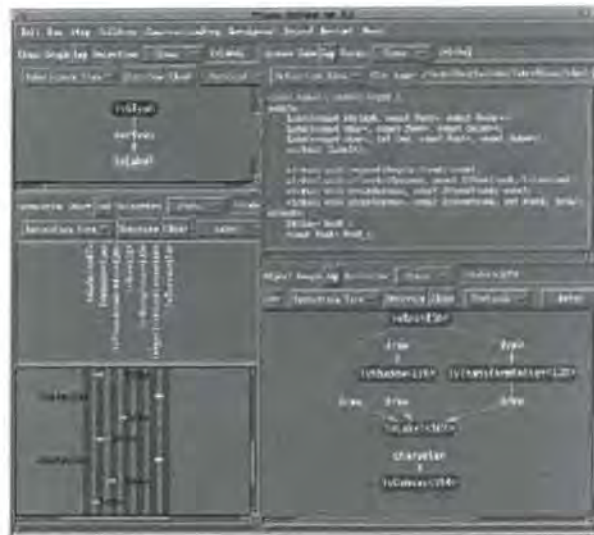


**Figure 4-8 Reference Pattern View**

A Reference Pattern View (Figure 4-8) shows object references. The view can be used to detect memory leaks, where objects that should be reclaimed aren't because they are still being referenced. Memory references can be recorded at two different program execution points allowing differences to be observed between the snapshots (old-generation objects and new generation objects).

### 4.5.6 Program Explorer

Program Explorer [Lang95] is a program visualiser for C++ developed by IBM research. It aims to provide class and object centred views of the structure and behaviour of large C++ systems and to allow programmers to maintain and re-use undocumented parts of such systems. This is tackled by using static and dynamic information on the program, with couplings from classes-to-objects and objects-to-classes. Classes-to-objects coupling is used to filter the dynamic information by using static information, for example selecting objects based on a class. Whilst objects-to-classes coupling uses the dynamic information to filter the static information, for example, providing information on relationships that are actually used, rather than all relationships



**Figure 4-9 Program Explorer showing multiple views [Lang95]**

Figure 4-9 shows the multiple views of the system, with the source code browser, inheritance view (top left), invocation chart (bottom left, showing object longevity (lengths of bars) and the creation order (from top to bottom)) and the object graph showing objects interactions (bottom right). The system uses two methods of information extraction. The static information is extracted from a program database generated by the compiler. This gives details of the classes and their inheritance relationships. The dynamic information is extracted by instrumenting the original C++ program, using the program database as a guide. The system allows the user to specify which classes should be instrumented, thus allowing trivial or very active classes to be ignored. The instrumentation captures events of object longevity (creation and destruction), function invocation (entry and exit) and variable access, which are stored by a Trace Recorder. The instrumented program is then executed under the control of the Program Explorer, allowing the program to be run or single stepped using typical debugger controls. This event recording technique has the advantage that when the program is halted, the system has a record of all the events up to that halt and not just the content of the call stack as would be available with a standard debugger.

### 4.5.7 HotWire

HotWire [Laff94] is a visual debugger for C++ developed by Laffra and Malhotra at IBM. The system works by annotating the C++ program using a special pre-processor. The instrumented program then makes calls into the runtime library of the system.



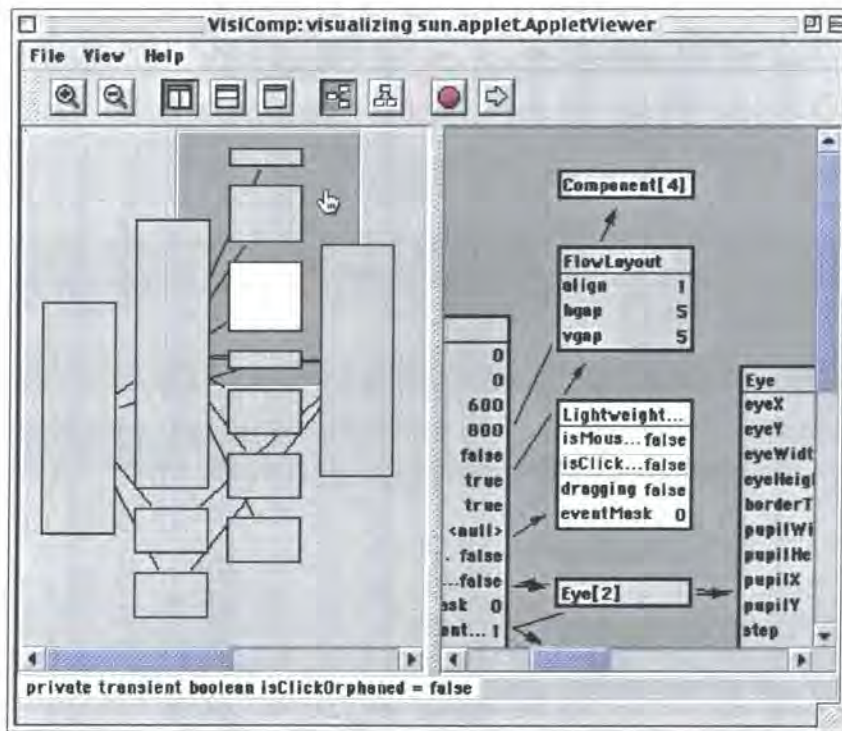


Figure 4-11 VisiVue™

The tool offers a simple mapping from the objects to the representation, thus making it easy to relate the resulting visualisations to the code. However, this also results in the complexity of the visualisation rising linearly with the complexity of the program. The tool is designed for educational use and for small-scale applications and therefore it would be of limited use in trying to aid the comprehension of a substantial program. The system does not offer support for user abstractions and objects with a large number of fields could be problematic using this representation, as the size of the node is directly related to the number of fields that an object encompasses.

## 4.6 The Benefits for Object-Oriented Software

The object-oriented (OO) programming paradigm introduces new language ideas and these affect its analysis and comprehension. Object-oriented software offers many advantages, however Jerding and Stasko [Jerd94] suggest it is "a double-edged sword". This is due to the discrepancies between the static class descriptions and runtime behaviour as networks of communicating objects [Gamm94] [DePa97]. For example, De Pauw et al. state that "There is a dichotomy between the code structure (static hierarchies of classes) and the execution structure (dynamic networks of communicating objects) of object-oriented programs. The programmer must understand and map between these structures, a significant burden even after the programmer is familiar with them." [DePa97]. It is due to this that De Pauw et al. state that "Insight into dynamic aspects is critical for understanding, tuning and debugging object-oriented software" [DePa97]. De Pauw et al. go on to say that "We believe that tools that focus on the dynamic behaviour are essential to fulfilling the promise of the object-oriented paradigm. We also believe that visual tools are most effective for this purpose" [DePa97]. A number of visualisations exist for object-oriented software, many of which are based on static analysis of the source code. These are useful, however, a number of issues arise when static analysis is applied to object-oriented software due to the dichotomy between the static specification and dynamic behaviour. Dynamic binding introduces

problems as the method calls are decided at runtime and not at compile time. Therefore, it is impossible for the calling structure to be extracted using static analysis in cases when dynamic binding is used. It can be very difficult to work out which class is referenced and/or called when the reference type is to a base class or interface that has a large number of classes derived from it. In this case, the possible structure or behaviour of the program can be very different dependent on which class is actually referenced. Problems also occur with “invisible connections” between classes. For instance, in Java every class inherits from a base class Object. Therefore, the standard data structure classes just need to store references to the Object class in order to be able to store objects of any class. When these data structures are used it is then very difficult, or even impossible, to see which classes are being stored in them. It can also be very difficult to see how the classes are used from the source code alone, for example in terms of the number of instances of the class and what other objects the instances reference. Inheritance, dynamic binding and polymorphism introduce new challenges into the comprehension process. Inheritance allows a large amount of functionality to be provided by an object, with the possibility of overloading methods of the inherited classes. Therefore, in order to understand a class, one must understand the classes it inherits from and observe which, if any, methods are overloaded. Runtime visualisation of OO software can therefore help with these challenges and allow a better idea of the structure of a piece OO software to be gained. It is for these reasons that this work focuses on visualising object-oriented systems using runtime information.

As with any code, a number of bugs can exist in an OO system. These can include traditional control flow errors, such as looping unexpectedly and also errors to do with the object-oriented nature of the program, such as creating too many instances of an object. A number of these are summarised by [Laff94] and highlight another area where OO specific visualisations can be beneficial, as such errors are only apparent at runtime.

## **4.7 Current Trends and Issues and Challenges**

Current runtime visualisation systems come under a number of trends. They can be broadly categorised into:

- Educational tools. These systems are designed to aid the teaching and learning of programming. They include systems such as JavaVis [Oech02] and VisiVue<sup>TM</sup> [Visi]. These systems therefore focus on the details of the running programs and are useful for very small "learning" programs. A number of systems have also been developed for demonstrating specific data structures by having specific layouts for trees and lists, or by allowing the user to declare the visualisation of these, for instance work by Korn and Appel [Korn98].
- Debugging tools. These systems focus on aiding the debugging and development of OO software. They are often based on debugging techniques allowing control over the execution of the program under study. They allow detailed inspection of specific sections of the code and often offer support for other views of the software. Systems designed for this include Look [Look], Program Explorer [Lang95] and VisiVue [Visi].
- Higher level performance and architectural aids. These systems focus on presenting summary information on the program's execution to aid in performance tuning tasks and showing more

abstract views of the program. They typically use an offline approach. Jinsight [Jins] is an example of such as tool though it focuses on lower level class and object interactions. Other systems include work by Walker et al. [Walk98], who use high-level models to visualise dynamic information about OO systems at the architectural level.

These are not exclusive categories and some systems could be classified as focusing on more than one area, such as educational and debugging uses. However, these categories aim to show the main trends in the field. This shows the current focus of the different systems. There is also growing use of UML representations to show dynamic features of the software. For instance, sequence diagrams and statechart diagrams have been used, examples of this can be seen in work by Mehner [Mehn01][Mehn02] and in work by Systä [Syst00] where variations on these diagrams are presented.

One of the main focuses of the existing tools is showing low level details with the aim of aiding debugging tasks. Tools such as VisiVue, can be useful for investigating specific parts of the software or for demonstrating how smaller programs and data structures work. However, it is not their aim to aid comprehension of large-scale software of which the user has no previous knowledge. Therefore, there is potential for higher level views combined with showing such low level details in order to aid program comprehension. The current tools provide limited levels of abstraction and typically use only one or two representations to represent the software. Progress could be made by using multiple representations to display different aspects of the system in a common environment. This combined with user abstractions and annotation and an increasing linkage to external data sources, such as JavaDoc, could aid program comprehension activities. Many tasks require both high level and low level information in order to complete them successfully. For instance, debugging tasks may require the user to first gain an overview of the software in order for them to localise their search to the problem area, once the search has been refined, then low level information may be needed to see which elements are actually causing the error.

Runtime visualisation systems face the generic software visualisation challenges discussed in section 3.6. However, the runtime aspects can introduce new intricacies and difficulties. Scale is a major issue for software visualisations in terms of dealing with the massive amounts of information that are present on a real world piece of software. This is a particular problem for runtime visualisation, as there can be even more information, making it difficult to manage let alone visualise! This is due to the combination of the large amount of static information available, plus the state of constant change that is introduced by the dynamic nature of the software. This runtime information also introduces temporal relationships between items and a large amount of new information may need to be presented at once in a coherent manner. This dynamism affects all aspects of the visualisation, from layout, to representation and evolution. The layout cannot be known in advance (for online systems) resulting in added complications for the layout of items within views. Many issues arise when incorporating the new information in existing views. Should the layout try to maintain as much consistency as possible with the old layout? Or should the items be laid out in the best possible way given the new data? How shall changes in the data be presented? When the user is focusing on a specific item of interest how shall changes in the rest of the data be presented? What about changes that affect views other than the one the user is currently using? Does the user want to see just the current state or all the states up to that point? Do the user annotations and abstractions want to be

preserved between different executions of the software or do they only apply to certain executions or inputs? There are no definitive answers to these questions and they can depend very much on the task and user requirements. However, it is important for the system to maintain a consistent approach and try and reduce user disorientation. Navigation is complicated in a runtime visualisation system, as not only can the user navigate through the information but also through time. The user may have to navigate through a large amount of runtime information in order to find an item or event of interest and it may only occur under certain conditions or at certain points in the program's execution. For instance, the execution of some functionality may be hidden behind the code for the user interface and may prove difficult to locate. Some systems allow the program's execution to be back stepped e.g. ZStep [Lieb97], however, this requires the preservation of previous states or the steps required to restore those states if the visualisation is using an online approach. This is obviously unsuitable for large programs and techniques that allow the visualisation to show previous states without affecting the execution of the program can lead to user disorientation with the visualisation and program being at different points in the execution. These problems do not apply to offline approaches where the visualisation is purely based on a trace, however, in such systems disorientation can occur if two views are presenting different pieces of the program's execution. The evolution of the visualisation is also affected by the dynamic nature of the software. Not only can the source code change but also the software's runtime behaviour can vary hugely depending on data inputs. The visualisation must be able to cope with changes in both aspects, yet try and preserve some consistency between the cases.

The changing state of the software can increase user disorientation due to the large information space and the possible lack of consistency. This is especially true for certain information, such as a call stack of a thread, which are in a state of constant change. The sheer speed of change of such details means that it is impossible to watch them in real time regardless of the representation. Such details are needed for low-level investigation when the program's execution is paused and summary information is needed to present the bigger picture. Such as summarising method calls from a particular method instead of requiring the use of the call stack to find the information.

Problems can also exist in generalising findings. The visualisation only shows how the software behaves for that particular execution and data set. Therefore, more investigation may be needed to allow items under study to be fully understood. Some generalisations may not hold for all executions of the program. This is an important factor for the user to consider, especially if the software only loads certain sections of code dependent on its input or environment. In this situation the software may appear simpler than it actually may be!

The performance of runtime visualisation systems can also be a major challenge. The usability of such systems requires that the user can feasibly extract and visualise the software without prohibitive performance overheads. This performance issue is not as important for visualisation systems based on static analysis where the data can be extracted pre-visualisation time using efficient parsing techniques. In this case, the data is static so it does not need updating or monitoring while the user investigates the visualisation. The main performance issue for static analysis visualisation tools is dealing with the large volumes of data generated by substantial software systems. However, when visualising information



extracted using dynamic analysis, i.e. runtime visualisation, even small software systems can generate vast amounts of dynamic information. This performance issue is especially important for online approaches where the visualisation occurs alongside the program under study. In this case the software can generate a very large number of events and the visualisation must aim to interpret and incorporate this information into its presentation within an acceptable overhead.

## **4.8 Conclusions**

This chapter has presented details of the existing approaches to software visualisation at runtime. The viewing of runtime information takes its most basic form through the use of debuggers to inspect a program's internal state. However, specific runtime visualisation tools have been developed that allow a greater insight into the structure of a program. A summary of some of the main work in the field has been presented allowing an overview of the common techniques and problems to be seen. The issues with current approaches were discussed, followed by features which are considered desirable in a runtime visualisation system. This chapter has highlighted the need for greater research and as Jeffery states "*Monitoring and visualizing the dynamic behaviour of programs is a major area of research that has not been fully explored*". [Jeff99 p3]

# Chapter 5 The DJVis Approach

## 5.1 Introduction

This chapter presents the approach taken in the development of a visualisation of the runtime execution of object-oriented software. The aim of this visualisation is to aid the comprehension of OO software by visualising both static and dynamic details of it. Firstly, the chapter provides a definition of terminology. The choice of language to target using the visualisation is then discussed before the actual constructs of the language are presented. This allows the mapping to the representations within the visualisation to be fully detailed. Alongside these descriptions of the representation and mappings, the aim of each aspect of the visualisation is detailed. The visualisations presented make up the DJVis visualisation and this is discussed as a single visualisation and as a combination of specific visualisations, each focusing on showing certain aspects of the executing program.

## 5.2 Terms

The description of the DJVis visualisation uses a specific terminology, which is defined here for clarity. These definitions are defined with the purpose of being used for the description of DJVis.

- **Representation:** A representation is the graphical items that make up an image for some data item in the software.
- **Mapping:** The mapping defines the relationship between the data items and the individual representations.
- **Visualisation:** A visualisation is a combination of representations and the mapping of these to items of data, along with an interaction method.
- **View:** A view, in the context of DJVis, is a visualisation of some specific aspect of the program's execution.
- **Abstraction:** The abstraction is the level of detail shown of an item or in a view. Higher levels of abstraction present less information on an item.

## 5.3 Language Choice

There are many different languages based on the object-oriented paradigm. Each language provides its own variations and therefore the choice of language to be visualised influences the resulting visualisation. For example, C++ allows multiple inheritance of non-abstract classes and code and variables can be outside of classes, whilst Java does not. Issues such as this influence the resulting visualisations. This work focuses on the visualisation of Java software. The reasons for this are:

- Java does not allow code to be outside the scope of classes. Therefore, the visualisations can exploit this object-oriented nature of all code in the representation.
- There is growing use of Java and it has a large existing user base. Therefore, there will be a need to maintain this software and there is large scope for the application of the resulting visualisations.

However, the visualisations are not restricted to Java alone and could be applied to other languages, though changes would be needed to take into account differences in their support for the OO paradigm from Java.

### 5.3.1 Language Constructs

The visualisation will be based on the language constructs of Java. The following are the elements of a Java program that are useful for providing information about the program's runtime behaviour.

#### Class

- Name
- Methods
  - Name
  - Access Rights (public, private, protected)
  - Arguments
  - Return Type
  - Local Variables
  - Number of Calls
  - Calls which other methods and classes
  - Called by which other classes and methods
  - Length
  - Coverage
  - Lines defined at
  - Final
- Fields
  - Name
  - Type
  - Access Rights (public, private, protected)
  - Values for objects of class
  - Accessed by which other classes and methods
  - Is it used to reference a sub type of the actual field type.
- Package defined in
- Inherits
- Implements
- Number of instances created
- Created by which other classes
- Inner, Abstract, Final
- File defined in

#### Object

- Class of object
- Values of fields

## Interface

- Name
- Inherits
- Implemented By
- Methods
  - Name
  - Access Rights (public, private, protected)
  - Arguments
  - Return Type
- Fields
  - Name
  - Type

## Package

- Name
- Classes within package
- Classes in this package used by which other packages / classes

## Local Variables and Method Arguments

- Name
- Type
- Scope
- Value

## Thread

- Name
- Thread Group it belongs to
- Call Stack
- State (Running, Sleep, Wait, Zombie, Monitor, Suspended)
- Owned Monitors
- Currently Contented Monitor

## Thread Group

- Name
- Threads it contains
- Thread groups it contains
- Parent thread group

The Java thread architecture imposes a hierarchical structure on some of these constructs as it executes the software. Java is a multithreaded language, so one basic element for the runtime visualisation is that of threads. Each thread belongs to a thread group, which can contain threads and other thread groups, therefore forming a hierarchical relationship. Each thread has a call stack, which comprises of all the methods that the thread is executing. Java code has to belong to a class and each method on the stack is part of a class. This outlines the runtime aspects of the Java thread architecture. At this level the representation is quite procedural in nature. However, due to the object-oriented nature of Java there are also higher level relationships between the active objects. These form a network of communicating objects, each being an instance of a particular class. There is also a static hierarchical structure between some items. Java code cannot exist outside of a class and all classes belong to a package. Classes, which are not defined as part of an explicit package, are loaded as part of a default package. Therefore, each package contains classes, which then have methods and fields. These relationships allow natural abstraction levels and the use of these are discussed in the detailed descriptions of the DJVis views. Using the language constructs a number of relationships can be extracted from the program under study: The following list is not exhaustive and other relationships exist, such as field belongs to class. Table 5-1 lists the main relationships.

<b>From</b>	<b>Relationship</b>	<b>To</b>
Class	Creates	Class
Class	Created by	Class
Class	References (Statically)	Class
Class	Referenced by (Statically)	Class
Class	References (Dynamically – through a base class or interface)	Class
Class	Referenced by (Dynamically)	Class
Class	Calls method of	Class
Class	Methods called by	Class
Class	Field accessed by	Class
Class	Inherits	Class
Class	Inherited by	Class
Class	Implements	Interface
Interface	Implemented by	Class
Interface	Inherits	Interface
Interface	Inherited by	Interface
Method	Calls	Method
Method	Is called by	Method
Method	Calls method of	Class
Method	Is called by method of	Class
Method	Accesses field of	Class

Field	Accessed by	Method
Field	Accessed by	Class
Thread Group	Contains	Thread Group
Thread Group	Contained by	Thread Group
Thread Group	Contains	Thread
Thread	Contained by	Thread Group

**Table 5-1 Dynamic relationships between Java items in an executing Java program**

The focus of the DJVis visualisation is to concentrate on showing class level and threading details of an executing program. The visualisation of object level details is not considered in the visualisation.

Therefore, the above table does not include relationships involving objects, which are defined in Table 5-2.

From	Relationship	To
Object	References (Statically)	Object
Object	Referenced by (Statically)	Object
Object	References (Dynamically – through a base class or interface)	Object
Object	Referenced by (Dynamically)	Object
Object	Instance of	Class
Object	Creates	Object
Object	Created by	Object
Object	Accesses field of	Object

**Table 5-2 Object level relationships**

## 5.4 DJVis

This section describes the DJVis visualisation. This is presented as an overview of the DJVis visualisation followed by a detailed summary of the individual views that combine to make up DJVis.

### 5.4.1 Overview

The DJVis visualisation is composed of a number of views, each of which aims to show some aspect of an executing Java program. It would be impossible to try and show all aspects of a piece of software within one view or using a single representation, and therefore interconnected views are used within an integrated environment. Table 5-1 highlights the different relationship types, and it can be seen that the information can be separated into three main categories based on class, method and thread relationships. The visualisation was developed using this separation of information, with a specific view for showing each information type. This allowed the different relationships to be logically separated, thus reducing the amount of information that each view needs to display. There is some overlap between the views, as some relationships follow logically from others, however, the main aim was to reduce the amount of information in each view and use co-operating views to allow the user to easily use multiple views to investigate the different relationships. These main views of the executing program are:

- Runtime View
- Class View
- Variable Watch View
- Method Pixel View.
- Query View

The Runtime View and Class View have been introduced in work by Smith and Munro [Smit02]. However, this chapter offers a full description of the views and their features. DJVis also provides a syntax highlighted text view for showing source code and a number of textual displays that are used by the main views and described in the relevant view description. The views each have different objectives. However, as previously mentioned some details of the program's execution are shown in more than one view. Table 5-3 highlights which views show each of the relationships previously identified Table 5-1.

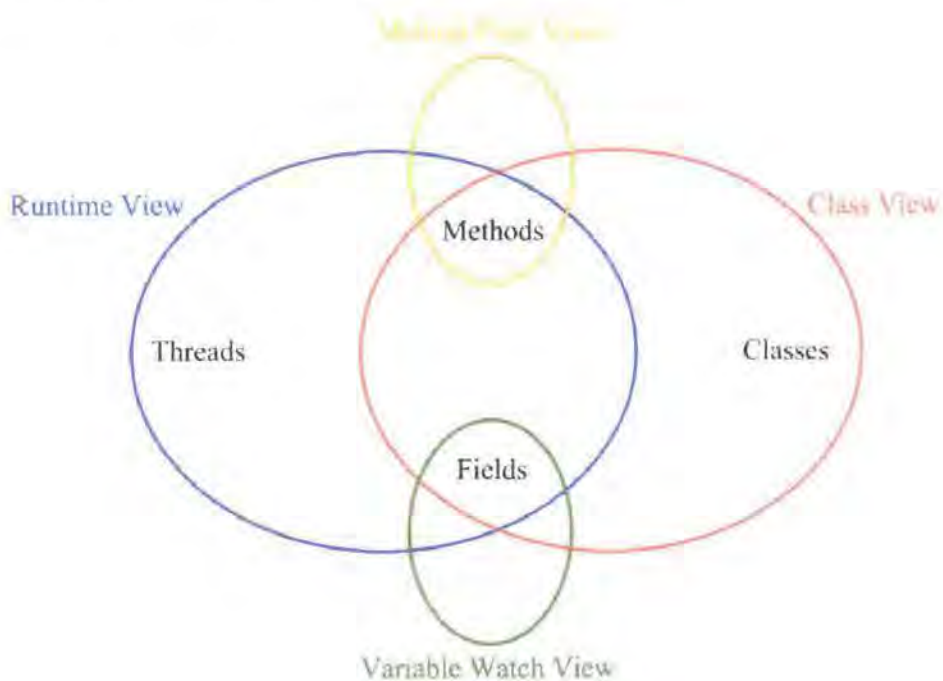
From	Relationship	To	Shown In
Class	Creates	Class	Class View
Class	Created by	Class	Class View
Class	References (Statically)	Class	Class View
Class	Referenced by (Statically)	Class	Class View
Class	References (Dynamically – through a base class or interface)	Class	Class View
Class	Referenced by (Dynamically)	Class	Class View
Class	Calls method of	Class	Method Pixel View, Class View
Class	Methods called by	Class	Method Pixel View, Class View
Class	Field accessed by	Class	Variable Watch View
Class	Inherits	Class	Class View, Query View
Class	Inherited by	Class	Class View, Query View
Class	Implements	Interface	Class View, Query View
Interface	Implemented by	Class	Class View
Interface	Inherits	Interface	Class View, Query View
Interface	Inherited by	Interface	Class View, Query View
Method	Calls	Method	Method Pixel View
Method	Is called by	Method	Method Pixel View
Method	Calls method of	Class	Method Pixel View
Method	Is called by method of	Class	Method Pixel View
Method	Accesses field of	Class	Variable Watch View



Field	Accessed by	Method	Variable Watch View
Field	Accessed by	Class	Variable Watch View
Thread Group	Contains	Thread Group	Runtime View
Thread Group	Contained by	Thread Group	Runtime View
Thread Group	Contains	Thread	Runtime View
Thread	Contained by	Thread Group	Runtime View

**Table 5-3 Viewing the different relationships in the different views**

Each view is defined in full in the following sections, though it must be noted that the views are not restricted to only showing the information displayed in Table 5-3. Additional information from the language constructs is also shown, but Table 5-3 shows how the main relationships identified in Table 5-1 are displayed. The description of each view takes the form of the aim of the view, followed by the full mapping and a discussion of how each view can demonstrate concepts of interest. A summary of the information obtainable from each view is also provided. Once the specifics of the views have been described individually, some other aspects of the visualisation are presented that apply to the whole visualisation and aim to provide consistent features across all views.



**Figure 5-1 An overview of the information shown by each view.**

Figure 5-1 shows an overview of the information presented in each view and summarises the information shown in Table 5-3. The Query View is not shown on this overview since it presents information from the other views and therefore can show all information types presented in the overview.

## 5.4.2 DJVis Views

There are 5 main views that make up DJVis, each of which is presented here using a template that describes the aim, mapping and summary of each of the views. Each of the main views also includes other helper views that provide additional information. The main five views can be summarised as:

- **Runtime View**                      Presents the threading aspects of the software using a 3D representation.
- **Class View**                              Provides details on the classes and their relationships using an augmented graph representation.
- **Variable Watch View**              Presents information on the access types and frequency of accesses for a field variable.
- **Method Pixel View**                  Presents the method calling relationships of a class' methods using a pixel based representation.
- **Query View**                              Acts as a grouping mechanism for information from other views.

The first four views provide their own representations of the information being displayed, whilst the Query View is distinct in that it provides a mechanism for the user to group information from the other views. In most instances the representation used is the same as in the original views.

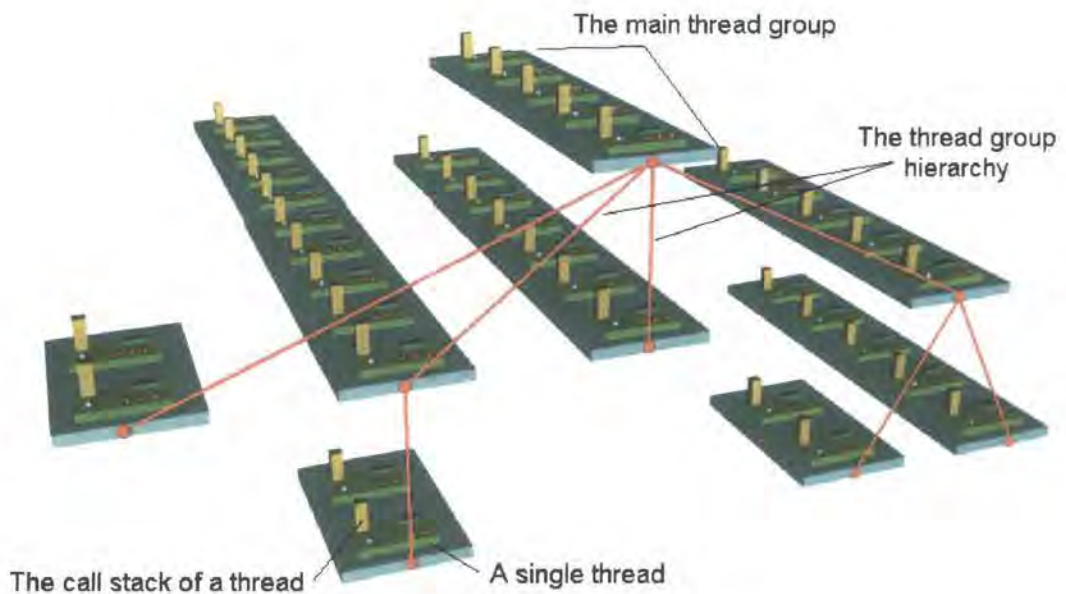
### 5.4.2.1 *The Runtime View*

#### **Aim**

The aim of the Runtime View is to present the low-level runtime information about the software's execution. This includes details such as the programs call stack(s), as well as information such as control flow and variable values. Much of this runtime information can be thought of as hierarchical, due to the thread architecture. Here, the executing program is made up of one or more thread groups. These structures can contain threads and/or other thread groups giving a hierarchical division of threads into groups. Each of these threads then has a stack, which contains method calls and local variable values. Each method being executed on the stack gives information on control flow and variable access.

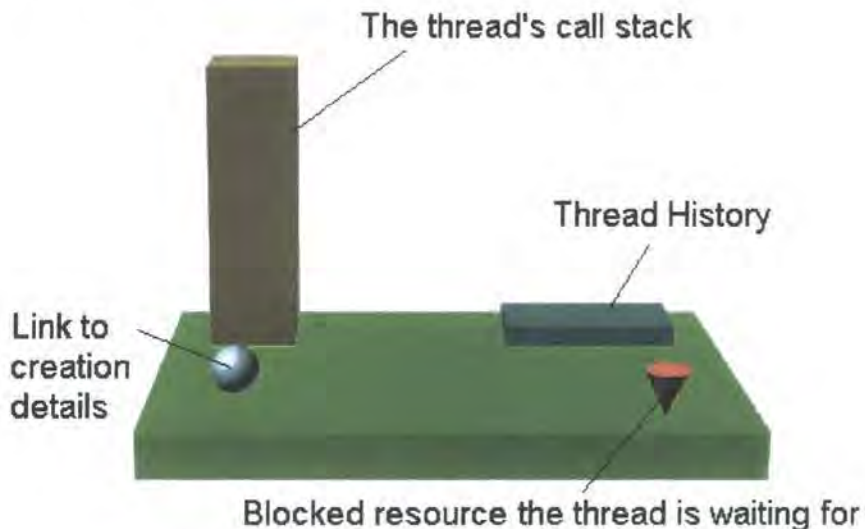
#### **Mapping**

The mapping of the Runtime View takes an approach which follows the basic levels of abstraction of the executing program, in terms of thread groups, threads and their call stacks. This allows a natural abstraction of the program, which will be familiar to users through its use in debugging tools. Also, it reduces information overload by restricting the amount of information that it is necessary to display at once. Using this, a visualisation was devised to display the information at each of these levels and to allow coherent changes between them. The visualisation used by the Runtime View initially presents the program abstractly in terms of an overview of its thread groups and the threads structure.



**Figure 5-2 The Thread Group Hierarchy**

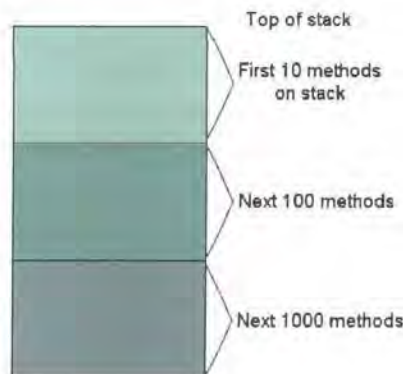
Figure 5-2 shows the thread group hierarchy as a tree of thread groups (each turquoise platform). On top of these sit the smaller green platforms of each thread in the thread group. From this overview of the thread groups and threads, the user can focus on items of interest. As the user moves the viewpoint closer to an item, then more details on the item become available.



**Figure 5-3 A single thread**

Figure 5-3 shows the details of the view of a single thread. This view presents and allows access to a variety of information on the thread. Most importantly perhaps is the call stack of the thread. At a distance this indicates the number of methods on the call stack and provides greater information on the actual methods when zoomed in upon. The view also gives an indication of when the thread is in contention for a monitor. This could allow threads that are frequently being blocked to be seen. The view could also provide links to thread creation details, such as the code where it was created and the parent thread, as well as details of thread history, such as common blocking etc.

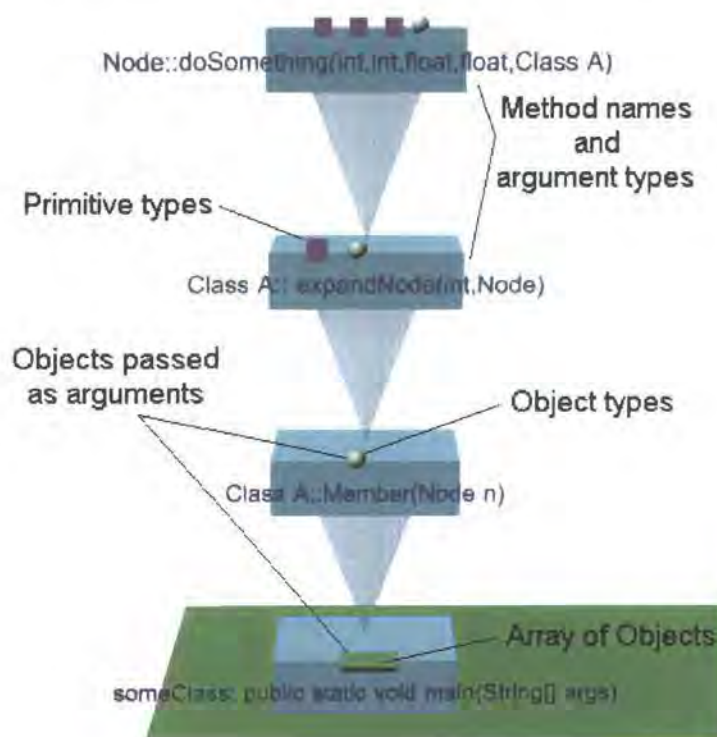
Problems can exist with this representation when the thread's call stack is large. This would result in a very tall call stack, which could make it difficult to observe in its entirety and it may occlude other objects above it in another Thread group. Such a large structure would also be difficult to navigate. A possible solution to this would be to use a logarithmic scale upon the call stack as demonstrated in Figure 5-4



**Figure 5-4 A possible scaling of methods on the stack**

This approach would allow the user to easily see the approximate number of methods on the stack due to the colour coding, whilst being able to handle a large number of methods without problems of scale. With a suitably chosen scale factor it should also prevent the stack from growing too large and intersecting objects above it.

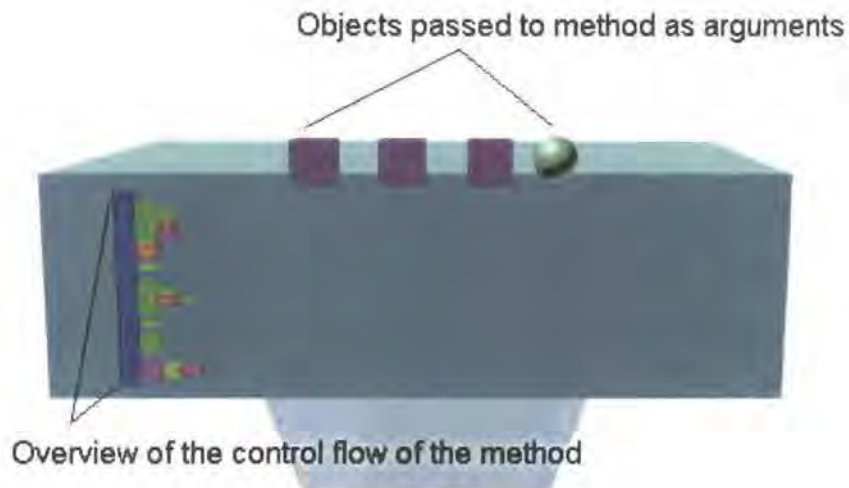
From the view of an individual thread platform, the user can focus further, if desired, by zooming in on the thread call stack. This uses Level Of Detail (LOD) to reveal more information as the user zooms in. The stack changes from the coloured rectangle, to reveal the individual methods on the stack as is shown in Figure 5-5.



**Figure 5-5 Methods on a call stack**

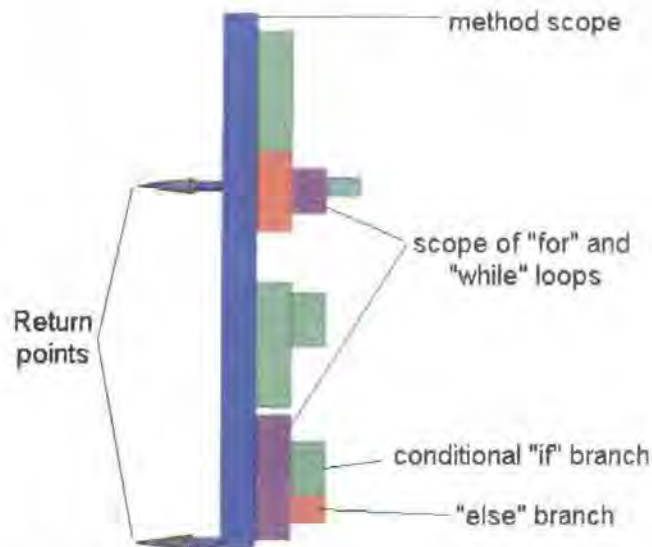
The use of a logarithmic scale for the stack representation prevents all the methods from being shown at once in order to maintain a constant size for the stack. Instead, the top ten methods on the call stack are displayed as normal. However, any remaining methods in the stack are scaled and presented using the logarithmic scale by maintaining the colour coded rectangle about them. This introduces issues in navigating the call stack, in particular to methods that are in the modified scale sections. A possible approach would be to allow the user to move the expanded section up and down the stack to inspect the area, in which they are interested, whilst other areas remain in the shaded logarithmically scaled rectangle. However, this prevents the entire stack from being seen at once and prevents the comparison of different sections of the stack at the same time. A text list of the methods on the stack could be optionally presented in a popup window alongside the graphics to allow quick scanning of method names and as an aid to navigating the stack.

This level of abstraction presents an overview of the methods on the call stack and the objects that they take as arguments. In order to find more details on a method the user can zoom in further on a single method.



**Figure 5-6 Details of a method on the stack**

The zoomed in details of the method are revealed using level of detail, to fade from the original method box to the additional details shown in Figure 5-6. The argument objects are scaled as the view is zoomed, in order for them to remain a constant visual size and prevent them from appearing dominant. This view of a method presents an overview of the control flow of the method.



**Figure 5-7 Presenting an overview of control flow information**

Figure 5-7 shows how the scope of the code in the method is presented, with the scope indentations coloured according to type. The scope is shown from the start of the method down to its completion, allowing the basic control flow of the method to be seen without browsing the source code and showing the proper scoping for source code that has been badly indented. The green signifies a conditional block of code from an "if" statement, while red shows the "else" conditional code. The purple code blocks represent looping and the blue block represents the entire method body. The arrows to the left of the method scope show return points for the method, thus allowing the user to easily observe possible return points and associated values.

This view is only intended to give an overview of the scope and as all method boxes are the same size, the scoping information is mapped at different ratios dependent on the length of the method. This means that for large methods only the general pattern will be visible. In order to allow the size of the method to be seen a scale is placed to the left of the scoping information. This is colour coded similarly to the call stack scale. This gives an approximate size for the method at a glance. The scoping information has the advantage that the correct scoping information can be presented even if the actual source is not indented properly. This could allow a rough guide to the complexity of the method to be gained and set up expectations for the user, which could be checked against the source code to highlight any scoping errors. The actual source code for the method is presented in the Source View, which is displayed beneath the Runtime View. The large free space to the right of the scoping information in Figure 5-6 could be used to show additional information, such as indicating any lines of break points, or highlighting if there are any user annotations, comments or hypothesis on the method.

## Summary

This section has given a detailed description of the Runtime View. The view provides access to low-level details of the software's execution. The representation closely maps to the underlying information and the view uses level of detail to allow the user to obtain detailed information on an item of interest. It allows a user to obtain the following information:

- Thread groups
  - Name
  - Parent and child thread groups
  - Threads in thread group
    - Name
    - Status
    - History and creation details
    - Call stack
      - Methods on stack
        - Name
        - Arguments
        - Execution position
        - Local Variables
        - Scope and control flow information

The Runtime View was not designed in isolation, but to be combined with the Query View in order to improve its usability.

#### 5.4.2.2 The Class View




##### Aim





The aim of this Class View is to present details on the classes and their relationships. The view aims to be customisable thus allowing the user to investigate different relationships within a common framework.

The aim of the view is to show the structure of the software at the class level. The user should be able to see which classes are involved and get an impression of a class' structure.

##### Mapping

The Class View is based on an augmented graph representation. The nodes represent the classes in the software whilst the edges represent relationships between the classes. The representations used in the Class View are summarised in Table 5-4.

Representation	Meaning
	Class (shading represents number of instance created)
	Interface
	Inner Class (shading represents number of instance created)

	Methods (length and shading represent user selectable metrics)
	Fields (shading represents user selectable metrics)
	Type yet to be loaded
	Package type belongs to (Colour coded by package)

**Table 5-4 Class View Representations**

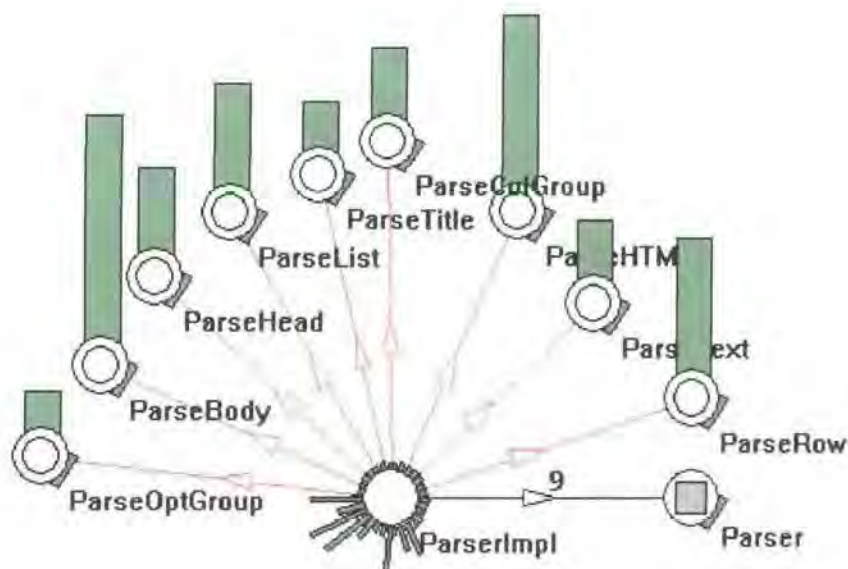
The circular nodes represent classes and are coloured to represent the number of instances of each class, from the start of the program's execution or from a specific point. Around the circular node of the class, the methods of that class are represented as lines coming out from the node. This allows the number of methods each class has to be easily seen. Methods that are inherited by the class are not displayed, as they are displayed for the parent class. These method lines can be shaded and altered in length to represent information, such as the method length, complexity, number of calls and the access rights of the method. For example, Figure 5-8 shows the GraphDesktop class, with the method line length representing the length of each method and the shading representing the number of times that the method has been called. The class has eleven methods (four that haven't been called (shown as white), four that have been called a few times (shown as light blue), two that have been called slightly more (shown as blue) and one that has been called many times (shown as dark blue)). Two of the methods are long in length whilst the other nine are short.



**Figure 5-8 Viewing the GraphDesktop class in Class View**

Figure 5-8 shows the representation of a single class. These class representations are connected by edges dependent on the relationship the user wishes to investigate. A number of relationships exist between the classes as Table 5-1 highlighted. These can all be shown in the Class View, using a drop down list to allow ease of changing between them. Class references can be displayed both in terms of the traditionally displayed static references, for example as with UML class diagram and also dynamic references through a base class or interface.





**Figure 5-9 Class View showing static and dynamic references**

Figure 5-9 shows an example of dynamic and static references. In this figure the method lines represent the length of the methods and the colouring represents the access rights of the method (green for public methods, orange for protected methods and red for private methods). The ParserImpl class has nine references to the interface Parser. However, at runtime these nine reference fields are used to reference the Parse\* classes. The red edges represent the dynamic references to these classes, whilst the static references are shown by the black edges. A black and red dashed line would show composite dynamic and static edges, though none are present in Figure 5-9. Figure 5-9 also demonstrates how interfaces (grey square within the class node e.g. Parser) and inner classes (inner circle within node e.g. ParseRow) are represented. The ParserImpl class can be seen to be static, as it has no instances (shown by the white node). It contains the nine inner classes Parse\*, each of which implement the Parser interface, which can be seen by changing the edge mapping. The ParserImpl class acts as an interface to the functionality of these inner classes to the rest of the program. This is also suggested visually by its Class View representation, as the ParserImpl class has many very short methods and it is the only class to reference these inner classes.

The examples presented so far have all used the length of the method lines to represent the length of the method. This mapping allows an impression of the method to be gained in terms of the amount of code it involves, thus allowing classes with a lot of functionality to be easily spotted. However, the length of the method lines can be used to represent other metrics, such as complexity metrics or even for representing the number of calls of the method. Using the method line to represent the number of calls allows the user to gain an indication of which sections of the code are being executed as the program runs. This can help in the localisation of functionality, for example by tracing the method calls whilst the program performs some functionality, it is then easy to observe which methods and therefore classes were involved. The method lines could also be used to represent metrics extracted from other sources than the visualisation tool itself. For example, version control information could be used to show the number of changes in a method by changing the shading or length of the method line. This could give a higher level indication of

where changes have occurred, between this and another specified version, thus allowing the user to investigate changes of interest.

The Class View uses a similar representation to show the field variables of the presented classes. Instead of using method lines the fields of the class are represented using triangles. This allows the user to easily distinguish them from the method presentation in order to reduce any possible disorientation as both methods and fields are shown in the same view with the user able to simply toggle between them. As with the method lines, the user has control over the shading and length of the field triangles in order to show different information. The shading of the field triangles can represent the number or type of accesses, type of the field and access rights. The distance the triangle comes out from the class node can be either fixed or it can represent the number of accesses of the field. Figure 5-10 shows an example of the representation in use. Here, the DisplayFrame class (left) is shown using the access rights shading mapping. This class has a number of private fields shown as red and two public fields, shown as green. The GraphCanvas class (right) shows an example of the field type mapping. Here, the blue shading shows fields that are based on Java primitive data types, whilst the purple shows fields of Java API class reference types and the yellow shows fields that are user class reference types.



Figure 5-10 Showing field variables in the Class View

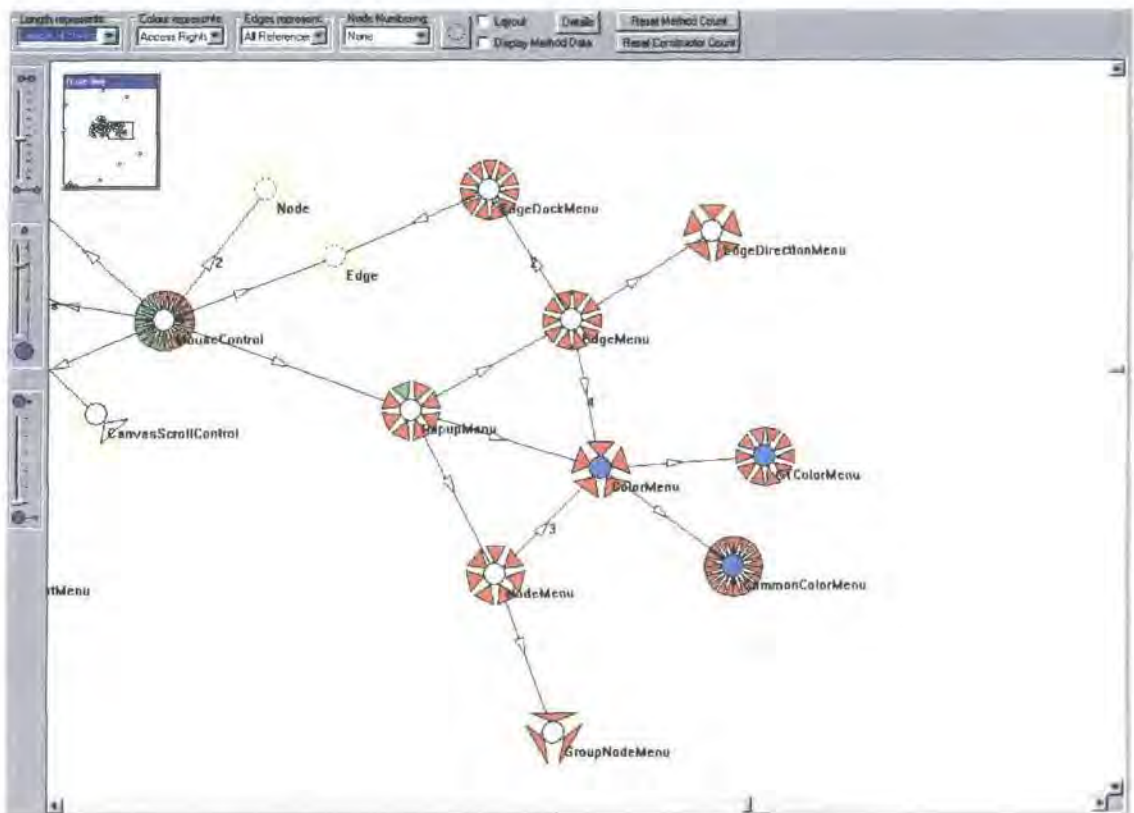


Figure 5-11 Showing data encapsulation using the Class View

Figure 5-11 illustrates the Class View showing the field data using the access type mapping for the shading of the field triangles. This view allows the user to observe how variables are used across the software and aids them in their comprehension of the classes. It allows the user to gain an impression of the data hiding that has been employed in the implementation of the software. For example, Figure 5-11 shows an example with a cluster of classes where nearly all the fields are private (shaded red). This suggests good data hiding, while a large number of green i.e. public fields would suggest areas where data hiding is limited and the classes may be heavily coupled to other classes which rely on these public fields.

The Class View visualisation is based on a graph representation. Graph representations offer an intuitive representation for software engineers, although, they suffer from problems of scale and complexity as the number of nodes and edges increase [Knig99b]. In order to improve the representation the view supports grouping of the nodes in order to abstract known or unrelated modules in the view. For example, classes responsible for the user interface may be grouped so the user can focus on the underlying processing classes, which they are investigating. Classes can also be filtered from the view in terms of being removed or being presented but highlighted as filtered (and no usage, instance or calling information is available for them). The view indicates that there are hidden classes, so the user does not forget that they have hidden some classes and therefore that they are not seeing the true picture.



**Figure 5-12 Displaying User Abstractions**

Figure 5-12 shows how user abstractions are displayed in the Class View. The left image shows the abstraction collapsed i.e. just showing its name, whereas the right abstraction is expanded allowing the circular nodes of the six classes it contains to be seen. This gives an impression of the contents of the abstraction and the names can be obtained using a mouse over operation. The method lines are not shown for the classes inside the abstraction and as the number of classes increases, then the size of their nodes becomes smaller to prevent the abstraction node from becoming excessively large. The abstractions can be removed from the display in order to allow the details of the nodes to be inspected. The user can also display a node in all its detail in a sub window by right clicking on the node in the abstraction and choosing the "Details" menu option. This option can also be used to show a zoomed in version of the node for any node in the Class View, in order to allow detailed inspection without needing to change any of the graph display options. The edge relationships of nodes within the abstraction are not shown, and only edges to nodes outside the abstraction are displayed. Abstractions can contain other abstractions and these are displayed as embedded abstractions as Figure 5-13 demonstrates.



**Figure 5-13 Embedded Abstractions**

The use of user abstractions and filtering allow the complexity of the view to be reduced. Other filtering options could also be supported, such as the ability to hide inner classes, or to remove static references to a base class or interface that are also shown as a dynamic references to a sub class.

The Class View provides an overview window to allow the entire structure of the graph to be seen and to aid in the navigation of the graph. This overview window highlights where the user is currently focusing in the context of the whole graph, thus helping the user orientate themselves. Nodes in the view can also be located by searching using their name or the start of their name.

Java files are located in packages representing the directory structures in which they are found. This offers a natural separation of the code and the packages are highlighted in visualisation by using a lightly coloured circle behind the class node.



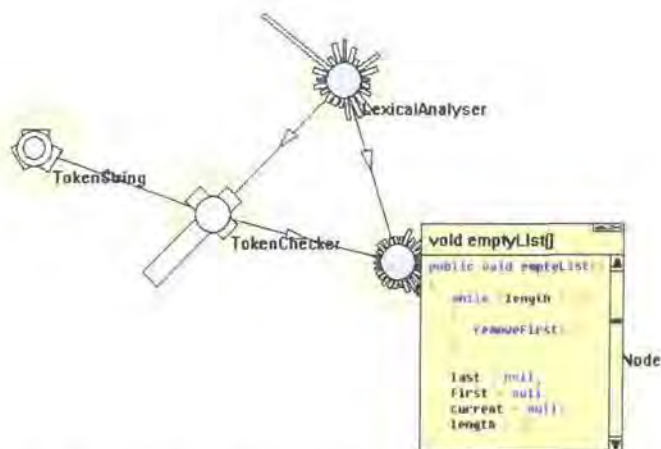
**Figure 5-14 Showing Package Inclusion in the Class View**

Figure 5-14 shows the class MessageCatalog and the interface XMLReader, which are from different packages. The light coloured circle behind the node shows the package the node belongs to, with each package being allocated a unique colour based upon its name. This provides a consistent colouring scheme across each run of the visualisation and even between programs that use some overlapping packages. The package circles provide an easy way to see how the package is used within the program. Packages can also be used to filter items in the view, allowing packages of no interest, e.g. utility code, to be filtered from the current view, or by allowing all items in a package to be grouped into a single abstraction node.

To aid the user in navigating the graph, the Class View provides an additional helper view that displays the names of the classes and interfaces that have been loaded. The classes and interfaces are displayed using a tree control where the package structure is used for the branching of the tree structure and the leaf nodes are the classes and interfaces. This allows the user to quickly see the contents of a package and which packages the program is using. The packages are displayed next to a coloured icon which shows

the colour used for the package circle on the graph's nodes. The tree control allows the user to quickly navigate to a class in the Class View by selecting it in the tree control.

The Class View supports interactive customisation, allowing the nodes to be zoomed in upon to show more detail and to allow classes with a large number of methods to be seen more clearly. The Class View also acts as a navigation method allowing classes to be inspected in other views such as in the Method Pixel View and in the source code viewer. The view supports mouse over and selection pop ups of additional information such as method or field names. The user can also have the option of having the source code displayed in the pop information to reduce user disorientation in having to change views. This is demonstrated in Figure 5-15. Alongside these features is the ability to annotate the items (classes, interfaces, methods, packages and abstractions) with user comments to allow the current hypothesis to be recorded or acquired information to be stored. This information would then be accessible in other inspections of the source code or by other users. For example, annotated comments recorded on classes in a package would be available when any software using that particular package is visualised. Annotations can be made in all views and this is discussed in section 5.4.3.

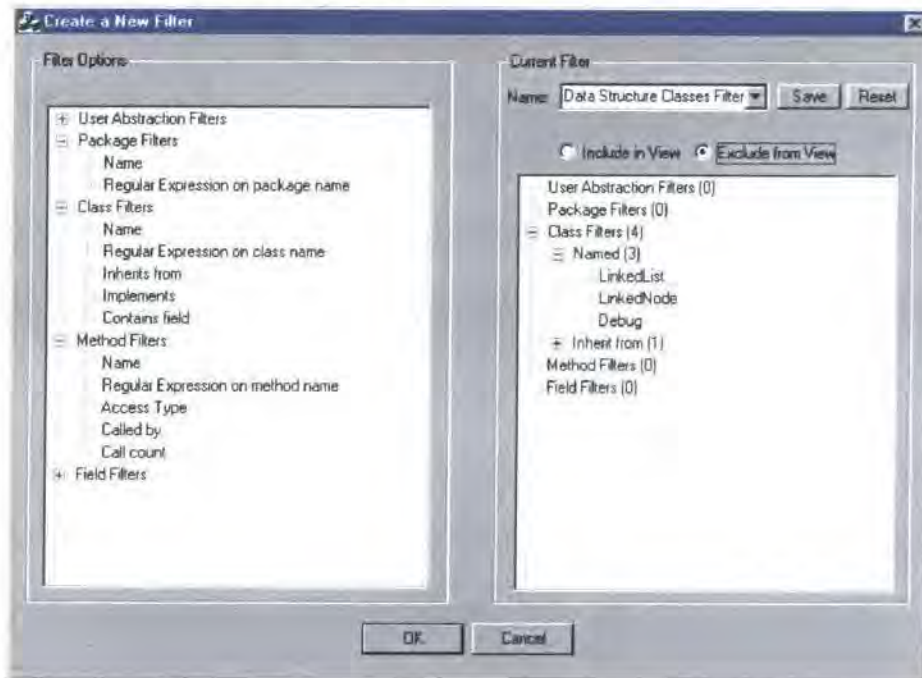


**Figure 5-15** Displaying the method name and source code in the Class View

The Class View presents only a single relationship at a time. However, it is often the case that a maintainer may wish to investigate multiple relationships at once, such as references and inherits. In order to allow this, multiple instances of the Class View can be opened and existing views cloned or saved. To provide ease of use the views can be linked together so that the selection of a node in one view results in all the other views focusing on that node. Views that are saved and then loaded are indicated as such by having a grey background. This same technique is used for snapshot views, where the view is frozen and no longer updated as the data changes. These loaded and snapshot views have their data fixed. This allows a user to compare the current state with a previous state.

The Class View presents a large amount of information and due to this the visual complexity of the view may be high for large software systems. In order to reduce this, the view supports multiple filtering facilities to allow the user to remove items of no interest. The filtering of items is provided at multiple levels of abstraction, namely filtering based on user abstractions, package structures, classes, methods and

fields or any combination of these. The user can define a filter using a simple tree control that lists each of these characteristics and then has the filtering options below it. This filter creation is managed in a popup dialog window as shown below in Figure 5-16.

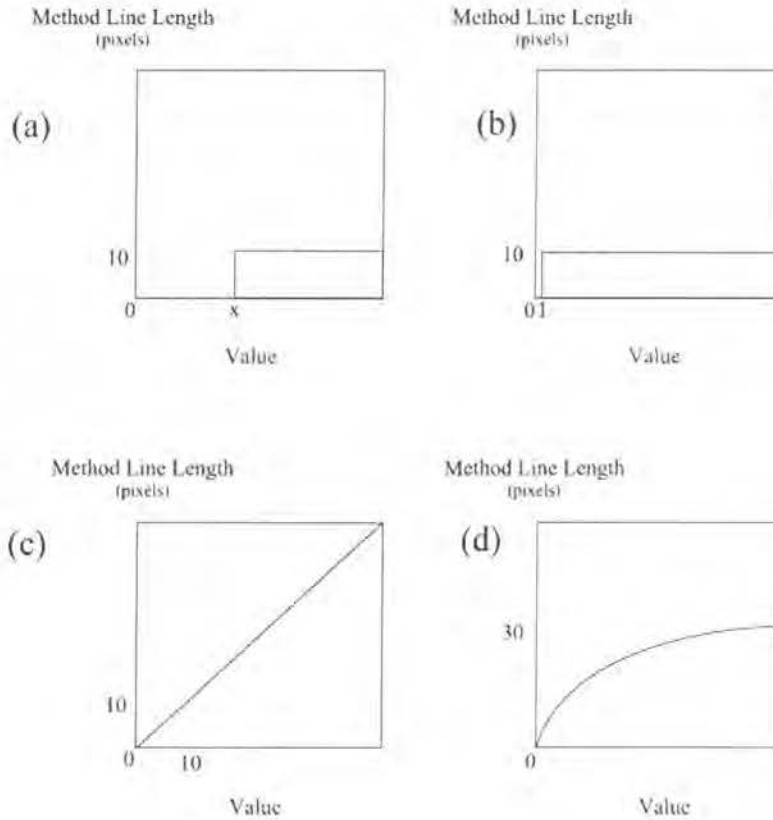


**Figure 5-16** Creating a user filter

The user can add an item to the filter based on the filter characteristics and this is added to the current filter. These filters can be named and saved to allow the user to quickly change between different filters and return to previous settings without the overhead of having to create the filter afresh each time. The current filter is displayed in a drop down list at the top of the Class View so the user can quickly see which filter, if any, is active and also change between filters. The definition of the filter allows the user to include or exclude items that fit the current filter characteristics. This provides a powerful mechanism for the user, allowing them to quickly focus on items of interest and reducing the volume of information they are presented with.

The method line idea allows the number of method calls to be seen by using the shading or the length of the line to represent the number of calls. The mapping of these values will depend on the program and task the user is undertaking. For example, a linear mapping function may be the most useful when the program is small and has a small number of method calls, or when the user is slowly stepping through a program observing the method calls from some particular point. However, when the time span of execution is longer and the method calls are high a logarithmic function may be more suitable. Otherwise, the method lines could quickly become shaded to the maximum intensity and the length of the lines become too long and start obscuring the view. This logarithmic scale allows the method line representation to scale up. However, the user may be looking for some distinct calling frequency or may want to make some particular values more important. In order to support this, a custom mapping mode is provided, where the user can define the mapping function between the method calls and length and shading intensity. This allows the user to easily create custom mapping functions that relate to their particular task, or to features in which the user is interested. For example, if the user simply wants to

know if a method has been called and they are not interested in the number of times, they could set all values except zero to map to a set length or shading intensity. This modification of the mapping function would be supported through the use of graphical manipulation of a graph as well as the ability to enter simple mathematical equations for the mapping function.



**Figure 5-17 Length Mapping Modes**

Figure 5-17 demonstrates four possible mapping modes that could be useful for different tasks. Graph (a) would show only items of values  $x$  and over and these would be represented with a constant length of ten pixels. This could be used for showing methods over a certain size, or highlighting frequently called methods. Graph (b) maps all values to the same value, except zero, whilst graph (c) shows the default linear mapping mode. Finally, graph (d) restricts the method lines to a maximum length of thirty whilst making smaller changes more prominent.

This approach can also be used for the other representations of the method line, such as the method length or complexity, which can be shown in terms of the length or shading of the method lines. This can allow the user to set values to effectively hide small or simple methods and highlight methods above some threshold of length or complexity. This can be useful for preventative maintenance tasks where there is a desire to find and improve complex methods.

## Summary

The Class View presents details of the software under study at the level of classes and their relationships. It provides access to a variety of information on the software and due to the key role of classes in the design and implementation of object-oriented software the Class View is essential in the comprehension

of the software under study. The view uses an augmented graph representation with method lines and field triangles for showing the details of classes. The view is highly customisable, providing drop down lists for changing how the underlying information is presented in the view, as well as facilities for creating user defined filtering and mapping modes. The view presents information on:

- Classes
  - Name
  - Methods
    - Name
    - Access type
    - Return type and arguments
    - Method Metrics (length, number of calls (total, from user defined point), complexity)
  - Fields
    - Name
    - Type
  - Relationships (inherits, implements, creates, references (static and dynamic))
  - Package inclusion
  - Number of instances
  - Type (inner, interface)

The representations used for the nodes in the Class View are summarised in Table 5-4. The Class View also provides an effective means of navigating the other information sources such as driving the Method Pixel View or the source code browser.

### **5.4.2.3 The Variable Watch View**

#### **Aim**

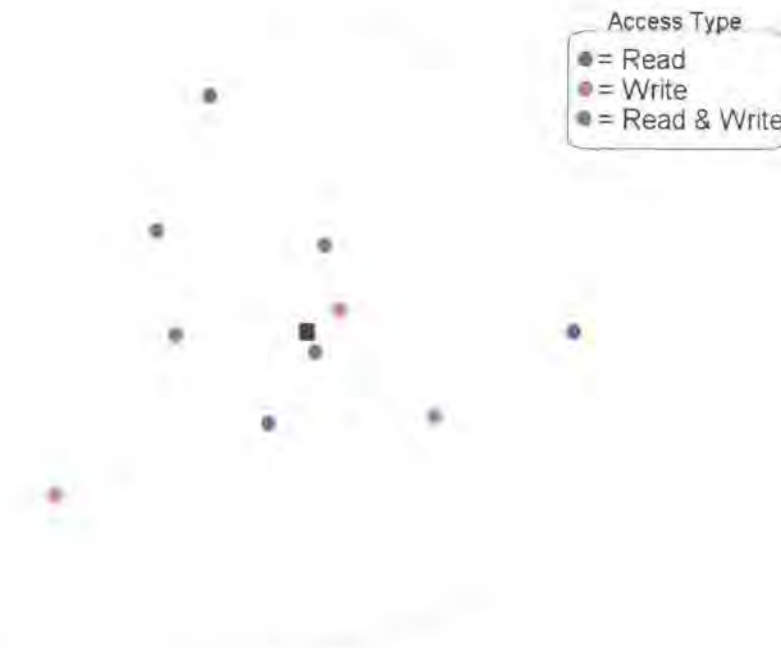
The aim of this view is to present information on the use of field variables. The view aims to provide the user with details of the use of individual class fields, allowing them to see the type of accesses (read, write or mixed) and the frequency of accesses. The overall purpose is to allow users to follow variable changes and locate dependent code. Therefore, the view aims to help users gain an understanding of how and where a field variable is used, as well as helping in tasks such as impact analysis by allowing the user to assess how tightly coupled the field variable is to the rest of the code.

#### **Mapping**

The view uses a radial layout of the information with the watched field variable being in the centre of the layout. The view offers three levels of abstraction by showing the accesses at different levels of abstraction. Initially, the view displays the classes that access the field variable and these classes are represented as small circles arranged evenly around the central field in a radial layout. The distance of the class circles from the centre of the main circle is used to represent the number of accesses that that class has performed on the field variable. Coming out from the central point are background circles which are designed to allow the user to easily compare the number of accesses between different classes and



provide them with an indication of the magnitude of accesses. These "scale" circles are displayed in the background using subtle shading so they do not obscure or distract from the data items. The mapping of the scale circles can be customised, with a logarithmic scale provided by default, starting with a single access on the outside of the layout and with increasingly high numbers of accesses towards the central point. The colouring of the data items (in this case representing classes) is used to show the type of the accesses. Red represents write accesses, blue represents read accesses, whilst purple represents an item that both reads and writes to the field variable. These colours can also be changed according to user preferences. The ordering of the data items can also be changed between alphabetical ordering, type of access and number of accesses.



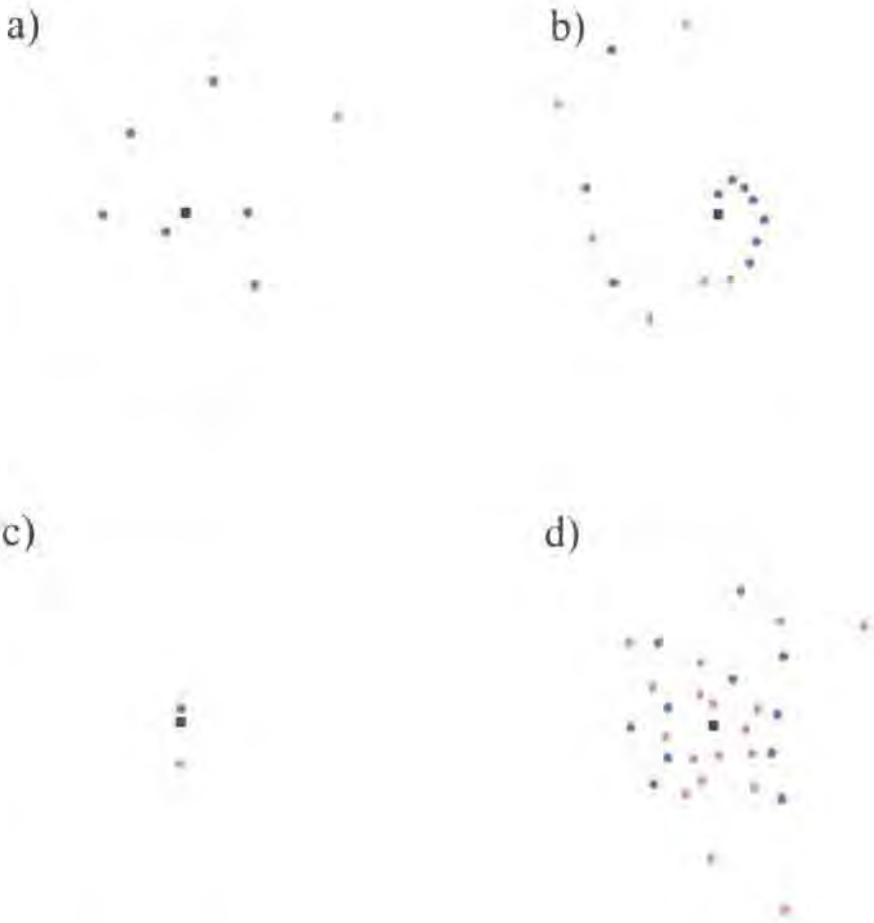
**Figure 5-18 The Variable Watch View showing field access by type and frequency**

Figure 5-18 shows an example of the Variable Watch View showing a field (the black centred square) being accessed by ten different items represented by the small coloured circles. The items are arranged alphabetically, hence the random spread of items in terms of access type and number of accesses. Using this view the user can identify how heavily coupled a field variable is and how the rest of the program uses that variable. For example, in Figure 5-18 it can be seen that field variable under study is accessed by ten items: two of which are write only accesses; three read only accesses; and five read and write accesses. It can be seen that the field is accessed a high number of times especially by the two inner items, while there are three outer items that access it a small number of times.

The view offers a number of abstraction levels as previously mentioned. After the presentation of the accesses at the class level, the user may drill down on one class and see the accesses in terms of the number of methods of that class that access the field. This allows the user to gain a more detailed view of the usage and if more details are needed the view can be drilled down upon again to show the accesses in terms of lines for a single method of a class. When the user uses this drill down mechanism the depth of the current view is presented at the top left of the view above the radial layout. This lists the abstraction in terms of the selected items using a path like structure where the user can select to jump back to previous

levels similar to a web page position bar. For example, "AllClasses \ ClassA \ MethodB" would be displayed when the users had drilled down on ClassA and then on MethodB. Field variables that are private will only be accessed by the local class, and therefore the first level of abstraction is unnecessary, although it is still provided for consistency.

The items in the view can be displayed with or without text labels, depending on user preference. The view also supports more detailed information using mouse over, which presents the name of the item and details of the accesses. This information varies depending on the abstraction level being displayed. At the class level the popup information displays the class name, the number of access and the number of methods accessing the variable, whilst at the individual line level the actual line of the source code is presented, along with two lines before and after to provide context.



**Figure 5-19 Multiple patterns in the Variable Watch View**

Figure 5-19 illustrates some of the patterns that could be found using the Variable Watch View and the possible interpretation of these can be found below in Table 5-5.

Pattern	Possible Interpretation
a)	In this case the field is read often by six items. However, it is only set by one item and its value has been set only a small number of times in the previous execution.
b)	This offers a more complex case of pattern a) with more classes involved and some mixed read and write accesses. In this case the items are ordered by the number of accesses, which

	<p>results in the spiral pattern present. The accesses seem to be in three groups. One group of items that all read from the field a large number of times along with two that both read and write. The next group of four items access the field a medium number of times with two reading and two both reading and writing to the field. Finally, there is a group of three items that only access the field once or twice. The field variable is used by a number of items and therefore could prove problematic if it needs its type modifying. However, if the user is trying to trace down a bug in which the value is being set incorrectly, then this view would be useful as there are only six items that update the field as most access are just read accesses.</p>
c)	<p>Pattern c) illustrates a situation where there is one item reading the field a large number of times and one item writing to the variable slightly less times. If this view were at the class level then there would be two classes involved, one of which could be local class that the field belongs to. This field is therefore in a high state of change as the writing accesses are high. This pattern could occur for a number of reasons such as a producer / consumer situation or maybe in a data structure class that is updated by one class and read from another. If the view was at the method level (showing the class that contains the field) then this would show one method for write access and one for read access which could suggest a <i>GetField()</i> and <i>SetField()</i> method. This pattern, could also have a number of writes from constructor functions. At either level of abstraction the view suggests that the field is not highly coupled to the rest of the program, though this would need further investigation at the class level to make sure the accesses are not widespread through the classes methods.</p>
d)	<p>This would represent a highly accessed field that is extensively used by classes or a class' methods depending on the abstraction level shown by the view. Either way it represents a highly coupled variable that could prove problematic, if it needed modifying. Its usage is typically mixed, with most items showing both read and write accesses.</p>

**Table 5-5 Possible interpretations of patterns from Figure 5-19**

The Variable Watch View can also utilise the method lines concept from the Class View. This allows the user to get an impression of the number of methods involved in the access for a particular class while still being at the class abstraction level. At this point the class doing the accessing is represented as a small circle with the distance from the field being watched representing the number of accesses. For these classes, the number of methods accessing the field can be indicated using the method lines idea, with the length representing the percentage of accesses for that method out of the total for the class. Figure 5-20 shows a simple example of this, with the view showing class accesses using the small circles which then have method lines to give an impression of the methods of that class involved in the access.



**Figure 5-20 Displaying class and method accesses through the use of method lines**

Figure 5-20 shows the method lines and how they can be used to gain an impression of the method usage across multiple classes without the need to drill down on a single class. The method lines are shaded using the same colour coding as the class circles so the user can tell the type of accesses in that method. This additional method line detail can be turned on and off at user preference and may need to be off when there are a large number of classes accessing a field in order to reduce visual complexity and prevent the occlusion of other classes. As Figure 5-20 shows, the method lines allow the user to easily see if the accesses are restricted to a few methods of the class, or whether the field is widely used by multiple methods in the class.

The view offers support for showing variable accesses up to the current program execution point from the creation of the field or from a user specified point in the program's execution. This allows the user to view the entire history of the field accesses, or just the accesses for some particular part of the execution, for example, accesses in a piece of functionality that is of interest to the user.

## Summary

The Variable Watch View displays information on the use of field variables. The view can be used to gain information on the frequency and location of field variable accesses at various levels of abstraction. The view uses a radial layout with a high number of accesses towards the centre. The view presents information on:

- Field Access
  - Type (read, write, mixed)
  - Number of accesses
  - Accessed by Class, method or line.

This information is only available on fields that are explicitly watched.

#### 5.4.2.4 The Method Pixel View

##### Aim

The aim of this visualisation is to show the calling information in a form other than presenting a call graph. This visualisation aims to highlight which methods of a class have been called so far in the program's execution. Not only should it be easy to see which methods have been called, but it should be possible to see how many different classes or methods call that particular method. Therefore, it should be possible to see methods that are localised and called only for one particular purpose, or at one particular point. Classes and methods that are called by many different methods could indicate to the user the use of the class, suggesting it may be a common utility class or data structure class. This information could also be useful for impact analysis tasks as it would show the possible repercussions of changing the class or methods in terms of related classes that may be affected. However, care would have to be taken with this, as the visualisation only shows the calls for the current execution and does not show other calls that have not yet occurred. The visualisation should also provide a summary of the calling relationships for the class, showing which classes it calls methods of and also which classes call its methods.

##### Mapping

The approach taken to show this information is a pixel based technique. Each method is represented as a line, which is colour coded on its access rights using the same scheme as the Class View. Above this line each method that is called by this method is represented as a single "pixel" (actually more than one screen pixel). This gives an indication of the number of methods called. Similarly, each method that calls the current method is represented by a pixel below the line. This allows a large number of methods to be presented and gives an indication of the calling relationships of the methods. The user can zoom in upon methods that are of particular interest to see the actual methods involved or the user can use a mouse over popup window to find the name of the method that the pixel represents. A multiple focus fish-eye technique could be applied here if desired to allow multiple methods to be focused on.



Figure 5-21 Methods of the GraphDesktop class shown in the Method Pixel View

Figure 5-21 demonstrates the view on the GraphDesktop class. It can be seen that the class has eleven methods, all of which are public except for three. Of these three methods one is private whilst the other

two are shown as white as they have no specified access rights (in this case they are compiler generated methods). These compiler-generated methods can be filtered out the case where the JVM supports identification of these synthetic methods. The pixel representing the methods are coloured according to whether they are from code that is being traced (user code), or whether they are calls from code that is not being traced (system code). Those from user code are coloured turquoise, while those from system code are purple. Typically, only code such as interface handlers will be called from system code. The layout of the methods is initially in the order they are defined in the class file, which is the same ordering as used by the Class View. However, they could be ordered by user criteria, for example, alphabetically, or by the number of times the method is called. This view makes it easy to identify which methods have been called, whilst also providing an impression of how widely the methods are used and how many methods they rely on to perform their task.

Figure 5-21 shows the calling relationships for the methods of a class. However, the Method Pixel View also shows summary information for the class using the same representation. Here the line becomes a box to make it distinct from the methods. The pixels represent a class that calls, or is called by the current class. This view allows the user to see how the class interacts with other classes and how widely it is used. The class calling information is displayed at the top of the view so it is not confused as being in the method list. The view can also show the calling at the class level for all of the methods. Here, the “pixels” represent classes, so it shows how many classes call a method and how many classes it uses. This offers a slightly higher level of abstraction than showing method to method relationships.

The Method Pixel View shows the methods that the currently selected class implements. However, this is only shown in terms of the methods defined in that class and not methods that it inherits. Therefore, support for browsing inherited methods is provided by having an inheritance section at the bottom of the view. This allows the user to explore the inheritance hierarchy whilst investigating the methods using the Method Pixel View representation. The parent type of the class is presented along with known sub types of the current class. These can be selected to become the current focus. The view also supports the interaction with other views. A method of interest can be highlighted in the Class View and have its source code displayed in the source code browser.

## Summary

The Method Pixel View displays details of the calling relationships of a class and its methods. It allows the user to investigate these relationships and uses a pixel based representation rather than a call graph. This allows the overall picture of a class' method calling relationships to be presented without significant display space requirements. The view presents information on:

- Classes whose methods this class calls.
- Classes which call this class' methods.
- Methods called by a method
- Methods which call a method.
- Type of method, including name, access type, return type and arguments.

The view presents information on the currently selected class in the Class View.

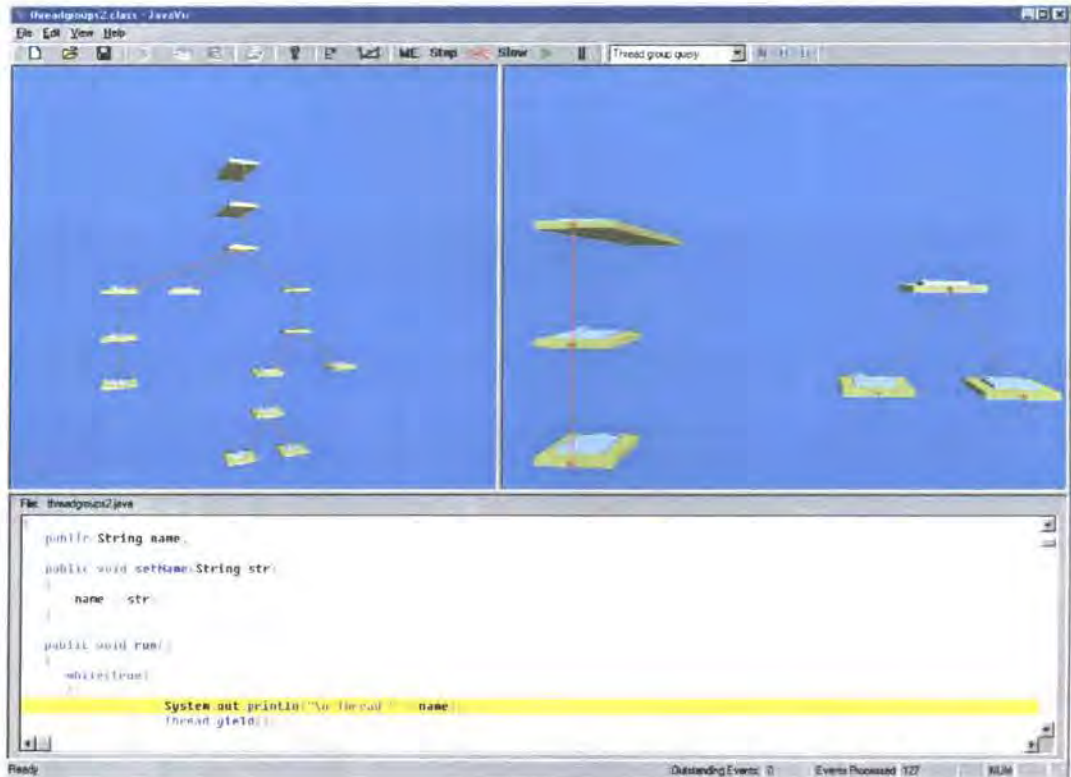
#### **5.4.2.5 The Query View**

##### **Aim**

The aim of the Query View is to allow the user to build up custom views based on the visualisations used in other views. The aim of this approach is to provide the user with an ability to store information relevant to their task in a way that has meaning to them. This is supported by allowing the user full control over the inclusion and placing of items and by providing ease of storing and displaying multiple concurrent views. This view is different to the other four views in that it does not have any specific representations that define it.

##### **Mapping**

In order to allow for flexibility of querying and the need to preserve or watch objects over a period of time, multiple objects can exist in the query view at once. This is supported in two ways. Firstly, a single query view can show multiple objects at once and secondly, multiple query views can exist at once. The user can create, clone, name and delete query views. A drop down list of available query views is used to allow easy switching between views and the preservation of useful views for future use. To support this, the view allows the user to create, name and delete query views or individual objects from within the view. Items can be dragged and dropped into the view from other views, providing a quick and consistent method to add and position items. The mapping of information to representations for the Query View is dependent on the views it is integrated with. The Query View provides a 3 dimensional space to place objects within. This allows objects from other 3 dimensional views to be simply replicated in the view with no changes in mapping in order to preserve consistency. However, the Query View can also support drag and drop actions from other views, such as explorer style tree controls and lists. The mappings for information from these views are described in this section, as these views themselves have no graphical representations.

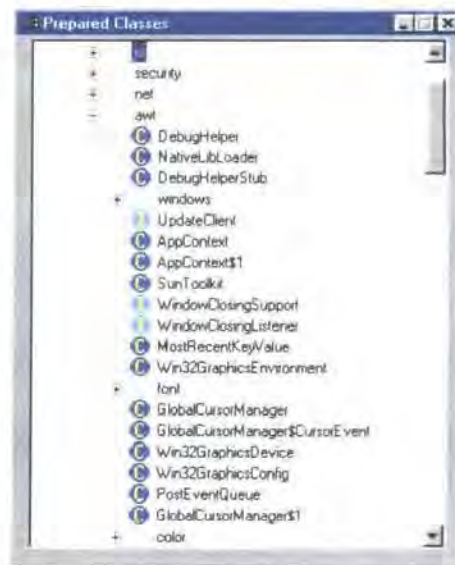


**Figure 5-22 Showing two thread groups in the Query View**

Figure 5-22 demonstrates the use of the Query View. The view on the left of the main window is Runtime View displaying the entire thread group hierarchy of the program under study. To the right of the main window is the Query View with the drop down list above the view showing the current query, called "Thread group query". The Query View shows two thread groups, which have been dragged and dropped from the Runtime View. The dragged thread groups are also showing their child thread groups. These items are synchronised with the Runtime View items. This allows thread groups to be compared side by side regardless of their position within the Runtime View, the same principle can be applied to view threads or any other objects that can be dragged into the Query View.

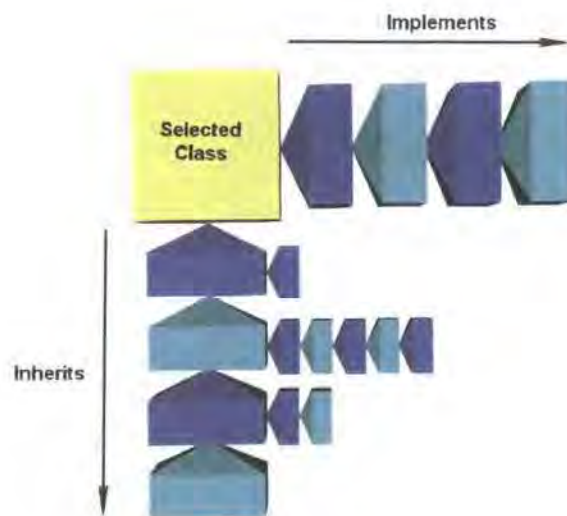
The Class View provides a tree control of loaded classes, with the tree structure based on the package structure of the classes, as shown in Figure 5-23.





**Figure 5-23 Displaying loaded classes**

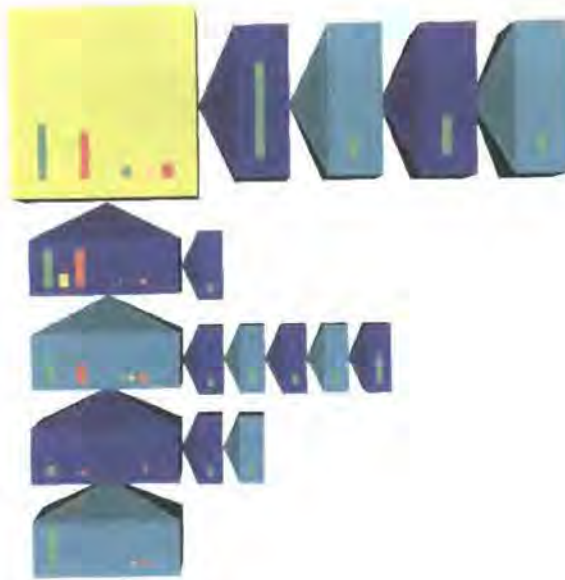
This view, supports the Class View and the classes are simply represented as their name with an icon to show whether they are a class, interface or an array. These items can be dragged into the Query View. Once dropped into the Query View the visualisation aims to highlight the structure of the class, in terms of the classes it inherits, the interfaces it implements and the methods that each of these provides.



**Figure 5-24 Visualising the structure of a class in the Query View**

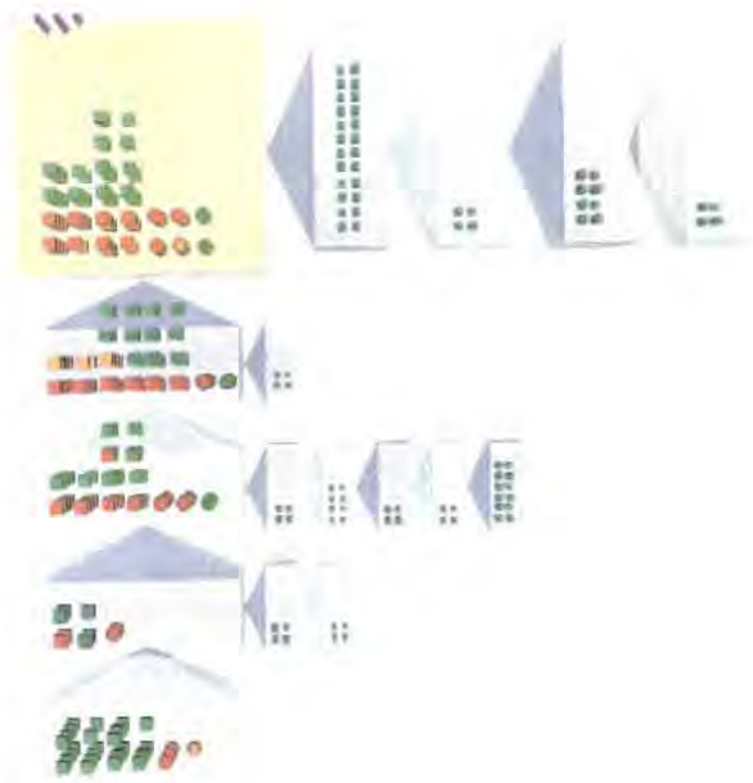
Figure 5-24 shows the structure of the representation of a class that has been dropped into the Query View. The class of interest is shown as a yellow box. Any interfaces that it implements are shown to the right of this box, so the class in Figure 5-24 implements four interfaces. The inheritance structure of the class is shown similarly below the selected class. These classes can also implement interfaces and these are shown to the right of each class. These interface items are scaled down from the selected class' interfaces in order to prevent them dominating the display and distracting the user from the selected class. This structure therefore provides a complete overview of the inherits and implements relationships of the selected class. Initially, the simple structure, as shown in Figure 5-24, is presented to the user. However, this does not allow the user to gain an impression of the size or structure of the classes and interfaces

themselves. Therefore, the view presents two further levels of abstraction on the details of the classes as the user focuses on the structure.



**Figure 5-25 Showing an overview of the methods and fields**

As the user focuses on the class, a summary of the methods and fields of each of the classes and interfaces is presented, as Figure 5-25 shows. Here, the structure is presented and additional information fades in as the user focuses on the object. Each class now has a number of columns and circles on their surface. The three columns to the left of the class represent the number of methods in the class, shown as public (green), protected (orange) and private (red). The height of the column represents the number of methods of that type. Similarly, the three circles represent the fields of each type (public, protected and private) using the same colouring and with the size of the circle representing the number of fields of each type. The heights of the columns and the size of the circles are scaled to prevent them being too large to place on the surfaces. The same scale factor is used for the whole structure in order to allow the user to compare between classes and interfaces. This level of abstraction provides an overview of the number of methods and fields that combine to constitute the selected class. The user can gain greater detail on the methods and fields through focusing further onto the selected class. As the user moves closer to the structure the items fade to become transparent and reveal details within each item as Figure 5-26 demonstrates.



**Figure 5-26 Detailed view of a class in the Query View**

Figure 5-26 shows the full details of the class shown within Figure 5-25. Individual methods and fields are shown as boxes and spheres respectively, using the same colour coding as the overview columns and circles. Additionally, a number of cones are shown on top of the selected class, which represent classes that inherit from the selected class. The user can use this view to navigate to items within other views, such as the Class View or the source code browser and to set up variable watches in the Variable Watch View. Additionally, information on the items is provided through user selection or through mouse over. This provides a pop window with further information, such as showing the type of a method and the lines of source code where it is defined. The user can also select any class or interface within the structure and make it the selected class if they desire.

The Query View provides an easy way of querying and allows objects that are from different views or locations to be placed together for the user to inspect. This can help overcome problems of layout and the need to keep switching between views or locations within a view. However, as a consequence the context of an item is lost. This offers the advantage of filtering the information being presented, so only the item of interest is shown. However, the user then cannot see how an item fits into the bigger picture, or even where that item came from. To alleviate this, detailed labelling is presented at user request and an option to locate the item in the view of origin exists. This, along with the ability to highlight the selected item in all views, aids in presenting the relationship from Query View items back to their original context.

## Summary

The Query View provides a mechanism for the user to build up custom views. It is designed to complement the other views, for example, it allows the user to place specific items from the Runtime

View in a query, therefore allowing the user to examine them closely whilst still being able to see an overview in the Runtime View. The view allows information to be dragged and dropped into a query. This information is then displayed using the representation from the original view when applicable. For example, items for the Runtime View are shown using the same representation as in the Runtime View. The view also supports information from some other views that do have a graphical representation, for instance from the list of loaded classes in the Class View. The Query View allows a number of queries to exist at once and offers facilities for the creation, deletion, duplication and naming of queries.

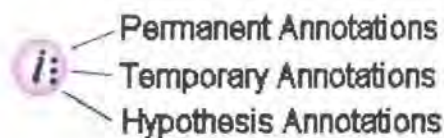
### 5.4.3 General Features of DJVis

DJVis provides a number of facilities across all, or multiple views and these are discussed in this section. The visualisation allows items to be annotated with user comments. This can be done in all views and the annotated comments are shared between the views. User annotations are provided to aid a number of tasks. Permanent annotations provide useful documentation on the software item that can be used in other investigations or by another user. These records act as known facts about an item of the software. However, as a user investigates a piece of software they may wish to record temporary notes about an item under investigation. The annotations are highlighted within the visualisation using an icon.



**Figure 5-27 Showing a class has user annotated documents in the Class View**

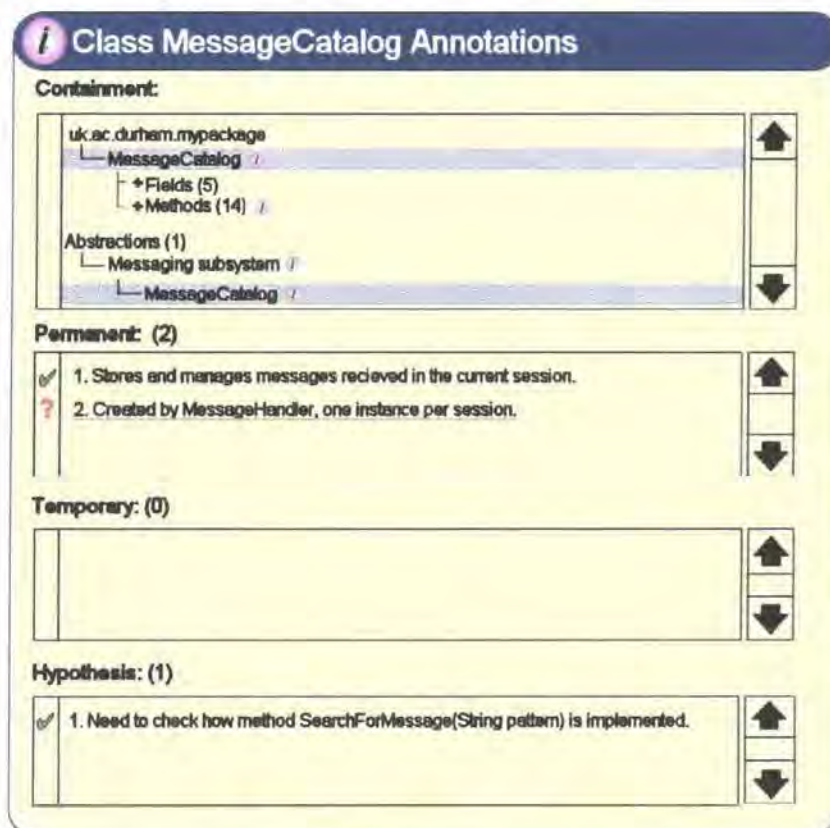
Figure 5-27 shows the class MessageCatalog in the Class View with the user annotated text icon. The annotations support different tasks and this is supported by having three levels of annotations, permanent, temporary and hypothesis. Permanent annotations, as their name suggests, are concrete facts about the item that do not change between different executions of the program. For example, "This class provides an interface to the database." Temporary annotations are comments that the user is not certain of, or that may only apply in some cases. Hypothesis annotations are designed for the user to record temporary ideas and notes on the class while they investigate the software. These three levels can be highlighted on the annotation icon, as Figure 5-28 demonstrates.



**Figure 5-28 Showing annotation levels**

The annotation icons can be filtered from the view so that they are only present for certain levels of annotations or not at all depending on user requirements. The colouring of the icon is different from any other item in the visualisation so as to be easily identifiable. The same icon is used in the Runtime View and Query View to indicate user annotations and clicking on the icon gives access to the details of the

annotations. The annotations are stored in a repository using their full signature as a key (e.g. `java.lang.String`). This allows the annotations to be preserved across multiple executions of the tool whilst also allowing annotations for a class that is used in more than one program to be shared. However, this introduces many issues with regard to the non-static nature of the classes and their source code. Classes may be in a state of modification and the comments may become outdated or incorrect as with all documentation. In order to highlight change, the date and size of the class file could be stored when an annotation is made allowing the visualisation to check them against the current class file. This will allow different versions of a class to be identified and the applicability of the annotations to be questioned. The match status of the annotation is presented in the annotation list, so the user can either update the annotation to apply it to the current version, or ignore it. Figure 5-29 shows the annotation list with the second item in the Permanent list being highlighted as applying to a different version (red underline and question mark). The other annotations apply to this version, as indicated by the green ticks before the annotation text. Some annotations, such as the role of the class may be constant across all versions and these can be marked as non-version dependent.



**Figure 5-29** Displaying annotation details for the class `MessageCatalog`

When displaying the annotations for an item it is also important to place the item in context. Each item, be it a class or abstraction etc. is part of a hierarchy. Packages contain classes, which then have fields and methods. These items can also be part of abstractions. Therefore, the view highlights the selected item in terms of its containment within other items. This is shown in the containment section, at the top of the window. As can be seen from Figure 5-29, the `MessageCatalog` class is part of the package "uk.ac.durham.mypackage" and the "Messaging subsystem" abstraction. The list also highlights which of the related items have user annotations, for example the "Messaging subsystem" abstraction. This allows

the user to inspect other related annotations. The annotation method described here is a simplified handling of the versions and their interactions. It is not the aim of this research to focus on the multiple issues of storing and providing correct documentation across multiple versions. It does not consider multiple users, or constraints on which users can update the annotations, especially permanent annotations. It merely aims to provide a simple way to allow for user annotations, with the aim of aiding users in the comprehension of a system. It could also be beneficial if the tool could be linked to existing documentation, such as being able to display the Javadoc for a class or method.

The DJVis environment provides support for the preservation of previous investigations through the use of a number of techniques. A history of previous actions is recorded in all of the views to allow the user to quickly retrace their steps. The history list can be ordered in a number of ways, such as the time of the change or by the view in which the change occurred. The history events are recorded for most views as the point at which the view is modified. However, the different representations and navigation methods of the views present unique differences. It must be noted that unlike traditional history steps the state of the view is only preserved in terms of the navigation state and not the data. This is due to the constant change of the runtime information, as it would be unfeasible to preserve the state of the program data at each history event. This would also introduce many issues in terms of providing synchronisation between the executing program and the presented past state, as well as the need to highlight to the user that they are seeing a previous state. Instead, the navigation state of the view is recorded. For example, storing the item which is the current focus or the current layout options. The Class View also provides the ability to save states of the view for later inspection.

## **5.5 Conclusions**

This chapter has described the features of the visualisation of Java programs developed in this research. The information that can be useful for understanding the runtime behaviour of a Java program was presented followed by a detailed description of the DJVis visualisation. DJVis is comprised of a number of views, namely the Runtime View, Class View, Variable Watch View, Method Pixel View and the Query View. These enable the majority of the features of the Java language to be visualised. Each of the views and the mappings and representation used by each were shown and discussed. The integration of the views was also discussed as well as global issues that are present across all views.

# Chapter 6 Implementation

## **6.1 Introduction**

This chapter provides an overview of the implementation of the prototype tool in terms of the techniques and tools used. This is split into four sections. Firstly, the method of extracting runtime information is discussed, before then looking at how this information is presented in the visualisation. A summary of the limitations of the prototype tool is then presented, highlighting which parts of the theoretical visualisation have not been implemented. Finally, a look at how the prototype allows for automation and a brief summary of the use of the prototype tool is included.

## **6.2 Information Extraction Method**

Runtime information is extracted using the Java Platform Debugger Architecture (JPDA) [JPDA]. This provides three layered interfaces through which debugging information can be obtained from the Java Virtual Machine. These are as follows:

### **The Java Virtual Machine Debug Interface**

The Java Virtual Machine Debug Interface (JVMDI) is a native interface that is implemented by the Java Virtual Machine. This is a low-level interface which is accessed by writing a native library (JVMDI client) for the JVM to load, this library uses the JVMDI to interface with the JVM and obtain runtime information. The combination of the JVM and this JVMDI client form the back-end. An implementation of this JVMDI client is provided with the Sun JDK distribution (Java 1.3.x onwards) and it uses the Java Debug Wire Protocol to communicate with the application requesting the debugging information.

### **The Java Debug Wire Protocol**

The Java Debug Wire Protocol (JDWP) is not an actual interface, but defines the protocol used in communication between the back-end and the application using the debugging information. The protocol is packet based, although it does not define the transport mechanism. The Sun distribution provides clients for socket based transport and shared memory transport (Win32 platforms only). The JDWP allows local and remote debugging depending on the transport mechanism used.

### **The Java Debug Interface**

The Java Debug Interface (JDI) is a pure Java interface. It provides a Java API through which debugging information can be obtained, using an object-oriented model rather than the packet structure of the JDWP.

The JDWP interface was chosen for the implementation of the DJVis prototype tool, as it allowed access from non-Java code. A C++ wrapper was written using a socket based transport to allow the connection to debugging information from a JVM on a remote or local machine.

## **6.3 DJVis Prototype Implementation**

The prototype tool was written in C++ using the Microsoft Foundation Classes (MFC) to provide the user interface. The MFC offers an object-oriented wrapper for standard Windows controls. These classes were used for implementation of the entire user interface except the 3D views (Runtime and Query) which are



based on MAVERIK embedded within a MFC window. MAVERIK (**MA**nchester **V**irtual **E**nvi**R**onment **I**nterface **K**ernel)<sup>3</sup> is a C toolkit for the creation of virtual environments [Hubb99]. It uses OpenGL for rendering and is freely available for all major platforms. MAVERIK provides a framework in which a virtual environment can be defined and managed, as well as providing spatial management functionality and support for multiple input devices. MAVERIK is made up of two components, a micro-kernel and a set of supporting modules. The micro-kernel provides the framework within which applications can be built. The supporting modules provide a means for customisation and provide techniques for navigation, spatial management and interacting with I/O devices. MAVERIK was used within the prototype tool to provide the Runtime View and the Query View. The MAVERIK source code was modified slightly for the implementation of the prototype tool in order to allow the embedding of the MAVERIK window within the main application frame window.

DJVis provides a number of views each of which offered its own implementation challenges. However a significant amount of the implementation effort also went into the “back-end” to support these views. This includes the network layer to link to the JVM using the JDWP and an event queue where processed events from the JVM are stored and then handled. The distributed concurrent data source of the JVM and the multiple views introduced issues of concurrency that needed to be addressed in this back-end layer as numerous requests for information from the JVM could occur simultaneously. The back-end also acted as a central store for information on the program that was then visualised in the different views.

The implementation of the Runtime View relies on the use of MAVERIK to provide the 3D support. The main data source of the view is reports from the back-end of the creation and termination of threads, together with the call stacks for each of these threads. In order to provide the drag and drop facilities with the Query View, the visual items drawn in Runtime View were constructed from a base object. This stored any item data such as the name, and also contained a list of visual instances of the item. This allowed changes in the underlying data to be reflected in all the visual representations, whether they are in the Runtime View or in one or more queries in the Query View.

The Class View uses the MFC to provide the user interface and provides drop down lists to allow the user to configure which data from the back-end is represented. The details of classes, their methods and their fields are stored centrally. However, the need to represent dynamic references through a base class or interface introduced a specific information request that was not available directly from the JPDA. In order to be able to provide this information the back-end watched for a number of events. Firstly, as classes are loaded by the JVM the class details and inheritance structure are recorded for use in the views. The back-end then requests details of the fields of the class being loaded and these are stored as static class references. However, if any of these reference types refer to classes that are known to have sub-classes, then a write watch is put on that field. The back-end will then receive an event reporting the old and new values whenever the field is updated. As these values are references, the objects and therefore the types of the references can be found. This therefore enables the back-end to determine if the new reference is to an

---

<sup>3</sup> Available for download at <http://aig.cs.man.ac.uk/maverik/>

object of the original field type (i.e. static) or to a sub-class of the field (i.e. dynamic). It is this information that is then used by the Class View to allow dynamic references to be shown.

## **6.4 Limitations**

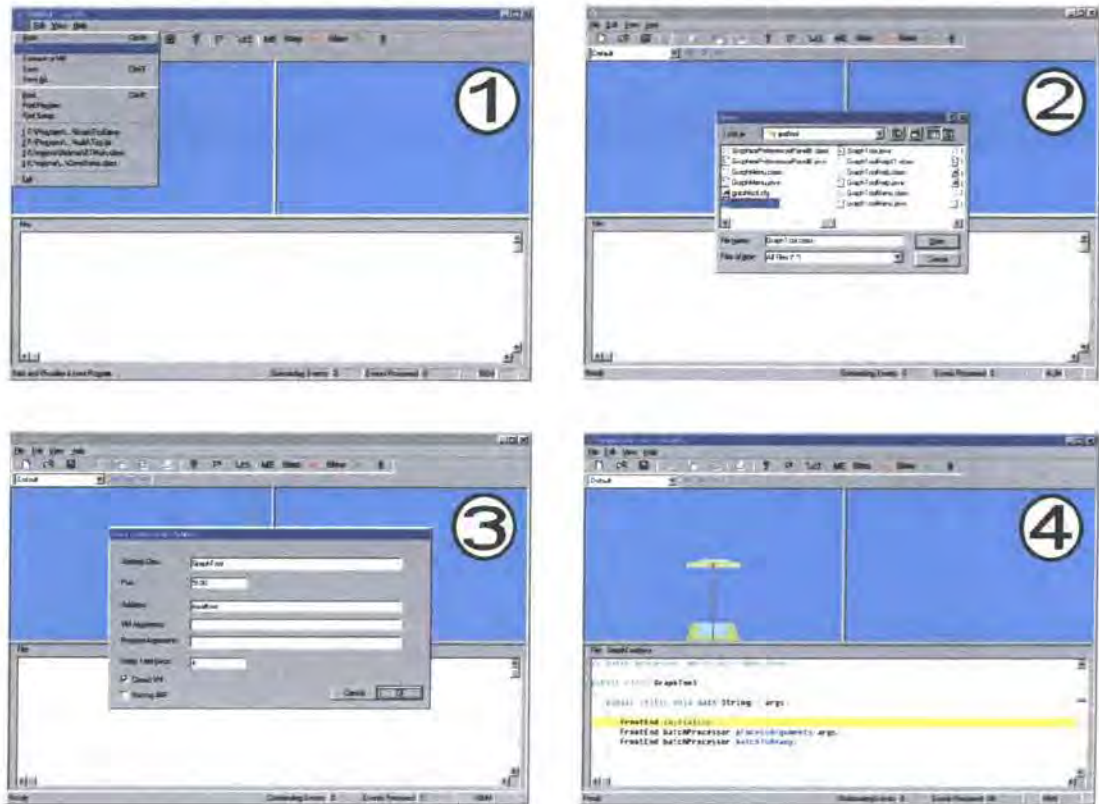
The prototype implementation of DJVis demonstrates the main concepts of the proposed ideas, although due to time constraints not all of the proposed visualisation has been implemented. The main focus of the prototype is to show that the required information can be extracted at runtime without the need to modify the source code of the software and to demonstrate the main concepts in each view. This has allowed the views to be tested on real programs and therefore allowed the concepts to be tested to a much higher degree than simply considering theoretical hand produced visualisations. The time constraints of the research reduced the ability to refine and integrate the views to the extent that would be required for a commercial visualisation tool. However, much of this work would have been using existing user interface elements and would therefore not have benefited the conceptual visualisation research. The implementation of the Runtime and Query Views lack the quality of the designed visualisations. This was due to the time constraints on the implementation and it was felt that the investment in time would be better spent on other areas, such as improving the Class View. The implemented versions of the Runtime and Query Views do however provide the main features and demonstrate that the proposed visualisations can be feasibly produced at runtime.

The Class View uses a graph representation for which a springs style layout algorithm was used. Using this, each edge is modelled as a spring, which has a desired length. The layout algorithm attracts or repels nodes in order to try and maintain the desired edge length. This algorithm gave acceptable results, although it is recognised that other algorithms may have produced more aesthetic layouts. It is not the aim of this work to assess or propose new graph layout algorithms, therefore only the one general algorithm was implemented for the Class View. Specialised graph layouts could be included for relationship types that have specific characteristics. For example, the inherits relationship will produce a tree and therefore a tree layout may be the most beneficial for this instance. The abstraction method for the Class View is only partly implemented. It allows nodes to be grouped and their edges are updated accordingly. However, the abstraction does not allow its contents to be expanded, nor does it allow itself to be removed once created. These improvements are not technically difficult and could be achieved fairly easily, however the time was needed to implement other views as the tool aimed to provide a sample of each view rather than just a refined version of one view. Unfortunately, there was not enough time to implement the annotation and history elements of the visualisation.

## **6.5 Use of the Prototype Tool**

The prototype tool has been designed to be straightforward to use when generating a visualisation, as Figure 6-1 demonstrates. Firstly, the user selects to start a new session (sub image 1). The user can then choose the class file (or corresponding Java file) to start using a file selector (sub image 2). A dialog box is then presented allowing arguments to be set for the program as well as allowing arguments for the Java Virtual Machine (JVM) to be entered (sub image 3). This allows for the setting of class paths for example. These settings could easily be saved to a settings file for that particular program, although this is

currently unimplemented. Once the user has specified any options they require, a Java session is spawned using the given arguments and those needed by the Java Platform Debugging Architecture (JPDA) which are automatically added by the tool. The tool then connects to the JVM using a socket connection to the JPDA. The program is suspended at the start of its execution. The starting view presents the Runtime View, Query View and the source code of the program (sub image 4), with other views accessible via the tool bar. The execution of the program can be controlled through the tool bar with options to run, pause and step to the next line or method call.



**Figure 6-1** Generating a visualisation within the prototype tool

The prototype can also connect to a JVM running on another machine provided that the Virtual Machine was started with the required Java Platform Debugger Architecture options. The remote session must be started first using the following JPDA arguments, where *classFile* is the class file containing the *main()* method of the program that is to be visualised.

```
java -Xdebug -Xnoagent -Djava.compiler=NONE
-Xrunjdwp:transport=dt_socket,server=y,address=5100 classFile
```

Once the Java session has been started then the tool uses the “connect to remote VM” option and specifies the address and port number to connect to. Once connected the visualisation is presented in exactly the same manner as a local session. The source code will be unavailable, although an option to specify the path of a local copy (or mapped network drive) of the source files could easily be added to the implementation. The tool can also be used to visualise Java applets provided the debugging options can be specified and the JVM used supports the JPDA. For example, applets can be visualised using the Sun

Java Plug-in which can be used as the Java Runtime Environment for Netscape Navigator or Microsoft Internet Explorer.

This section demonstrates the ease of use of the prototype tool. The prototype allows visualisation sessions to be started quickly and with no user intervention, other than requiring the same options that would be needed to run the program from the command prompt in the standard manner.

## **6.6 Conclusions**

This chapter has highlighted the implementation techniques used in the realisation of the DJVis prototype tool. It presented the methods used to extract the runtime information and to display the views. The limitations of the current prototype are highlighted, listing elements of the conceptual visualisation that were not implemented due to time constraints. The chapter also showed how the prototype tool is used to visualise Java programs with a storyboard demonstrating its ease of use.

# Chapter 7 Evaluation Approach

## 7.1 Introduction

This chapter presents an overview of software visualisation evaluation techniques and discusses them in regard to evaluating this research. The actual approach to be used to evaluate this research is then proposed and justified.

## 7.2 Evaluation Techniques

Many techniques exist that can be used to evaluate a software visualisation. However, despite these possibilities little research has been done in the area and the application of evaluation techniques is usually limited. Evaluation is often seen as unattractive and most of the research effort in the software visualisation field is focused on new representations and visualisations. This section aims to give an overview of the different techniques that can be used and it is based on a summary from Hatch et al. [Hatc01]. A number of techniques will be summarised, namely: design guidelines; feature based frameworks; scenarios and walkthroughs; and user and empirical studies.

Design guidelines and frameworks offer "rules of thumb" for assessing the quality of a visualisation. These guidelines are by their very nature generic and are aimed at providing discussion points and quick checks rather than any in depth evaluation. A commonly referenced design guideline is that of Shneiderman [Shne96]. He proposes seven tasks that information visualisation tools should support, these are shown in Table 7-1.

Task	Summary
<b>Overview</b>	Gain an overview of the entire collection.
<b>Zoom</b>	Zoom in on items of interest.
<b>Filter</b>	Filter out uninteresting items.
<b>Details-on-demand</b>	Select an item or group and get details when needed.
<b>Relate</b>	View relationships among items.
<b>History</b>	Keep a history of actions to support undo, replay and progressive refinement.
<b>Extract</b>	Allow extraction of subcollections and of the query parameters.

Table 7-1 Shneiderman's seven tasks [Shne96]

He also summarises the central principle of a visual-information-seeking mantra: "*Overview first, zoom and filter, then details on demand*" [Shne96]. Such design guidelines offer an aid for the design of visualisations, although, they are very generalised. They are not prescriptive and do not offer guidelines on which representations to use. Therefore, it can be easy to satisfy the guidelines, regardless of the effectiveness or appropriateness of the method used. Design guidelines are therefore not useful as an evaluation themselves, but their broad requirements could be included into a more in depth evaluation.

Feature based evaluation frameworks offer a more in depth evaluation approach, yet one that can be applied with ease. Kitchenham and Jones [Kitc96] offer a detailed summary of feature based evaluations. Feature based frameworks are advantageous as they require no prerequisites on infrastructure and they

allow multiple items to be compared. They consist of a number of questions that are asked of each item under study. However, despite appearing to be simple, they offer a number of challenges that must be addressed in order to achieve a successful and useful evaluation. One of the most important issues is the structure of the questions. Questions can be answered in terms of Yes/No criteria, however, this can be too restrictive and problems can arise if the item under study only partly fulfils the criteria. Conversely, questions may be answered on a sliding or numerical scale. This approach offers more flexibility in the answer, but then it is also more subjective with different evaluators producing different evaluations. Numerical scales also have the danger of being misconstrued and simply added together to give a simple numerical result in order to compare systems. Such an approach hides the details of the evaluations and thus the failure of an item to provide some core feature may be hidden by its ability to provide other less important functionalities. Often questions may be contradictory, such as "*high information content*" and "*low complexity*" from Young and Munro's framework [Youn99]. These questions make perfect sense and it should not be the aim of the item under study to get "full marks" on all questions. Some aspects of a visualisation are a trade-off, as the previous example shows. Much effort is needed on the design of the framework questions and problems will always exist with the subjective nature of their answers. Often the evaluator's experience or preference can bias the results of the frameworks. An example of this is when previous experience of the representation or navigation method used in a visualisation may prove advantageous over a visualisation where the evaluator has no experience and finds the visualisation more difficult to use, as they are not familiar with the methods used. Therefore, there is a need to invest learning time before the evaluation, when the evaluator has no previous experience of the item under study. However, feature based frameworks offer a valuable aid in evaluating visualisations and provide an approach that is typically not dependent on any specific resources, allowing them to be applied in most situations.

Scenarios and walkthroughs offer an approach that demonstrates how the visualisation is used and is therefore useful, in more practical terms. It is not a traditional evaluation method and instead puts the role of evaluating the idea to the reader of the scenario/walkthrough. The reader is presented with how the visualisation can be used for certain tasks and from this they can decide, if it is useful based on their needs and preferences. Scenarios and walkthrough should therefore not be the only evaluation approach, but used for demonstrating features of a visualisation, as used by Chi et al. [Chi98], or alongside other evaluation techniques such as used by Knight [Knig00]. The use of scenarios and walkthroughs has the advantage of being able to demonstrate the visualisation in more concrete terms, therefore allowing the reader to get a greater sense of its use, whilst not needing to have a prototype tool to test. However, care must be taken when assessing a visualisation based solely on a scenario or walkthrough, as the chosen example will probably only show the positive aspects of the tool and use favourable data.

The most in depth evaluations are user and empirical evaluations. Empirical studies attempt to quantify the benefits of the visualisation and provide hard evidence about some hypotheses. A limited number of evaluations of visualisations have been carried out and these studies typically measure the time to complete a particular task using the visualisation, for example, search, count and compare data tasks in studies by Wiss and Carr [Wiss99]. User studies use questionnaires and user observations to assess the visualisation and gain individual and collective impressions of it. Work by Storey [Stor98] has been one

such application of this within the software visualisation field, based on evaluating the SHRIMP tool alongside the existing RIGI tool. This approach can highlight usability issues that may have been overlooked by other methods. However, care must be taken not to overgeneralise individual experiences. User and empirical studies offer the most comprehensive evaluations and can help to overcome the problems of self-evaluation. However, they are also the most costly in terms of resources. A suitable group of subjects is needed who are familiar with the tasks being analysed and they must be given sufficient learning time, in order not to bias the evaluation towards the current known approach. Studies need significant planning, if they are to be successful and a significant implementation of the visualisation is needed to allow for the evaluation. These large demands on resources make user and empirical studies unfeasible or unattractive for some software visualisation research, especially short-term projects.

The task of choosing a technique to evaluate a visualisation by is not an easy one. As this section shows there are many techniques each with their own advantages and disadvantages. However, despite the numerous techniques there is typically limited evaluation of ideas within software visualisation. Very little published software visualisation material focuses on or shows the evaluation of the ideas and even “accepted” visualisations and representations have typically only been accepted through community acceptance and by being reused or referenced, rather than any formal evaluation of their quality. Some visualisations and tools may be evaluated internally by an organisation, but if this does occur the results are not published. Software visualisation is still in its infancy as a field and has yet to show its true promise. However, for it to become accepted a greater focus is needed on demonstrating the advantages of visualisation rather than how nice the pictures look. There needs to be more concrete support for visualisation in general, if it is to be accepted in an industrial context. The small size of the research field also produces fragmentation of focus and ideas. The diversity of ideas is beneficial, but each visualisation is designed for a slightly or widely different task. This makes it very difficult to compare two systems against each other.

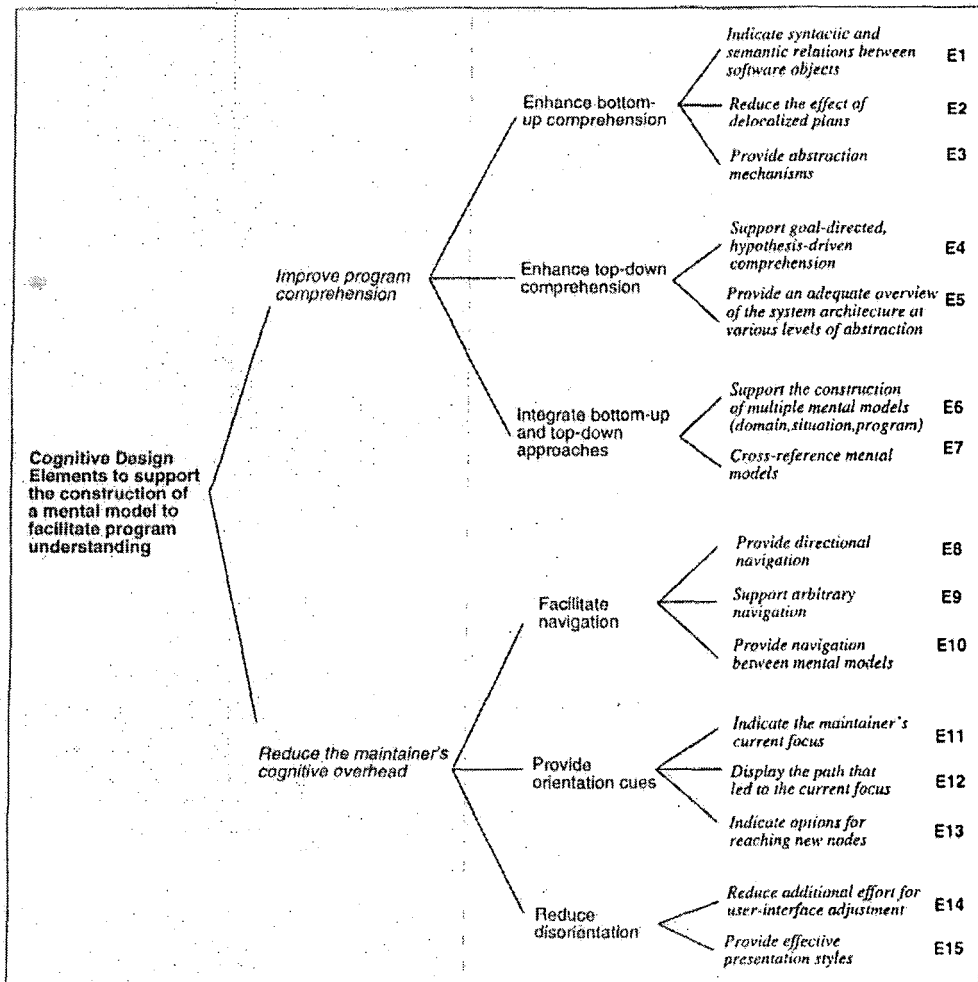
Work on evaluation needs to focus on two main areas, the quality of the visualisation and its suitability for the purpose for which it has been designed. Authors such as Knight [Knig01] have highlighted the importance of the suitability of the visualisation for the task. However, the diverse application of visualisation techniques to different software engineering tasks means that there are only generic frameworks focusing on higher level support for general tasks. One of the most task specific frameworks is that of Storey et al. [Stor97a] for the cognitive design elements for software exploration tools. This assesses the general task of software comprehension yet there are obviously sub tasks within this, which may have specific requirements or demands on a visualisation. Currently, software visualisation is not perceived as advantageous by the very people that it is trying to help, i.e. those working with source code. This could be due to a number of reasons, such as their lack of knowledge of software visualisation, the lack of good visualisation systems, or the resistance of people to move away from the code they are familiar with especially as they are typically under time pressures to complete tasks. Therefore, there needs to be work done in order to overcome such issues. This may be through the incorporation of visualisations into programming environments to reduce the separation between it and the rest of the development process, or through greater emphasis on the adoption of ideas into full products. This problem is aggregated by the lack of industrial visualisation development. The academic software



visualisation field focuses mainly on new ideas and rarely has the resources, or even desire, to turn these into commercial quality tools. However, there are currently very few software visualisation companies, due to its lack of acceptance and therefore the ideas are rarely transferred to industry or even tried in an industrial context. One approach to this may be the industry-as-laboratory approach proposed by Potts [Pott93], however, the lack of any perceived advantages makes it difficult to find industrial links willing to invest and risk the adoption of visualisation techniques. Software visualisation needs to address these issues as it matures and provide a more theoretical background for the development of new ideas, rather than being an art form as it is now. This current situation makes it difficult to obtain a detailed evaluation of a visualisation and to fully assess the relative worth of new ideas.

### **7.3 Chosen Evaluation Approach**

This section outlines the approach taken in choosing the evaluation method for DJVis. A number of evaluation techniques could be used as the previous section highlights. However, a feature based evaluation framework was decided upon for a number of reasons. This allows the visualisation to be evaluated without significant cost overheads in terms of time and facilities and in a manner that will allow others to compare different visualisations when evaluating against the same framework. This evaluation does not aim to provide a quantitative measure on the quality of the visualisation but simply highlight the support for beneficial features identified within the program comprehension domain and provide scope for future refinement. The application of a user study would have been unfeasible given the time constraints. In particular, significant development investment would have been required to make the prototype tool fully featured and refined enough to prevent the evaluation focusing on the issues of the implementation, rather than the issues of the underlying visualisation concepts. An existing framework was chosen, in order to prevent the need to design a custom framework which could introduce the problem of bias. There are no frameworks that focus specifically on the dynamic issues that exist in a runtime visualisation system and therefore a generic software visualisation framework was chosen. In fact, DJVis will be evaluated against two frameworks, namely one by Storey et al. [Stor97a] and one by Knight [Knig00].



**Figure 7-1 Cognitive Design Elements for Software Exploration [Stor97a]**

Storey et al. [Stor97a] define a hierarchy of cognitive issues which should be considered during the design of software exploration tools (Figure 7-1). Software exploration tools provide graphical representation of software structures linked to textual views of the program source code and documentation, therefore software visualisation tools are included in this definition. The first branch of the hierarchy deals with issues on the cognitive theories of program comprehension, whilst the second branch addresses factors to reduce the cognitive load on the maintainer when they are browsing and navigating the software. Program comprehension strategies vary due to a number of factors such as program and task; therefore tools supporting maintenance should support a wide variety of comprehension strategies. Storey et al. suggest that bottom-up comprehension can be supported by allowing immediate access to the lowest level of the program (typically the source code) and by showing relations between software objects, indicating both the syntactic and semantic relations (E1). The tool should also reduce the effect of delocalised plans (E2), which result from the fragmentation of source code related to a particular plan or algorithm [Stor97a]. They can result in disorientation for the maintainer, as frequent switching is required between source files. Storey et al. argue that the tool should also provide abstraction methods to support the difficult task of building hierarchical abstraction from low-level details (E3). They suggest that maintainers may understand the software better when they have created the abstractions themselves, through aggregating low-level objects in to higher level abstractions which are labelled according to the maintainers understanding of them.



Few systems currently support top-down comprehension by allowing goal directed hypothesis-driven approaches. Such top-down comprehension requires either knowledge of the program's domain, or previous knowledge of the program, whether through experience of its code or documentation. Storey et al. argue that tools should provide methods to record hypotheses, linking them to the program, whilst allowing them to be refined (E4). This should be combined with overview presentations of the program at various levels of abstraction allowing the system to be explored top-down (E5). The framework suggests that integrated program comprehension methods [Mayr95] should be supported through the integration of bottom-up and top-down approaches. There should be support for several linked views representing a variety of cross-referenced mental models (E6 and E7). This is due to the variety of mental models used by maintainers and the frequent switching between them.

Storey et al. argue that there is a large cognitive overhead when comprehending a large system, but that this overhead can be reduced by providing good navigation facilitates, orientation cues and the effective presentation of information. Navigation can be directional i.e. mechanisms for sequential navigation, data and control flow navigation and navigating using hierarchical abstractions (E8). Alternatively, arbitrary navigation allows maintainers to reach locations without following specific navigation routes defined by the system (E9). Examples of arbitrary navigation include searching and saving views (allowing the user to store arbitrary steps). Navigation also occurs between mental models and support for this should be provided (E10). Orientation cues allow the maintainer to see what they are currently looking at and how and why they got there. Areas of interest in the code may often be fragments e.g. delocalised plans and therefore cues to indicate current focus are needed (E11), especially as operations such as searching can take the maintainer to distributed locations. The history of the navigation should also be provided to show how the current location was reached (E12). Storey et al. point out that few systems allow temporary information to be recorded, such as the maintainer's current thoughts on an object of interest. This can be useful for aiding hypothesis verification and recording which sections of the code need modifying in some way. The context of the maintainer's current focus should be presented thus allowing them to observe its position in the larger picture (E11) and new locations that can be navigated to should be presented (E13). Storey et al. suggest that disorientation can be a major problem for visualisation tools especially when switching between different views and mental models. Discontinuity in these changes between views and mental models can result in disorientation. Therefore, the tool should reduce the effort of user-interface adjustment by having well designed interfaces that display only relevant functionality (E14). They also argue the need for effective presentation styles, in order to reduce the visual complexity of the presentation (E15). This is especially important for visualising large systems where there will be a large amount of information to present.

The framework by Storey et al. identifies many important issues for exploration tools. They identify the need for research into navigation methods, encompassing orientation cues and presentation styles in order to allow large systems to be handled. They also identify the need for providing more support for mapping domain knowledge to code and for switching between mental models.

As can be seen from Figure 7-1 the framework focuses on evaluating a tool against features known to support program comprehension tasks and it has limited emphasis on the evaluation of the actual visualisation and representations used. It is for this reason that a second framework is also used which addresses more visualisation related aspects of the evaluation. This framework, which is defined by Knight [Knig00], is based on that of Storey et al. and work by Young and Munro [Youn98] and Globus and Raible [Glob94]. Due to its inclusions of the Storey et al. framework, there is an overlap between the two frameworks, especially on issues to support program comprehension strategies. Therefore, any duplicated questions will only be addressed once in the original Storey et al. framework evaluation. Repeated questions in the Knight framework will simply record the result and refer back the Storey et al. framework for the description of the evaluation. Knight's framework is split into three sections: visualisation features; comprehension features; and features that are applicable to both. Knight proposes this separation in order to allow the different concerns of the visualisation to be addressed in the evaluation. For example, it is no good having a good visualisation and representation, if it does not support the program comprehension tasks that the user requires it to perform. The questions posed by the Knight framework are:

#### **Visualisation Features**

1. Does the level of visual complexity reflect the visualisation metaphor being used?
2. Is the visualisation able to scale to accommodate varying amounts of data?
3. In the first instance, can the visualisation be generated automatically?
4. Can the visualisation evolve in a meaningful way (i.e. within the constraints of the metaphor) as the underlying data changes?
5. Does the visualisation interface (underlying metaphor as well as the implementation) facilitate easy interaction?
6. Is the representation used (for the visualisation and hence the metaphor detail) fully and completely documented in some way?
7. Is annotated information, over and above the graphics, available to the user of the visualisation in some way?
8. Does the visualisation display extreme data (i.e. possible anomalies) with no problem?
9. Can the visualisation be viewed as both an environment and as still views (even if the still views exist within the environment), under user direction?
10. Can the visualisation be viewed from more than one angle, at user discretion?

#### **Comprehension Features**

11. Is the visualisation capable of indicating syntactic and semantic relations between data objects?
12. Does the visualisation provide different abstraction mechanisms in some way (this may be through the metaphor)?
13. Does the visualisation support goal-directed, as needed, hypothesis driven comprehension?
14. Does the visualisation provide an adequate overview of the data architecture and structure (if one exists) at various levels of abstraction?
15. Does the visualisation support the construction of multiple mental models (domain, situation and program) by enabling information and knowledge gathering in different ways?

16. Does the visualisation provide some form of cross-referencing of mental models, at least through maintaining context, when changing from abstraction levels of querying data?

#### **Features Applicable to Both**

17. Does the user interface present an approachable front to the user?
18. Are there various ways of interacting with the system, both the visualisation and the underlying information necessary for the process of program comprehension?
19. Is directional, focused, navigation possible?
20. Is arbitrary, exploratory, navigation possible?
21. Is navigation between (a) aspects of the various mental models (program comprehension) and (b) various abstraction levels (visualisation) possible?
22. Is the users current focus both obvious and indicated?
23. Do features that are known to act as orientation and navigation cues exist in the visualisation so that the user is able to trace (at least to a degree) the path taken to the current point of focus?
24. Do features that are known to act as orientation and navigation cues exist to show the user paths and directions available from the current location for moving elsewhere?

This framework was devised for evaluation of three-dimensional visualisations, due to the shortcomings of other frameworks to take into account issues relating to three-dimensional visualisations. Therefore, the application to DJVis with its mixed two and three dimensional views may result in some questions not being applicable to the two dimensional aspects of the visualisation, such as the Class View.

This section has highlighted the evaluation frameworks to be used. There are currently only a limited number of frameworks to choose from when evaluating a software visualisation and none that specifically focus on the visualisation of dynamic aspects of a piece of software. It is for this reason that two generic software visualisation frameworks were used, rather than a specific runtime visualisation framework.

## **7.4 Scenarios**

The evaluation of DJVis will also incorporate the use of five scenarios. These aim to demonstrate how DJVis could be used to aid specific software maintenance tasks. Each scenario will be presented in terms of a real world task. The required information to solve this task will then be presented, followed by a description of how DJVis could be used to aid in the completion of the task and find the necessary information. This solution will be one possible approach to the problem, since there may be many possible solutions and techniques as the different program comprehension factors identified in Chapter 2 highlighted. These scenarios aim to show how the tool can be used for general exploration tasks as well as for investigating specific information or code sections. Following the scenarios, there is a more in-depth example of the use of the tool on a specific piece of software and a description of how DJVis can be used to aid a maintainer in gaining an understanding of this software, as well as aiding in a specific task.

## **7.5 Conclusions**

This chapter has outlined the main evaluation techniques that can be applied to a software visualisation tool, discussing both their advantages and shortcomings. This has highlighted the need for more research within the evaluation area of software visualisation, as there is still little work being done in this area making evaluation a difficult task. The chosen evaluation method for the DJVis visualisation is then discussed in section 7.3, along with the use of scenarios in 7.4.

# Chapter 8 DJVis Evaluation

## 8.1 Introduction

This section discusses the evaluation of DJVis. This is split into a threefold evaluation: firstly, DJVis is evaluated informally with each of the views discussed in turn, as well as DJVis as a whole and the suitability of the chosen implementation approach; secondly, the two frameworks identified in the previous chapter are applied to DJVis; finally, the visualisation will be applied to a number of scenarios to show its use on real tasks. The evaluation is against the concepts of DJVis rather than against the implemented system, though experiences of the approach to information extraction etc. will be discussed in the informal evaluation.

A brief synopsis of each view that constitutes DJVis is presented here to reaffirm their content before the views are discussed in this chapter.

- **Runtime View**                      Presents the *threading* aspects of the software using a 3D representation.
- **Query View**                        Acts as a grouping mechanism for information from other views.
- **Class View**                         Provides details on the classes and their relationships using an augmented graph representation.
- **Variable Watch View**            Presents information on the access types and frequency of accesses for a field variable.
- **Method Pixel View**                Presents the method calling relationships of a class' methods using a pixel representation.

## 8.2 Informal Evaluation of DJVis

This section presents an informal discussion and evaluation of the merits of the DJVis visualisation. This is provided alongside the framework evaluation in the next section, in order to allow for a more detailed discussion of aspects that may be missed or simplified by the use of a framework. The informal evaluation discusses each view followed by the collaboration of the views and how they work together as a single visualisation. Many aspects of the approach are discussed, including the information extraction method used and the choice of an online approach. This section aims to provoke thought and questions on the ideas and concepts used with DJVis. It also aims to allow experience gained in the development of DJVis to be recorded, whilst highlighting opportunities for improvement or future work. This could possibly guide future development of runtime visualisation tools by highlighting some of the issues involved.

### 8.2.1 The Views

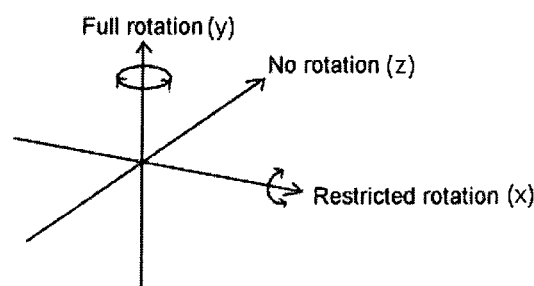
#### The Runtime View

The aim of the Runtime View is to display the thread information on the program. This is achieved using a 3D representation of the threads and thread groups forming a tree structure. This view offers a simple mapping. However, one shortfall is that of the problem of scalability. A large number of thread groups would produce a large tree preventing it from being viewed as a whole and a large number of threads in a single thread group would produce a very long thread group going off into the distance. This would



prevent all the threads from being seen at once and could cause navigation issues. However, while this issue exists, it can be argued that in “real” Java programs only a limited number of thread groups are used. Often none are used other than the default main thread group. The number of threads used is also typically limited to a relatively small number by a “typical” program. The Query View was designed to aid the Runtime View by allowing items of interest, such as particular threads, to be dragged into a query. This, allows them to be easily examined without any of the layout constrictions to which the Runtime View is subjected. The Runtime View presents an overview of the threading structure of the program, whilst also allowing details to be accessed by using level of detail on the objects. This allows details such as the methods and their arguments to be inspected. However, scanning the call stack was found to be time-consuming in terms of navigation, especially when the stack is large. Therefore, a simple textual display of the call stack could be presented alongside the graphical presentation, allowing the user to quickly scan the stack for method names and navigate to particular methods to obtain more information. This could take the form of a popup window, which only appears when the user is investigating a call stack.

The view allows free navigation around the 3D space, however, this is restricted in the prototype tool to remove rotation about the z axis and limit rotation about the x axis. This is to prevent disorientation, as these extra navigation options are not needed to view the information. Until the user becomes accustomed to the 3D navigation, problems of disorientation may occur. The current prototype supports re-centring the view to its initial viewpoint, if the user has become disoriented. There are currently no specific orientation cues within the view, though the tree structure and the connecting lines act as cues when visible, as the tree always grows down and the lines are always at the front of the tree.



**Figure 8-1 Navigation options within the Runtime and Query Views**

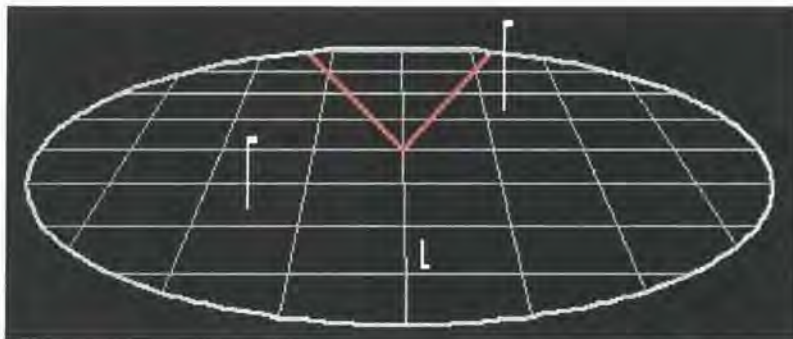
The Runtime View is designed to be used alongside the Query View, thus allowing items of interest to be investigated and preserved across different threads of investigation.

### **The Query View**

The Query View was designed to allow the user to focus on items of interest and preserve user defined groups through the changes in their investigations. It allows items to be dragged into the current query and positioned at user discretion. Query instances can easily be created, deleted, named and duplicated, in order to allow for easy management. Query instances can be quickly selected through the use of a drop down list of instances. The current query objects and navigation positions are preserved for each query instance in order to prevent disorientation. The current discrepancies in the use of 2D and 3D views

prevents the Query View from being as useful as it could be. Items from 2D views could be placed in the Query View on a plane, however, this would introduce many issues with regard to interacting with such structures and how to preserve the familiarity once 3D navigation and perspective is introduced. This lack of support for all views introduces more cognitive load on the user, as they must remember which views the Query View supports, introducing inconsistency in the user interface. The Query View could be more beneficial, if it was supported consistently across all views, however, the current mix of 2 and 3 dimensional representations prevents this in a coherent way.

The Query View uses the same navigation techniques as the Runtime View. The restrictions in navigation are again provided to help reduce disorientation. The viewpoint of each individual query is stored, so that when a query is selected, it is still in the same position as the user left it. However, when there are many active queries and objects in those queries, it may be difficult to remember the position and contents of each. Therefore, some overview of the objects in the query could be beneficial in order to provide context and prevent objects, other than those in the current viewpoint, from being forgotten. This could take a number of forms, for example, presenting a small wider angle view in the top of the picture showing objects to the side or even behind the current view point. However, this different viewing field may prove to be confusing and would still face problems when objects are situated above, or below, the current viewpoint. One possible technique would be to use a 3D radar style approach used in 3D games such as Elite and its sequels [Frontier]. Figure 8-2 shows an example of this style of display. This would allow the position of objects to be seen in relation to the viewpoint, although no information on the actual type of the object would be presented. The scaling used for this aid would be very important for its effectiveness, as objects may typically be closely spaced, when multiple objects have been added to the query without navigation of the view. However, other objects may be distant to the viewpoint and showing these two extremes clearly will require careful design of the scaling mechanism used.



**Figure 8-2 Game based navigation aid for showing the position of objects relative to the user [Frontier].**

Figure 8-2 shows a 3D radar style approach used in the Elite family of games [Frontier]. The user is at the centre of the circle and the red lines indicate their field of view. The white marks represent objects in the world and the line coming from the object indicates the object's height relative to the plane of the user and the end of the line represents where the object is in relation to that plane. Figure 8-2 therefore shows a situation where there are three objects. One object is in front of the user, to the right of their current viewpoint and high above them. The other two objects are behind the user, one directly behind and slightly below the user and one to the left and above the user.

## The Class View

The aim of the Class View is to present information on the class aspects of a piece of software. In order to present the large amount of information on classes it allows the user to easily control the viewing aspects of the visualisation using drop down lists to change mappings within the visualisation. Using this, the user can investigate the classes, their relationships, their fields, as well as their methods and method metrics (number of calls, number of lines and access rights are currently supported by the prototype).

The use of a graph representation offered a compromise. Graphs have instant meaning to programmers due to their common use, yet they suffer from problems of scale as the size and complexity of the graph increases. This trade off allows the Class View to be easily understood and explained due to the common mapping. In order to increase the information presented the class nodes are augmented with method lines representing different method metrics. This allows an impression of the structure of the class to be gained at a glance. Classes with a very large number of methods stand out easily, however, the decreased width of the method lines can make the shading of the lines difficult or impossible to see unless the user zooms in upon them significantly. Sliders provide easy facilities for such zooming, however, it prevents the information being instantly accessible in the initial view. The underlying problems of the graph representation are the main problem with the Class View. It does provide facilities to reduce this effect by allowing the user to group nodes into abstractions, filter nodes from the view and change the node size. These can help the representation, but do not overcome the problems faced as the graph's complexity and size increase. One approach to this problem could be to use a clustering of the graph to allow the graph to be seen at varying levels of abstraction, for example, work by Quigley on the FADE paradigm [Quig02]. This approach offers an automatic clustering of the graph allowing it to be displayed at multiple levels of abstraction. Therefore, higher level abstractions provide an overview of the overall structure, whilst lower-level details provide the details on the individual nodes. This technique allows for variable levels of abstraction across the graph, thus allowing the user to focus on specific details, whilst still being able to observe the overall structure of the rest of the graph. However, thought would have to be given on how such an approach could be incorporated with the augmented method lines of the current nodes. The current user defined clustering would also be unsuitable and therefore the adoption of such an approach would require significant forethought, in order to integrate it consistently. However, such an approach could offer an attractive avenue for future work and allow very large graphs to be handled and displayed more easily.

The interactive layout acts as a means to try and reduce the cognitive load when changing the layout of the graph by animating the changes, for example, due to changing the edge mapping. This does, however, introduce a large number of moving items in a large graph and adds to the time that it takes to change between edge mappings. In order to reduce the need for frequent changes of edge mappings the user can open multiple instances of the Class View, thus allowing them to investigate numerous class relationships simultaneously. The instances can be linked together so that a selection in one graph selects and centres the same node in the other instances of the view. This aims to allow cross-referencing between the different instance of the view, although it can be prevented, if the user wishes to explore different classes

in each view. This is highlighted in the title bar of the window in order to indicate to the user that the window is not synchronised with the other instances.

As with all views the amount that can be seen at once is limited by the screen size of the display. This can be an issue if multiple instances of the Class View are being used at once, as it can be difficult to position the windows to have them all visible and yet at a useable size. The controls for the visualisation take up a certain amount of the view and therefore these take up an increasing amount of the display, as the number of instances of the view increases. Reducing the size of these could benefit the visualisation, as there will be more screen space used for information display rather than supporting graphical controls. However, a compromise must be met between providing clear controls for the view, whilst using a small amount of screen space.

The use of a 3D graph layout was considered for the Class View and some of the graphs were tested in 3D to see how they compared to the 2D layout. Whilst the 3D view did offer some improvements in the layout of certain graphs, for example, by reducing the number of crossing lines, it also introduced a number of other issues. The application of 3D to graphs has been tried many times. For example, Young highlights some of these issues in the discussion of Zeebedee [Youn99]. He highlighted the problems that can occur, such as different view points altering the view of the graph making it difficult to recognise and the crossing of lines also being dependent on the viewpoint. The use of a 3D layout for the Class View would also have been unsuitable as the method lines metrics relies on comparisons of the length of the method lines. This would have been complicated in 3D, as the depth of an item would have introduced changes in size due to perspective. The application of the method line idea also relies on the ability to see all the methods of a class simultaneously. In order for this to occur in a 3D representation, the method lines would have had to use a billboard style effect, where they are rotated to always face the viewpoint. This would allow them all to be seen without the user needing to navigate to a suitable viewing position. The height of the viewpoint would also affect the usability and perceived length of the method lines. The only advantage of using a 3D representation, except for any possible graph layout improvements, would be that it would allow items in the Class View to be coherently dragged into the Query View. This would have allowed nodes and sub-graphs to be preserved and named in queries, alongside other items of interest.

The prototype tool uses a springs style algorithm to layout all the graphs. This provides acceptable results in general. However, it does not support incremental change well, and even when the spatial relationships of the nodes are the same the whole graph may appear to be rotated from its previous layout. This can make it more difficult for the user to recognise parts of the graph. This is an issue with the prototype implementation, as the actual visualisation does not place any restriction on the type of graph layout to use. However, the current layout does tend to show clustering well. It could be beneficial to have a choice of layout algorithms, or specific algorithms for certain mappings. For example, when the graph is displaying the “inherits” relationship it could be beneficial to lay this out using a tree layout as this will re-enforce the hierarchical relationships within the graph.

The Class View allows the definition of custom mapping functions for the length and shading intensity of the method lines. This allows for greater customisation of the tool to user defined tasks, whilst supporting the scalability of the method line representation. However, the trade off for this is the additional cognitive load in remembering which mapping function is being used. This should not be too much of an issue for single users. However, it does introduce complexity when the tool is being used to communicate ideas, because other people investigating the visualisation would have to consider the details of any custom mappings being used. Individual instances of the tool would therefore no longer be consistent and error could be introduced, if a user failed to observe the custom mapping being used by another user.

Overall the Class View allows a large amount of information to be presented on the class structure of a piece of software. It shows the static aspects of the classes and methods, as well as dynamic information such as dynamic references, the number of instances created and method calling details. It allows navigation of the software, for example by allowing the source code of a method to be displayed. It also makes it possible to hypothesise the role of a class, for example, whether the class acts as an interface to other classes or a controlling object, by using the method lines. It does, however, suffer from the underlying problems in the graph representation as the graph size and complexity increases. However, even a complex view indicates complexity within the program and therefore may suggest the need for preventative maintenance of the system.

One possible improvement maybe the incorporation of time dependent relationships with the graph. Currently some information can be recorded over a user-specified piece of the execution, for example the number of method calls. It would also be possible to record this information for the relationships encoded in the graph's edges, such as "creates" or "calls". This would allow the user to investigate specific areas of a program's execution, for example, tracking object creations for some functionality. This change could be easily incorporated in the DJVis.

### **The Variable Watch View**

The Variable Watch View is designed to allow the user to observe how class fields are utilised in the program under study. It provides three levels of abstraction in order to allow the user to observe the overall pattern of accesses, as well as being able to observe the details at the level of individual accesses. The Variable Watch View does not attempt to display the pattern of accesses for all field variables in the software under study. Instead, it is designed to focus on specific fields in which the user is interested. One advantage of the view is the ability to trace the accesses to public fields. These fields can be accessed outside the current class, and therefore it can be very difficult for a user to identify the usage of the field and in particular how it is used for some specific functionality. This is simplified by the use of the Variable Watch View. The view is less useful when the field is private. It still allows the actual usage of the field to be seen, though the class level of abstraction is not of any real use. The internal class usage could be easily found without the view simply by searching the source code as accesses are restricted to that class. However, the Variable Watch View can still be useful for providing a quick summary of the type of accesses and restricts the displayed accesses to those for a particular execution which could filter

the number to be considered. The view also presents the number of accesses, which could be of importance, for example in displaying unexpected patterns in usage.

The main issue with the Variable Watch View is the need to provide a suitable mapping of the number of accesses to the position on the circle. The best mapping will of course depend on the variations within the data, however, the logarithmic scale provides a good compromise allowing variations between small values to be observed, whilst being able to present high numbers of accesses. The view is quite effective at presenting a large number of accesses at once using this mapping, although if a high proportion of the items have a high access count, then they cluster around the central point making it difficult to distinguish between individual items. A zoom facility would help to reduce this effect.

### **The Method Pixel View**

The Method Pixel View allows an impression of a class' method calling to be gained relatively quickly. Highlighting methods that are called by many other methods, or call many other methods, as well as methods that have not been called so far in the execution. However, the view does not show the number of calls for each method and this information could be desired for the user's task, without the added cognitive load of having to refer back with the calling counts presented in the Class View. Therefore, the view could be modified to allow this information to be displayed, for example by using an intensity shading for the pixels. The small size of the pixels could make this choice of scale difficult, as it is difficult to compare small blocks of colours accurately and the colours will have to stand out from the background colour.

One issue is the placing of the method names within the view. Each method is labelled with its name, although, due to the small size of the representation there is only a small space provided for this. This can be problematic when the name is long and has to be concatenated. Also, it is currently difficult to quickly identify the methods that the pixels represent, as it requires placing the mouse over each pixel in order to get the name of the method in a popup window, or zooming in upon a method of interest.

### **View interaction/collaboration**

Each view shows some aspect of a piece of software with the aim of providing a more complete picture of the software when used together. Therefore, the inter-view interaction is important in the usability of the visualisation. The use of multiple windows allows for flexibility in presentation, however, it does complicate the user interface and makes it difficult to use multiple views together, especially on lower resolution displays. The views provide cross-referencing by allowing a selection in one view to select the same item in another view (or showing related information regarding that item in the context of the view). For example, the Class View can be used to navigate to the source code of a class and selecting a method in the Class View will display the associated method's source code. This cross-referencing aids the user in switching between views, though it is not always possible to synchronise all views on the same item, due to the different information displayed in each of the views. For example, selecting a thread in the Runtime View has no effect on the Class View, as it does not present threading information. The views also

support popup information, which aims to provide pertinent information and reduce the need to switch views. For instance, in the Class View the source code of a method is presented, when the user queries it with the mouse.

## 8.2.2 The Visualisation and implementation approach

### The Visualisation in General

The DJVis tool offers a different slant on the display of runtime information. The views show different aspects of a running program, however, they do not present the full picture and there are many further aspects of the software that could also be presented. Currently, there are a number of areas of DJVis that are lacking. In order to provide a broader picture of the software at runtime, the visualisation also needs to present the objects and their relationships. The focus of this thesis was to show class level and threading details. Therefore, the visualisation needs support for displaying objects and their relationships and this is discussed in the further work section of this thesis.

DJVis offers a mix of two and three-dimensional displays. This has the disadvantage of views not sharing the same interaction mechanisms. The use of the Query view is diminished by the lack of interoperability with the two dimensional views.

The visualisation currently only focuses on the method calling level, with no more detail automatically extracted. This is not to say that the user cannot step through the code line by line, or examine the value of fields and local variables. However, the views currently do not have access to sub-method calling details automatically, so for example, references to classes that are through local variables are not currently recorded. It would be fairly easy to record and show the types of the local variables in each method in order to show any new class references. However, the performance overhead of watching local variables for dynamic references and updating them in real time would be too great and difficult using the current information extraction technique. This issue is a combination of issues in extracting the required information and issues in how to present that within the current visualisations.

### Attaining information for visualisation using the Java Platform Debugger Architecture

The use of a debugging mechanism, in this case the Java Platform Debugger Architecture, offers both advantages and disadvantages for a software visualisation tool. It allows the visualisation of Java programs without the need to modify the source code of the program and in a way that provides support for different VMs and platforms (provided they support the JPDA). The only requirement on the code is that it was compiled to include debugging information. The JPDA provides a wealth of information on the program under study, from the details of the static classes descriptions, to the actual usage of the classes and their dynamic referencing. This can be accessed through a fixed interface that supports different VM implementations running on a variety of platforms, on possibly remote machines. Thus, despite the fact that the DJVis prototype tool only runs under Microsoft Windows, it is still capable of visualising a Java program running on a different platform, for example Linux. The JPDA provides access to a substantial amount of information on the executing software. In general it is able to provide all

the information necessary for the generation of the DJVis views. However, there are some pieces of information that are not available using this information extraction method. These areas are discussed below in Table 8-1.

Information Required	Required For
Scope information on methods	The Runtime View allows methods on the call stack to be investigated and at a fine grained level it presents details of the method's scope information as well as indicating loop and conditional blocks. This information is not directly available from the JVM however, the source file name of the class can be extracted along with the line numbers of the method. Therefore, in cases when the source code is present this information can be extracted by parsing the method source code and then storing the results for future use. The method source is already parsed to an extent to allow the syntactic highlighting to be shown in the Source View so this is not a significant issue. The visualisation of method metrics, such as complexity measures, would also rely on the parsing of the source code rather than on information available directly through the JPDA.
Field Variable Access counts	This information is available from the JPDA, however, watching all class variables would be unfeasible, due to the performance overhead.
Local Variable Accesses	The JPDA provides details of the local variables in methods, however, there is no support for watching local variables. This can be done by manual inspecting the values of local variables as the method is stepped through, but this is obviously unfeasible due the performance overhead. This means that the display of class reference relationships in the Class View does not include local variable references.
Current number of instances of a class	The Class View shades the class node to represent the number of instances of the class that have been created. However, it would also be useful to be able to show the number of instances that exist at the current execution point. The JPDA does not provide any information on the collection of objects or on objects being no longer referenced. It would be possible to build an object graph allowing the system to show the current object graph, however, this would only be suitable when the program is paused and could be very time-consuming.

**Table 8-1 Information that is unavailable using the JPDA**

### **The Performance of JPDA and DJVis for software visualisation**

The main disadvantage in the use of the JPDA is the large performance overhead for the VM in recording and communicating the events of interest to the visualisation tool. Watching the method entry and exits puts a very large overhead on the VM and significantly slows the execution of the Java program. For



large programs this time overhead may prevent the prototype tool from being feasible, especially if the event of interest only occurs after a lengthy execution and requires multiple runs of the tool to investigate. The JPDA supports filtering of these events and this is done for the Java API classes by default. It could also be done for users defined classes and packages, which can significantly improve the performance if large amounts of uninteresting execution can be removed from the trace. However, this then introduces the issue of the “truth” of the visualisation as some of the execution is filtered. This is already an issue that needs to be considered as the current default settings filter out all the Java API calls so usage of these classes and methods is hidden from the user.

The main overhead of using the JPDA is the tracing of method entry and exits, due to the extensive number of method calls even in a small program. This calling information is used to record method call counts, method calling relationships and object creations. Removing the tracing, allows the visualisation to generate views of the class referencing and method lengths and access rights, however the usefulness of the information is greatly reduced. As a compromise between depth of information and performance overhead, it may be useful to allow the user to enable and disable tracing whilst the program is executing. Therefore, the user could quickly execute sections of the program in which they are not interested without the method tracing, such as the program’s initialisation. The tracing could then be switched on for a section of the execution in which the user is interested. This introduces issues of inconsistency as the presented information is not of the entire execution, but only of the sections when method tracing was enabled. Provided the user took this into account, then the visualisation could still be useful without being too slow. However, the views must also indicate that the information is incomplete using some visual means, even if it is something as simple as indicating it in the title bar text. Another option would be to allow the saving of calling states at some specified point, for example, a breakpoint set after program initialisation. Subsequent executions could then run to this point without the tracing, greatly reducing the performance overhead. Execution from that point could then use the loaded calling history, but now with method tracing enabled. This would allow the full information to be used in cases when the same software is being executed many times. However, this would be error prone and dependent on the program having identical execution to the specified point. There would be no way to check that data was consistent other than using the current state of the call stacks. Therefore, this would be unsuitable for multithreaded programs or on sections of the code that were affected by user inputs if they were not identical. Such a feature could introduce a significant error into the visualisation and it would be the responsibility of the user to know how and when to apply it.

A number of simple experiments were performed to gain an understanding of the performance overhead introduced by use of the JPDA and the visualisation itself. A number of sample programs were timed for a section of their execution. The time taken was recorded when running the program normally, when the program was monitored using the JPDA and when using DJVis to monitor the program. These experiments were performed on a 500 MHz Celeron PC with 320 MB of memory running Windows NT and using the Sun JDK 1.3.1. The timings for the use of the JPDA give the basic overhead of using this method for information extraction, when tracing method entry and exits. The client program for this simply consumed the events as it received them. The average overhead to extract the runtime information using the JPDA was an increase in running time by a factor of 50. This overhead factor varied between 10

and 90 depending on the task being timed. The large variations in the overhead factor are due to differences in the programs being traced and the code called. The tracing produces an event and therefore an overhead every time a method is called or returns. Therefore, programs with a large number of small methods will incur a higher overhead compared to programs with large methods. This tracing overhead is also affected by the use of the Java API, as this code generates events that are filtered by the JVM. The overhead to then visualise the extracted information was only a factor of 1.5 increase. Therefore, the main overhead is the extraction, rather than the actual visualisation. This visualisation overhead was for the prototype tool and could be reduced further, if additional time was spent optimising the storage and visualisation code.

The use of the JPDA offers one of the least intrusive methods of tracing a program. It requires no changes to the source code of the program and the addition of debugging information to the class files is handled automatically by the Java compiler. JPDA is now widely used by IDE developers [JBuilder][NetBeans] [IntelliJ] and therefore should remain supported and provide a consistent interface to future releases of the Java platform. This isolates the tool from any possible changes that occur to the class file format that could influence class file modification approaches. However, it is also possibly the most costly in terms of the overhead incurred. The occurrence of an event, for example a method entry, results in the JVM having to switch from the executing the Java byte code to a debugging thread which then checks to see if the event is included in the current filters. If the event is not filtered, then it extracts the required information for the event and sends this as a packet to the tool using the JDWP. This, is obviously a very expensive operation when performed frequently, as is the case with method entry and exits events. Extraction techniques that add trace code to the actual Java byte code (through manual or automatic techniques) have a reduced overhead as the code is executed when the method executes and therefore does not require the constant changing of threads. It also prevents the need to filter classes, as classes of no interest will simply not be augmented with trace code. This offers a reduced overhead, however, it requires greater investment in order to generate the augmentation method. Furthermore, care must be taken that the information displayed to the user has not been modified due to the augmentation, for example, the lines of code of a method. Such methods would also suffer from being unable to step through the code line by line, whilst inspecting the contents of variables, without substantial augmentation which would offset any performance advantage.

The use of a debugging technique reveals the inner details of the program under study that the programmer may not wish to consider. For example, the implementation details of the Java API and the threading of the JVM. These events are filtered out by default, but this does lead to a loss of information. For example, if filtered API code calls code that is not filtered, the caller is not available in the method calling history. Instead, only the fact that there was a call from some filtered method is recorded. The filtering of method entry and exits occurs at the JVM and methods are therefore filtered from the call stacks presented in the visualisation. This is an implementation detail as the call stacks can be queried to find all methods on the stack, however this then introduces additional information that the user may want to remove and may present information on details that are not available in the other views introducing inconsistency. The classes are presented as the JVM uses them, which can introduce additional information as the compiler can add methods and fields that did not exist in the source file declaring the

class. These synthetic fields and methods can be filtered provided the JVM is capable of identifying them as synthetic, which is not always available depending on the JVM used. An understanding of these concepts is needed when using the tool so as not to interpret the visualisation incorrectly.

### **Online vs. offline information extraction**

The online approach provides yet another area of compromise. Both online and offline approaches were considered in the design of the visualisation and the JPDA could have been used for an offline approach, by extracting events into a file for later processing. However, an online approach was chosen with the visualisation and target program running alongside each other. This allows the visualisation to control the execution and allows the user to relate actual program events to visualisation events. There may be a lag between the details presented in the visualisation and the program's actual execution, therefore the number of outstanding events still to be processed is presented in DJVis, from which the user can tell if they are synchronised. This synchronisation makes it easier to relate the details presented in the visualisation to actual states in the program. For instance, the recording of method calls can be done between two points in the execution with the user directly able to see how the execution has affected the state of the program. Using an offline approach would have allowed summary statistics to be presented and even the possibility of back stepping the execution. However, the increased separation between the visualisation and the target program would make it difficult to know what state the program would be in at that point in the execution. The online approach also allows details to be extracted, if they are needed, for example, a user can extract the values of field variables or local variables of methods on the call stack. It also allows the use of standard debugger commands such as breakpoints and the ability to watch fields for accesses or modifications. The offline approach has the advantage of being able to have the complete execution trace in advance of producing the visualisation. Therefore, the trace can be processed to produce summary statistics and allow future states to be known. The layout of the visualisation can also be optimised to take these future states into account. The separation between the target program and the visualisation tool also means that they are not competing for the same processor and screen space, when running on the same machine. This can be combined with the ease of rerunning the visualisation as the same trace can be used and there is no need to run the program again, which may be computationally expensive. However, this trace file also introduces its own overheads, as the storage required for the execution trace may be massive, especially for a large program.

## **8.3 Application of the Frameworks to DJVis**

This section provides the details of the application of the two evaluation frameworks to DJVis. The framework by Storey et al. [Stor97a] is applied initially, followed by the framework by Knight [Knig00]. Questions in the framework by Knight, that overlap with the first framework will refer the reader back to the appropriate answers in the first framework. Each framework has a simple scale for the response to the questions, followed by a justification of the response. The responses are:

- Yes : The feature is fully supported.
- Yes? : The feature is mainly supported.
- No? : The feature is mainly not supported.

- No : The feature is not supported at all.
- N/A : The question does not apply to DJVis.

The response could be classified on a sliding scale with more increments, however, this introduces the possibility of error and personal judgements biasing the results. The responses provide greater details on how well the response fits into the categories by providing more details on the level or lack of support for a particular feature. It must also be remembered that it is the concepts of DJVis that are being evaluated and not the prototype tool or the implementation techniques used to implement it. Table 8-2 shows the results of applying the framework by Storey et al. [Stor97a] to DJVis.

Element		DJVis support	
E1	Indicate syntactic and semantic relations between software objects	Yes?	DJVis allows access to the source code through the Source View. The other views also allow access to the source code for the item under study. Syntactic information is provided on items and semantic information on classes can be attained from the Class View whilst the Method Pixel View shows calling relations between individual methods. The Variable Watch View allows the user to observe data usage, by presenting usage relationships between class fields and classes, methods and source lines.
E2	Reduce the effects of delocalized plans	No	The visualisation does not offer any support for reducing the effect of delocalized plans, such as through visualising program slices. Delocalized plans were not considered in the design of the visualisation.
E3	Provide Abstraction mechanisms	Yes	DJVis supports abstractions in the Class View and each view offers a different level of abstraction of the software. Currently other views do not offer the ability to define abstractions, however, they could be modified within the constructs of their representation to allow user abstractions. For example, user grouping of methods within the Method Pixel View.
E4	Support goal-directed hypothesis-driven comprehension	Yes	DJVis allows the user to explore the data freely, with options to display different relationships dependent upon their task. There is also support for searching for specific information, for example for a particular class in the Class View.

E5	Provide an adequate overview of the system at various levels of abstraction.	Yes?	Each of the DJVis views offers a picture of the software at a different level of abstraction. Views such as the Class View provide overview windows alongside the more detailed view and allow the ability to abstract further. However, the views do not show all abstraction levels of an executing system. Therefore, there is scope for the addition of different views, for example, to show object level interactions or higher level design patterns.
E6	Support the construction of multiple mental models (domain, situation, program)	No	DJVis enables information and knowledge gathering in a number of different ways, however, there is no explicit support for domain, situation and program mental models. In particular, domain knowledge is not explicitly supported, for example through links to external documentation or domain specific analysis of the information. User annotation and grouping is provided and information on items can typically be attained in a number of ways.
E7	Cross-reference mental models	Yes?	The different views are synchronised so a selection is reflected in the other views. For example, selecting a class in the Class View updates the Method Pixel View to focus on that class. However, due to the different nature of the information types in the views this is not always possible. For example, the Runtime View does not show classes directly so a selection of a class in another view will have no effect. However, all views support popup menus to show the current item in all the other views. The synchronisation of selection in the views can be switched off, if desired by the user, to allow them to compare different items. In this case the selected item is highlighted in all the other views in which it is currently displayed, but it does not become the focus of the view.
E8	Provide directional navigation	Yes	The visualisation offers techniques for finding specific information, such as searching by name or regular expression. The helper views, such as the tree of loaded classes, can drive the other views and allow information, for example on a particular class of interest, to be found easily.
E9	Support arbitrary navigation	Yes	The user can freely use the views to explore the information. The views can be configured e.g. the Class View, to explore different relationships dependent on user needs. History lists allow arbitrary navigation between viewpoints. The visualisation does not constrain the order or type of information that the user can investigate.

E10	Provide navigation between mental models	No?	DJVis does not explicitly support multiple mental models. However, it does provide support for navigating between views. This can be done implicitly by selecting in one view and viewing in another, or explicitly, by the use of the pop-up menus on items, which provide links to other views in which the information can be presented.
E11	Indicate the maintainer's current focus	No?	The Class View provides an overview window to indicate the present focus, whilst the Method Pixel View labels the window with the current class that is being presented. However, the Runtime and Query Views do not present the current focus in terms of the user's position within the 3D space of the views. The current items in the view can be labelled, but there is no support for showing an overview of the entire view or the user position within it.
E12	Display the path that led to the current focus	No	The views preserve a history of actions that lead up to the current view, however the path that lead up to the current focus is not displayed. This could be displayed, for example in the Class View, by highlighting the nodes which have recently been selected. However, this would be visually complex and therefore the history list is the only record of the path.
E13	Indicate options for further exploration	Yes	All graphical elements in the visualisation represent information, therefore, the user can see where they can explore. Related items are shown in the Class View by edges, which highlight further areas for investigation. Popup context menus are available for all items that allow them to be displayed in other views highlighting further options for investigation.
E14	Reduce additional effort for user-interface adjustment	Yes?	The prototype tool uses standard GUI controls and help techniques such as tool tips. However, the multiple window approach can make window placement and switching problematic, especially on small displays or when multiple instances of a view are open.
E15	Provide effective presentation styles	No?	Without more detailed evaluation it would be incorrect to say that the representations chosen are effective presentation styles. DJVis is composed of a number of views, each with its own presentation style. The Class View uses an augmented graph representation, which has the advantage of being very familiar to software developers and maintainers.

**Table 8-2 Application of Storey et al. [Stor97a] framework to DJVis**

The application of the framework to DJVis has shown mainly positive results, with DJVis providing a number of exploration techniques to users. One aspect of the evaluation that DJVis offered no support for was E2 ("Reduce the effects of delocalized plans"), as the visualisation was not designed with delocalised

plans in mind. Support could be added to show delocalised plans by means of displaying trace information, such as a dynamic slice. However, how feasibly this information could be extracted using the current technique and incorporated into the existing visualisation would have to be considered. Two other areas that DJVis does not offer much support for are elements E11 and E12. There is some support for E11 ("showing the maintainers current focus") however, this is difficult in three dimensional views as the user may actually be looking at something other than at the centre of the view. This is not a major problem for the use of DJVis, as the views provide naming of items, if the user selects them. Element E12 ("display the path that led to the current focus") is not supported by the visualisation apart from through the storage of history events, however these are not displayed. This is not considered to be a major shortfall. Displaying the path that lead to the current can be difficult especially in the 3D views and also due to the dynamic nature of the views as the path to the current focus may only make sense in previous states and not the current. DJVis was then evaluated using Knight's framework and the results are shown in Table 8-3.

### Visualisation Features

Element		DJVis support	
1	Does the level of visual complexity reflect the visualisation metaphor being used?	Yes	The visual complexity reflects the abstract metaphor used. The visualisation aims to use the minimum amount of graphics to represent each item, so visual complexity relates directly to information content.
2	Is the visualisation able to scale to accommodate varying amounts of data?	Yes?	The visualisation is designed to handle large amounts of information. The Runtime, Method Pixel and Query Views focus on specific information and are therefore unaffected by the size of the program. For example, a large call stack size can easily be presented in the Runtime View. The Class View does suffer from problems of scale, if a large number of nodes or edges exists. To reduce this problem, overview and abstraction methods are provided. The Runtime View could also suffer, if a very large number of threads exist in the same thread group though this would just increase navigation and threads of interest could be placed in the Query View, if desired, to prevent the need to keep repositioning the viewpoint.
3	In the first instance can the visualisation be generated automatically?	Yes	The visualisation can be generated completely automatically. It requires no user intervention or modifications to the source code (NB. The code must be compiled with debug information). The only requirement on the user is to specify the Java or class file that contains the program entry point and to specify any arguments needed.

4	Can the visualisation evolve in a meaningful way (i.e. within the constraints of the metaphor) as the underlying data changes?	Yes	DJVis uses abstract metaphors for the different views. These do not impose strong expectations on the evolution of the visualisation as the data changes. Data changes are reflected within the constraints of these metaphors, for example, a new class becomes a new node in the Class View graph. The visualisation of runtime information implies that some information will be in constant state of change as the program executes. This could make it very difficult to apply real world metaphors due to the rapidity of the changes.
5	Does the visualisation interface (underlying metaphor as well as the implementation) facilitate easy interaction?	Yes?	While no user experiments have been carried out the visualisation was developed to allow ease of use. The views focus on showing specific information and provide multiple interaction /customisation options. The actual implemented prototype uses standard windows controls and the MAVERIK default interaction methods of mouse and keyboard for the Runtime and Query Views.
6	Is the representation used (for the visualisation and hence the metaphor detail) fully and completely documented in some way?	Yes	The representations for each view are completely documented in Chapter 5. The interaction between the views is also described in detail.
7	Is annotated information, over and above the graphics, available to the user of the visualisation in some way?	Yes	Additional information is available about items through a number of techniques (mouse over text, popup menus and user annotation).
8	Does the visualisation display extreme data (i.e. possible anomalies) with no problem?	Yes	The visualisation displays extreme values with no problem. For instance, extremely large methods are displayed without a problem in the Class View, the method line for such methods would be extremely long and clipped, if it goes off the screen. This extreme value may occlude other items, but it will be very obvious within the visualisation, therefore, drawing user attention to investigate it. The custom mapping modes for the method lines can also be used to show extreme values effectively.



9	Can the visualisation be viewed as both an environment and as still views (even if the still views exist within the environment), under user direction?	N/A	This does not apply especially to the two dimensional aspects of DJVis.
10	Can the visualisation be viewed from more than one angle, at user discretion?	N/A	Yes for the Runtime and Query views, however, the other views are 2D and so this does not really apply. The Runtime and Query views allow free navigation with the only restrictions being the removal of rotation around the z axis and restriction of rotation around the x axis. These do not prevent the user from seeing any information and are used simply to ease navigation and reduce disorientation from unnecessary rotations.

### Comprehension Features

11	Is the visualisation capable of indicating syntactic and semantic relations between data objects?	Yes?	See E1 of Storey et al. framework in Table 8-2.
12	Does the visualisation provide different abstraction mechanisms in some way (this may be through the metaphor)?	Yes	The different views provide details on the executing program at different levels of abstraction. The user can also create their own abstractions.
13	Does the visualisation support goal-directed, as needed, hypothesis driven comprehension?	Yes	See E4 in Table 8-2.
14	Does the visualisation provide an adequate overview of the data architecture and structure (if one exists) at various levels of abstraction?	No	Currently the data architecture is not visualised, just the structure in terms of classes and packages and threads. This could be extracted from the program under study and the visualisation of this is further work.

15	Does the visualisation support the construction of multiple mental models (domain, situation and program) by enabling information and knowledge gathering in different ways?	No	See E6 in Table 8-2.
16	Does the visualisation provide some form of cross-reference of mental models, at least through maintaining context when changing from abstraction levels of querying data?	Yes?	See E7 in Table 8-2.

#### Features Applicable to Both

17	Does the user interface present an approachable front to the user?	Yes	The user interface uses standard GUI widgets and all interactions can be done using the mouse, from navigating within views to selecting items for more details.
18	Are there various ways of interacting with the system, both the visualisation and the underlying information necessary for the process of program comprehension?	Yes	The visualisation supports various views of the data, which can be accessed in a number of ways, as well as supporting textual views of the software.
19	Is directional, focused, navigation possible?	Yes	See E8 in Table 8-2.
20	Is arbitrary, exploratory, navigation possible?	Yes	See E9 in Table 8-2.
21	Is navigation between (a) aspects of the various mental models (program comprehension) and (b) various abstraction levels (visualisation) possible?	Yes?	This is possible due to the free use of the different views and their options, along with options for locating items of interest.
22	Is the users current focus both obvious and indicated?	No?	See E10 in Table 8-2.

23	Do features that are known to act as orientation and navigation cues exist in the visualisation, so that the user is able to trace (at least to a degree) the path taken to the current point of focus?	No	The visualisation does not support features that are known to act as orientation and navigation cues. This does not really apply to the 2D views, where techniques such as overview window are used to show user orientation. The 3D views (Runtime and Query) provide no known orientation cues, however, in the Runtime View the overall structure forms a tree, which the user can use to orientate themselves. The Query and the Runtime View both support centring the view to its initial position.
24	Do features that are known to act as orientation and navigation cues exist to show the user paths and directions available from the current location for moving elsewhere?	No	The visualisation does not support this. All graphical elements in the visualisation provide details on some aspect of the software, which can be inspected by selecting them with the mouse.

**Table 8-3 Application of Knight's [Knig00] framework to DJVis**

The application of Knight's framework to DJVis shows the same weakness as highlighted by the framework by Storey et al. DJVis does not use features known to act as orientation cues, nor does it show the current focus for the three-dimensional views. The visualisation performed well on both frameworks, although there was overlap between the frameworks so this was to be expected for some sections. The application of the frameworks demonstrated one of the aims of the visualisation, which was to allow the user to control the exploration and investigate different aspects of the software, as they require. Therefore, DJVis has good support for the different comprehension and navigation features in the frameworks. The use of feature-based frameworks provides an evaluation process where it is the process of answering the questions and the thinking it promotes, that is as important as the actual answers. The frameworks provide support for the benefits of DJVis in program comprehension activities.

The frameworks used do not evaluate the dynamic aspects of DJVis, but just its general program comprehension and visualisation features. Therefore, more work is needed on the evaluation techniques for runtime visualisation tools and this needs to have a specific focus on the issues introduced by the dynamic aspects of the visualisations. These issues need to be identified in detail, however a number of sample issues are presented here for discussion. This is not the definition of a framework, or a definitive list of issues. However, it aims to act as a starting point for considering the issues involved when evaluating a dynamic software visualisation. Each issue is followed by a fuller explanation of the issue and the support DJVis provides.

**Does the visualisation offer up-to-date views of the software as well as summary and history views?**

- When dealing with dynamic data it is not only important to be able to observe the current state of the software but also be provided with details of how the software arrived at that

state. This may be through a summary of previous states or through showing views of previous states.

- DJVis supports up to date views of the program e.g. Runtime View, whilst showing summary details in the Class View and Method Pixel Views by showing the history of method calls. The Variable Watch View also presents summary details on field accesses. However, there are no explicit history views that show the in depth history of how the program reached the current state.

#### **Does the visualisation allow the user to see or restore previous states of the execution?**

- This follows from the last issue. The user may wish to examine previous states in full detail or even restore the execution to a previous state so they can investigate a hypothesis. This may be simple for offline systems driven from a trace file, however for online approaches it is a major challenge and very few visualisations provide this. ZStep 95 [Lieb97] is a notable example, but this focuses more on visualising small-scale Smalltalk programs. The issues involved in restoring previous states are not visualisation related but instead focus on the execution of the software. The main problem is the computation costs and the resulting performance overhead of allowing reverse execution. One trade-off may be the idea of providing roll back to set points in the program's execution, which could be user definable.
- DJVis does not offer any support for restoring the execution to a previous state. The Class View allows the state of the view to be saved and displayed at a later date however, this is displaying previous states of the view which represent some execution point rather than actually shown the previous execution across DJVis.

#### **Does the visualisation system allow the program's execution to be compared to previous states?**

- If the visualisation is capable of displaying previous states can these be compared directly to the current state, or can only one state be shown at once. Is there any support for automatic identification and highlighting of changes between the states? The display of previous states is useful, but more beneficial if it is provided in a way that can be directly compared to the current execution state. Thus, allowing the user to compare changes that have occurred, due to some event.
- The only previous states displayable within DJVis are previous saved instance of the Class View. This can be displayed alongside the current Class View representation, however there is no automatic support for highlighting differences between the views. Some information, such as method calling counts, can be reset and therefore used to show changes from a specific point.

#### **Does the visualisation allow the program to be stepped through at a number of speeds and abstraction levels?**

- The visualisation should be able to control the speed of the execution of the program under study. This may be literally through direct control of the program's execution, or by controlling the speed through a trace of the program's execution. The visualisation should be able to represent the program at various levels of abstraction, in order to allow the user to

deal with the vast volume of information. The speed of the execution is also linked to the abstraction levels, as higher abstraction levels will allow quicker execution, while low level views will only be comprehensible at slower execution speeds.

- DJVis provides direct control over the program's execution offering the standard debugging controls of run, stop, pause and step. While DJVis supports different stepping sizes it may be useful to have a speed control for the run option. Each view provides a different abstraction level on the software and most views provide their own abstraction levels also.

**Does the visualisation provide traceability between the outward state of the program and the details of its execution being presented in the visualisation?**

- Dynamic analysis of program results in a huge amount of information and the user needs to be able to easily relate the abstract information they are being presented with against the external behaviour of the program. For example, relating the current section of the call stack to some specific functionality.
- The use of an online approach for DJVis means that the program events are visualised in real-time<sup>4</sup>. Therefore, the user can easily observe the current external behaviour of the program compared to the internal state as displayed by the visualisation.

**Does the visualisation aim to preserve consistency across visualisation of the data?**

- The visualisation should aim to try and preserve consistency between visualisations. The issues involved in the evolution of static analysis based software visualisations have been mentioned by a number of authors, for example work by Knight [Knig00] and Young [You99]. However, the dynamic nature of the runtime information means that it is constantly changing, through changes in the input data, the program's state and traditional evolution of the program's source code. In order to reduce user disorientation the visualisation should aim to provide consistency in the presentation of the common aspects of the data. This is a major challenge in a dynamic data environment due the scale and frequency of the possible changes, especially if the data is viewed online with limited opportunity to process it based on previous presentations.
- DJVis stores any user annotations and specific colouring options across executions. Therefore, details such as package colouring in the Class View remain consistent even when visualising a different program. The Class View relies on a graph layout and therefore the evolution consistency of this depends entirely on the layout used in the implementation. The prototype tool uses a springs style layout, which does not provide consistent results given the same input. Some of the views such as the Variable Watch View are based on recorded events that only hold for some particular point and therefore they will be consistent provided the data is the same. The layout of the method lines in the Class View is based on the order of the methods in the class file so the methods will be added dependent on where they are

---

<sup>4</sup> The visualisation displays events on the program as soon as it able to process them. However, despite the direct connection there may be a lag between the actual event and the visual representation of it when there are a large number of events. The number of outstanding events is indicated by the visualisation.

added into the class' source code. However, unless there are a large number of new methods this will not result in substantial changes. The method line visualisation does not indicate details on previous versions of the class, therefore classes that have undergone a large amount of change will only be observable to the user if they compare with previous visualisations or knowledge.

**Does the visualisation allow the user to observe the program as it runs or just at points when the program's execution is paused?**

- This reflects whether the visualisation can be used to observe the program while it is executing, or whether the visualisation is only suitable for showing the current state while the program is paused. This is not to say that all representations or information are suitable for this dynamic update. For instance, views that focus on displaying low level details such as local field variables or method call stacks will be unlikely to be of use while the program is executing, due to the speed of change of such information. This relates to whether the visualisation can control the speed of execution and whether the views have been designed to provide details of the current point in the execution or an overview of the current state. Some representations are more suitable for displaying rapid changes whilst others are more useful for fixed points when the user has time to study and investigate the visualisation.
- DJVis provides a variety of views and each of these has its own capabilities in terms of displaying the program's execution. The details of the call stacks are unsuitable for displaying rapid execution due to their low-level nature. While the Class View's method lines allow the method usage to be seen with the speed of growth of the lines relating to the frequency of calls. However, a suitable mapping function must be chosen to allow the large number of calls from cluttering the display. Even this representation is best suited for specific sections of the execution as this allows the user to relate the calling pattern with some specific functionality.

These issues highlight some of the considerations when designing a runtime visualisation and to some extent the desired features determine the applicability of the different information extraction techniques. These issues focus on runtime visualisation tools rather than the generic issues of dynamic data. However, some of the issues will be applicable to both, such as the ability to compare the present and previous states. Other design decisions exist for dynamic data that have no right or wrong answer, but depend purely on the aims and task support of the visualisation. One such decision is how new information should be incorporated into the visualisation? This can be done through the addition of new information to the view automatically, regardless of the users focus or knowledge of that information, or the user can be alerted to the information by making the addition of the information indicated. In some situations it may be the change in the information that is more important than the current state or values, therefore, the design of representations needs to consider such issues.

This section has shown the application of the two frameworks to DJVis. This evaluation has highlighted a number of important aspects of DJVis and illustrated areas where improvements could be made. This was followed by a discussion of issues that runtime visualisations face, but which are not addressed in the

current evaluation frameworks due to their generic software visualisation focus. This is not a definitive list of issues, but is designed to promote thinking before the design of a runtime visualisation.

## **8.4 Scenarios**

This section aims to demonstrate how the visualisation could be used on real world problems. Five scenarios are presented along with a walkthrough of how DJVis could be used to aid a user in such a situation. Each scenario is presented in terms of: a task that needs completing; the information requirements that a software engineer would need to complete the task; and then how DJVis could be used to aid the acquisition of that information. This is not to say that the information could not be obtained in another way, whether using the tool or not, as each user differs in their program comprehension approach, depending on their experience and knowledge of the task, language, software and domain.

### **8.4.1 Scenario 1: Corrective Maintenance**

#### **8.4.1.1 Task**

A maintainer is presented with a piece of software that a user has reported a bug in. The maintainer has no real knowledge of the software that they are to fix.

#### **8.4.1.2 Information Requirements**

The maintainer knows which functionality has the error, but they will need to locate the cause of the error and plan a fix to cure the problem without introducing any new side effects. Initially, they will need to gain an overview of the software and identify the classes responsible for the functionality where the error is occurring. Once the classes responsible for the functionality are identified, closer inspection of them is needed to identify the possible cause of the error. This could involve numerous hypotheses on what is causing the error each of which will need investigating. This investigation could require inspection of classes and methods, as well as class fields and local variable values.

#### **8.4.1.3 Application of DJVis**

The first step is to locate the code that is causing the problem. This can be a difficult task even when the maintainer has some knowledge of the system. DJVis offers support to help localise the search for the code. The maintainer can run the software in DJVis. The maintainer can then pause the execution of the program using DJVis just before the functionality in which the error occurs. The Class View can then be used to present an overview of the class structure. The method line mapping can then be changed to represent the temporary number of calls and this temporary count of method calls can then be reset. The maintainer may have some expectations about the location of the code that performs the functionality purely from the names and structure of the class in Class View, however, in order to see what methods and classes are actually involved, the program is restarted. Once the program has performed the functionality with the error in it, the program can be paused again and the results of the method calling

inspected in the Class View. This view could also be watched as the functionality occurs, to see if there are any temporal patterns.

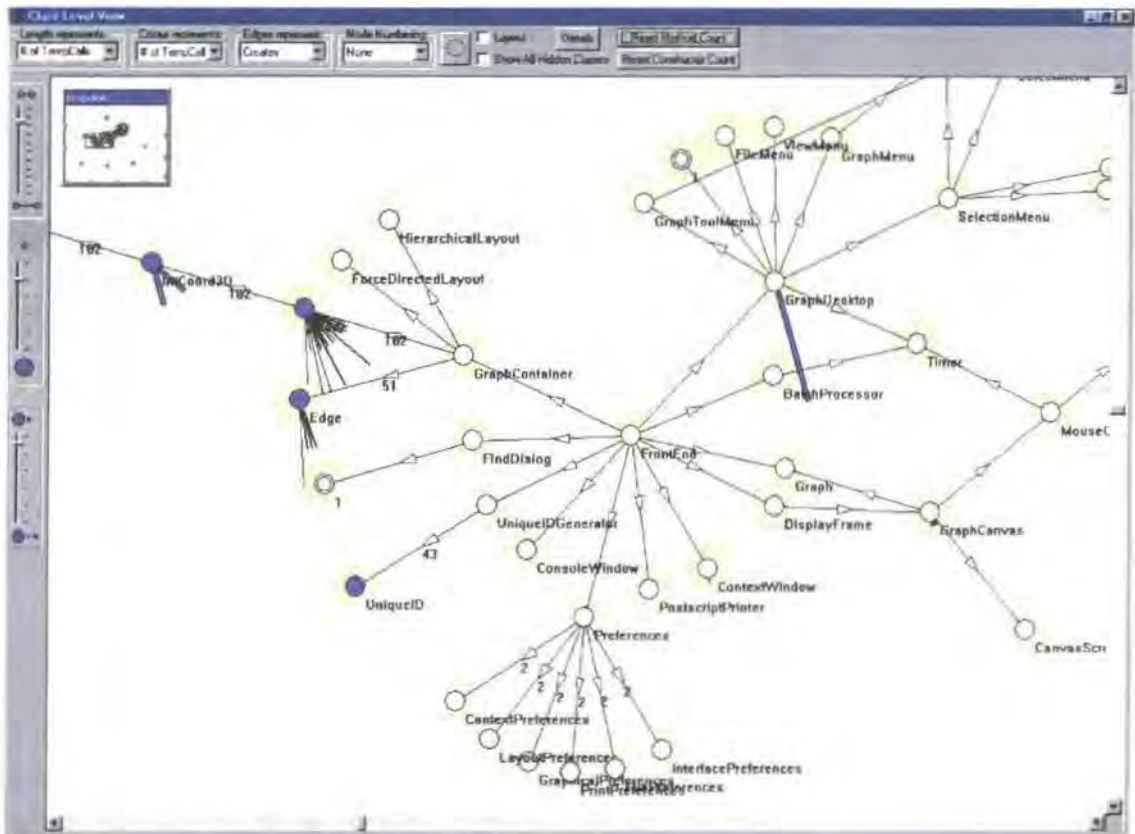


Figure 8-3 Method Calling shown in the Class View

The smaller the amount of the code called for the functionality, the easier it will be to localise the search for the problem. Identifying the point to reset the temporary method calling count may be easy, if the functionality is clearly initiated from some point, for example, if it is menu driven. However, if it is mixed in with other functionalities a process of iterative refinement may be needed. Figure 8-3 shows an example of displaying the calling count as the method line length. It is easy to see which classes and methods are involved and this can be explored further, using mouse over to get details of the method's name and signature. For example, in Figure 8-3 one method of the GraphDesktop class and multiple methods of the Edge, Node and IntCoord3D classes are called a number of times. This view can help prevent the maintainer missing interactions that may not be obvious from the class and method names and hidden in the source. The maintainer can then focus on investigating the methods involved using the Class View as a means of navigating the source code and by using the Method Pixel View to investigate the calling information of the methods. The Variable Watch View can also be used in order to observe field accesses on fields of interest. Once the maintainer has located the error, its cause can be investigated using the views and the standard debugging options that DJVis supports, such as field modification watches and breakpoints. In order to fix the error, modifications will need to be made which may affect the rest of the software. The maintainer can then use DJVis to plan how to fix the error without introducing any side effects. The information needed for this task is dependent on the type and the location of the error. For example, if it is confined to the control flow of a method, then it does not affect the rest of the program, however, if it involves changing the interaction with some other classes, then the



maintainer will need to investigate this in more detail. This may involve the use of the Class View to inspect different inter-class relationships, such as references and inherits, as well as the annotation features of DJVis to record current hypotheses. The Variable Watch View can also be used to inspect how any class fields are used if they are in need of modification.

## **8.4.2 Scenario 2: Code Familiarisation**

### **8.4.2.1 Task**

A new developer has joined a team maintaining / developing a piece of software and needs to quickly gain an understanding of the software's source code and structure. For this scenario, the example program is a specific server for an input device. The software reads the device's outputs and serves it to clients that connect over a network. This is all the developer knows about the software. This is a very simple example with an easy architecture, however, a specific example program is used for this scenario, in order to demonstrate some features using a concrete example.

### **8.4.2.2 Information Requirements**

This scenario will involve finding out about the classes in the system and their relationships in terms of inheritance and referencing. The developer will then need to have an overall view of the role of the classes and the functionality they perform. The developer also needs to identify "key" classes i.e. those with a specific or important role within the program and investigate how they are used and how key actions are performed. This may involve studying source code and existing documentation on the classes. The structure of the code at the package level will need to be understood, as well as the identification of "helper/utility" classes. The developer will also need to gain an understanding of the threads the program uses and what their roles are.

### **8.4.2.3 Application of DJVis**

The developer runs the program within the DJVis tool. Once the program has been initialised and its main window has appeared, the developer pauses its execution in DJVis. Initially, the developer is presented with the Runtime View of the software, showing its threading details. This shows that the program uses a number of threads. The program uses three user threads as well as the standard system threads and the Java AWT threads. The main thread has completed its execution after setting up the application and starting the other threads. The developer investigates these threads, but they do not have meaningful names assigned, therefore the developer investigates the call stacks of the threads one at a time. Using the names and source of the methods on the stack the developer quickly sees that each thread has a specific role. One thread is for reading the outputs from the device, another thread listens for new socket connections from clients connecting. The remaining thread sends each registered client the device's latest outputs. The developer steps through the program at the method level to confirm the hypothesis on the role of the threads and to get an idea of which methods are called. The three threads of interest are dragged into the Query View so they can be compared easily, as the program is stepped through. This, is shown in Figure 8-4.



**Figure 8-4 Focusing on three threads of interest in the Query View**

The developer then investigates the classes of the program using the Class View. The developer notices that the package structure has some similarities to the threading aspects of the program, as there is a package that contains the code for interacting with the input device. The rest of the code is in the default package, except for a few classes that are in a *utils* package. The program has been studied before in the DJVis environment by the existing team members and the developer notices that there are some existing annotations on the classes and packages. They investigate this in the annotation view and see that the *utils* package has a comment saying that it provides utility classes for manipulating the readings taken from the input device. Using the annotations and the Class View the developer investigates the classes and their relationships. The method line mapping is used to show the length of the methods. This allows an idea of the complexity of the code to be gained. The developer decides they are not interested in the *utils* package, therefore, they abstract this into a single user group in order to reduce complexity of the graph. The developer also notices that the graph is heavily connected to a specific class, which they investigate. They discover that this class provides logging and error reporting facilities, which are written to a log file. The developer filters this class from the view, which allows for an improved layout of the other classes when showing references relationships within the Class View. The developer then hypothesises the role of the other classes based on their names, methods and the user annotations, as well as on the previous call stacks they investigated. However, in order to confirm this they decide to trace a number of key actions such as a new client connecting to the server. In order to do this they reset the temporary method and constructor counts before the action occurs. Once the action has been completed they then pause the program and inspect which methods have been called and which objects have been created using the Class View. If the developer is confident that they understand the threading structure, then they can also pause any threads, which are of no interest. However, this must be done with caution as it may affect the execution of the program and could lead to deadlock of the program. This tracing of method calls allows the developer to observe which classes are involved in the specific functionality and confirm or reject any hypothesis that they may have. The developer can also use the creation patterns of the objects to help them understand their use. Studying the creation of objects can give indicators on the type of the class. For example, a singleton object tends to suggest that the class is used to manage or store something, whilst a data structure or common helper class will typically have many instances of the class being created. The

developer can use the shading of the classes nodes to get an impression of the overall picture, along with the actual object creation counts if they require more details.

This scenario demonstrates the use of user annotation, abstraction and filtering. The visualisation allows exploratory navigation and investigation of the software under study.

## **8.4.3 Scenario 3: Preventative Maintenance**

### **8.4.3.1 Task**

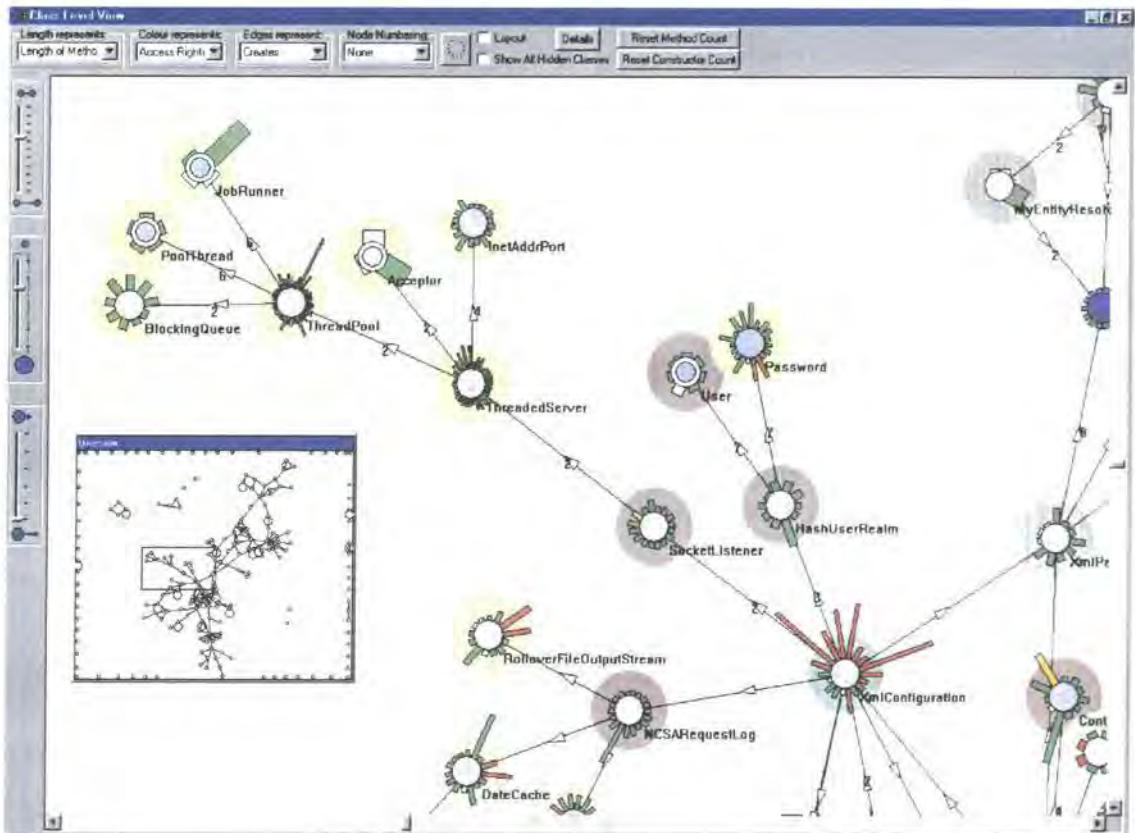
Restructuring of a heavily maintained program is planned. The aim is to improve the structure of the code to aid future maintenance. The maintenance team is particularly looking to remove redundancy of similar code and to try and reduce dependencies within the code by trying to separate out specific functionality. The maintenance also aims to reduce the complexity of complex classes wherever suitable.

### **8.4.3.2 Information Requirements**

The maintainers will require a summary of the classes in the software and will need to identify complex classes and heavily interconnected classes. They will also need to identify which classes are involved in different aspects of the functionality of the program and any possible overlap of functionality or code. The maintainers will be looking for classes that are closely related or similar, as such classes may be suitable for possible grouping or for extracting their commonalities into an inherited base class.

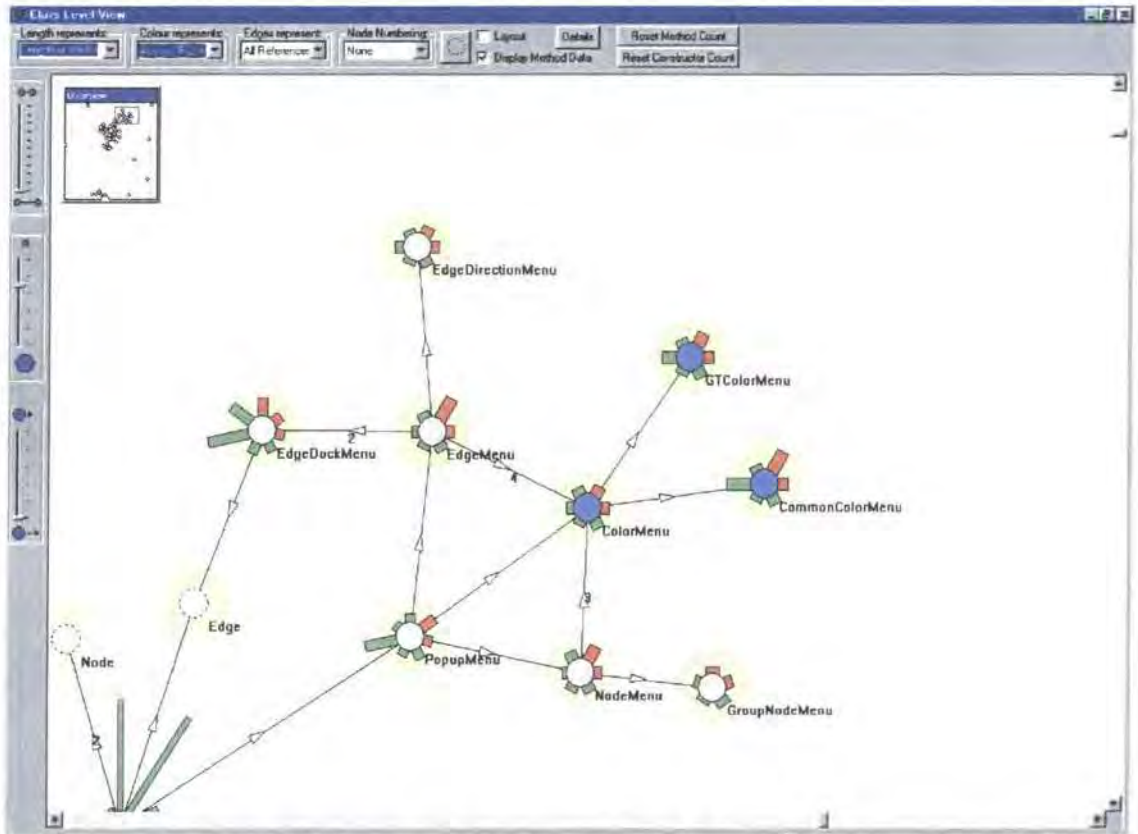
### **8.4.3.3 Application of DJVis**

In order to gain an overview of the classes involved, the software is executed from within DJVis. Once the program has been initialised and some operations of interest have been performed the maintainer can bring up the Class View. The view mappings can then be set to access rights for method line shading and to the length of the method for method line length. The resulting graph gives an overview of the software at the class level. Figure 8-5 provides an example of this, showing the class structure of a web server being investigated in the Class View. The maintainer can explore this structure using the overview window as a guide in order to look for certain characteristics.



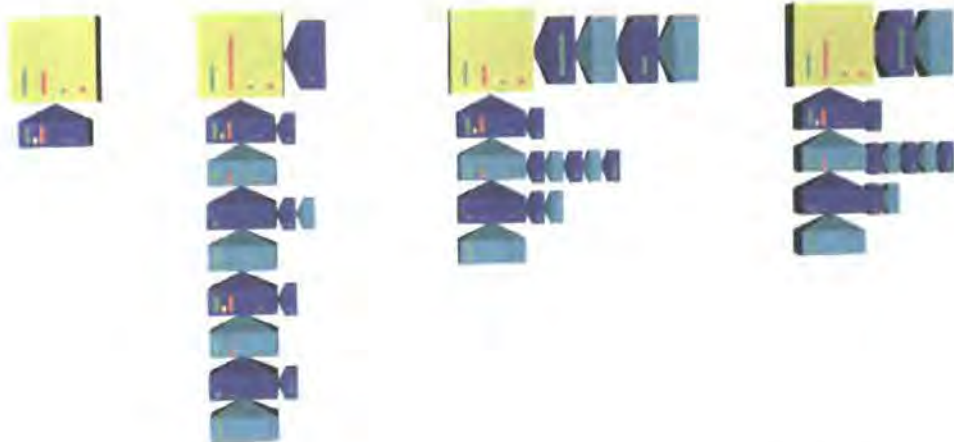
**Figure 8-5** Showing the class structure of a web server

The view can act as an indication of the complexity and coupling of the code, through how heavily interconnected the classes are. The method lines allow the relative length of the code to be compared which allows any anomalies to be seen, such as classes with a very large number of methods or methods of long length. These could be possible candidates for preventative maintenance and the maintainer could investigate this further by selecting the method of interest and browsing its source code as well as investigating its calling relationships in the Method Pixel View. The software can also be inspected in terms of the access rights of the methods. If nearly all the methods are public, then this could indicate that there is scope for making some private and restricting access to the class to a more defined interface. The maintainer can gain an insight into how the program is structured and how it uses interfaces and inheritance by changing the mapping in the Class View for the edges. If the code uses very little inheritance or interfaces then there maybe more scope for restructuring the existing classes to remove any replicated code into base classes, or through improving class usage by providing interfaces to ensure consistency. The name and shape of the classes may provide a visual clue to this, if they have similar names and similar method line characteristics and appear to cluster in the reference view. For example, Figure 8-6 shows a group of menu classes. These have similar method line patterns, particularly the red and the green shaded method lines of the bottom right of the menu node, which may form possible candidates for restructuring. If these methods are similar and they contain replicated code then this could be extracted and placed in a base class to be inherited by these classes.



**Figure 8-6 Inspecting for inheritance candidates**

Any candidates for restructuring can be investigated further by using the Class View to navigate the source code of the classes and methods. The maintainer can also investigate the existing inheritance aspects of the program using the Query View to show details on full inheritance and implements structure of a class as Figure 8-7 demonstrates.



**Figure 8-7 Inspecting the inheritance/implements structure of four classes in the Query View**

This scenario shows how the tool can be used for exploratory navigation and for investigating patterns within the views.

## **8.4.4 Scenario 4: Impact Analysis**

### **8.4.4.1 Task**

A piece of software is being modified and the return type of a method needs to be changed. The maintainer must assess how this will affect the rest of the program and what other changes, if any, will be necessary.

### **8.4.4.2 Information Requirements**

The class under study is already known, therefore it is a case of assessing how the change in this class' method will affect the rest of the software. This will involve looking at the coupling of the class to other classes and the calling information for the method. The calling methods will need to be assessed to see what they do with the returned value in terms of storing it, or using it in interactions with other classes. The fields of a class may need changing, or some local variables in the methods, which are calling the changed method, may need to be modified.

### **8.4.4.3 Application of DJVis**

In this instance the class to be changed is known in advance, therefore it can be searched for directly within the visualisation, using the named search options. This allows classes to be searched for by name and presents a list of all the classes, which is scrolled to match the characters of the name inputted so far. Once the class has been located, the method that will be modified can be inspected in the Method Pixel View to inspect its calling relationships. Using this view, the maintainer can see which methods call this method and then browse their source code to see how the returned value is used. The Method Pixel View may then be used to find out their calling relationships and to check the arguments of these calls to see if the value is passed. Figure 8-8 shows Method Pixel View being used to gain an overview of a class' method calling relationships. This gives an impression of the calls and called by relationships for each method and a particular method of interest can be selected to focus on it and to display the names of the related methods. The temporary annotation feature can be used to store comments on the type of change needed, while the maintainer investigates the impact on the other methods and classes. If the new return value is placed in a field variable then the field can be investigated using the Variable Watch View to observe the pattern of accesses and locate related code.



**Figure 8-8** Gaining an overview of the method calling relationships for the Lexer class using the Method Pixel View

This is one task where the visualisation could benefit from being coupled to a repository of statically parsed information on the software. In this scenario, the method trace presented (and therefore impact tracing) is limited to methods calls that have actually occurred in this execution, rather than all possible calls. In this case, it would be beneficial to be able to present method calls that are not made in this execution, as these will also be affected. This also applies to watching fields for modification, as here statically parsed information could again be used to allow all known access to be easily presented. This demonstrates the importance of the user remembering that the information presented in the views is only based on the current execution. It cannot therefore be assumed to be the case for all possible executions. The incorporation of a static repository of information on the software being visualised could increase the benefits of the visualisation, for example in instances where information regarding all possible execution sequences is desired.

## 8.4.5 Scenario 5: Test Case Validation

### 8.4.5.1 Task

A tester has been given a number of test cases, which test specific parts of a piece of software. The tester wants to validate the test case inputs, in order to confirm that they cover the reported areas of code.

### 8.4.5.2 Information Requirements

The tester will be given the inputs to the program and the details of which areas of the code they are designed to test. Given this task, they need to identify if these areas of the code are tested by checking the anticipated code usage against the actual code usage.

### 8.4.5.3 Application of DJVis

DJVis does not directly support testing or the development and evaluation of test cases. However, it can allow the actual code usage to be shown, which can then be checked against the code that the test case is designed to exercise. This checking can only be done at the method level and not at the individual line level. Using DJVis the tester can run the program with the given test data to the point of program termination. The Class View can then be invoked to show class level details. Using the method line length to represent the total number of calls of a method, the tester can see which methods are called allowing them to easily check if the correct areas of code are exercised by the test case. If the tester wishes to check a specific class for its method calls, then they can locate the class using the "find by name" menu option in the Class View. The custom mapping feature of the method lines can aid the investigation, if the tester wished to identify code not called by the test case. In this case, the mapping can be set to be a substantial length for zero calls while one or more calls are mapped to a very short length as shown in Figure 8-9.

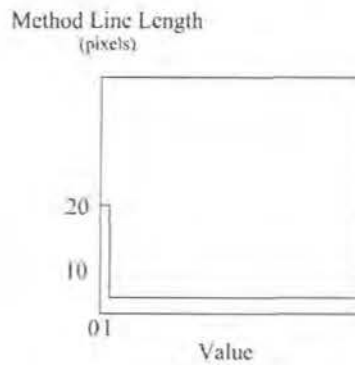
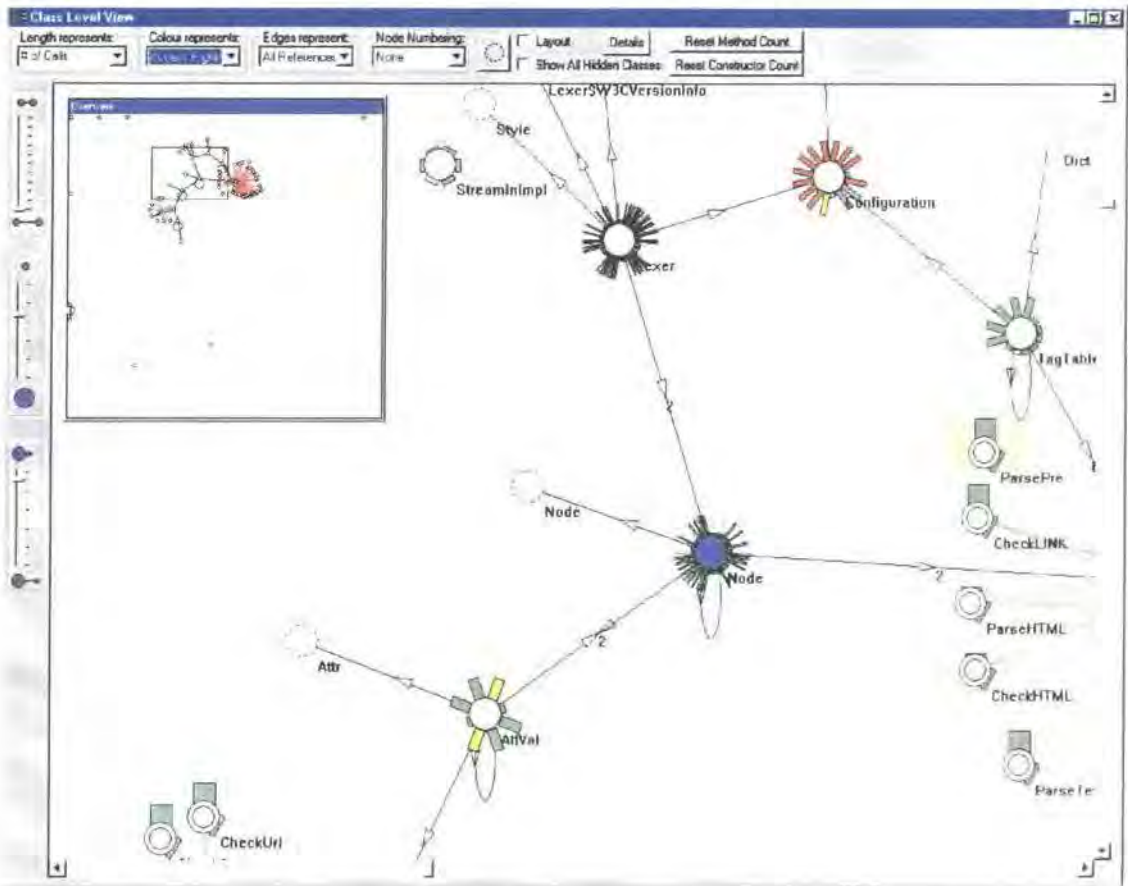


Figure 8-9 Custom mapping for the easy identification of methods which are not called.





**Figure 8-10 Highlighting uncalled methods**

Figure 8-10 demonstrates the effect of using the custom mapping on the JTidy [JTid] HTML syntax checking software, as it parses the standard input. Methods that have not been called are clearly visible, due to their long length compared to the called methods. This shows classes that still need exercising. The Lexer class (top middle) has a large number of methods, many of which have not been called. This is also true of the Node class (centre). There are only a limited number of classes that have had all their methods called, such as StreamInImpl (top middle) and CheckHTML (bottom right). The example program used in Figure 8-10 also demonstrates extensive use of dynamic binding as the large number of red edges in the overview window shows. It can be seen that this is mainly in two clusters, which are connected through a small number of other classes.

This scenario demonstrates how the custom mapping feature allows the visualisation to be tailored to a number of tasks. It can allow the user to specify the type of pattern they wish to see in the calling information (or other length metrics). This also allows the user to filter the display by removing values of no interest through specifying a zero length for those items.

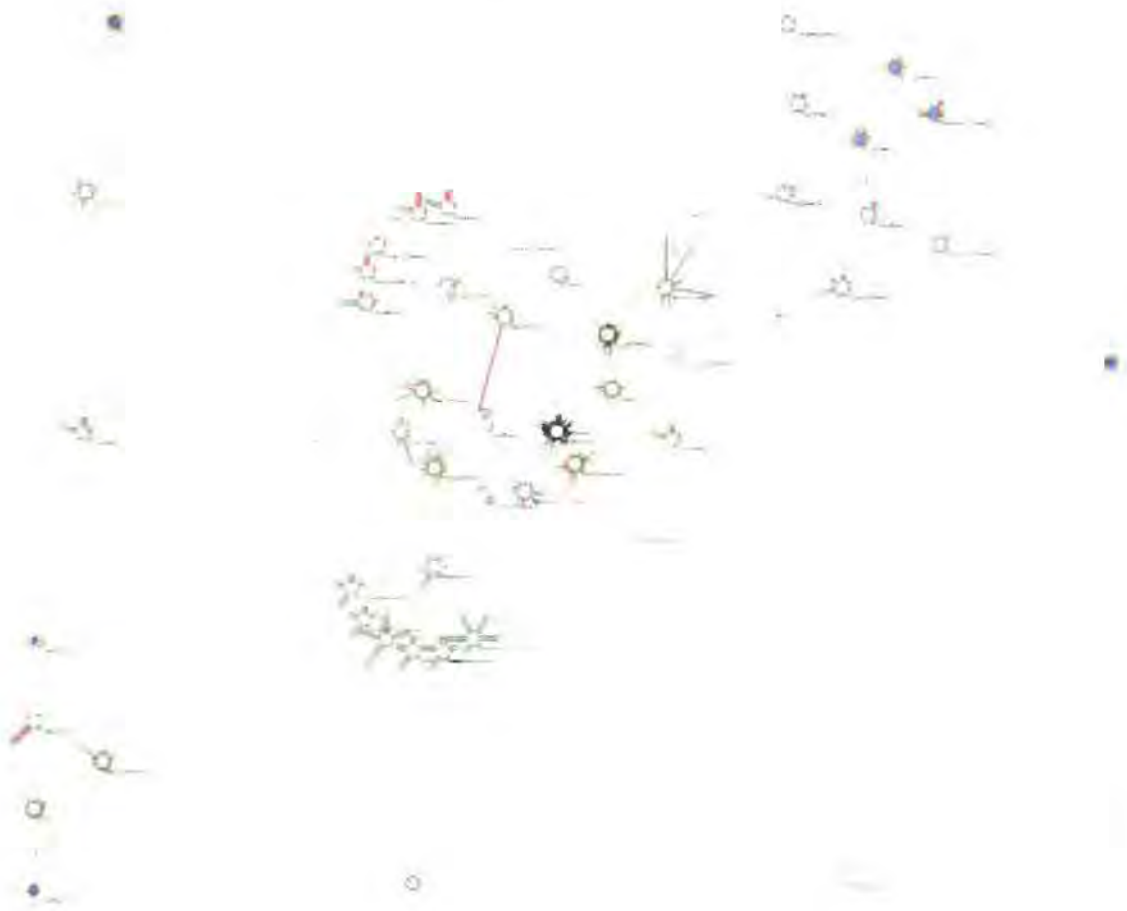
## 8.5 Case Study: Understanding GraphTool

The previous examples have all looked at theoretical software systems or tasks. This scenario will focus on analysing a real piece of software called GraphTool. GraphTool is a graph editing tool that provides some simple layouts and the ability to group nodes and edit graph display properties. It is approximately nineteen thousand lines of Java and so represents a small to medium scale application. The tool is used

internally within the Department of Computer Science at the University of Durham and was rewritten in Java in 1999. It is made up of eighty six classes defined in sixty six source files. The author had no experience of the source code before applying DJVis to it and only very limited experience of using GraphTool. Therefore, all knowledge gained about its structure came through the use of the visualisation and not through experience of the source code or through the use of other tools. The task set was to find out the overall structure of GraphTool before investigating how new layouts could be added to the tool.

### Application of DJVis

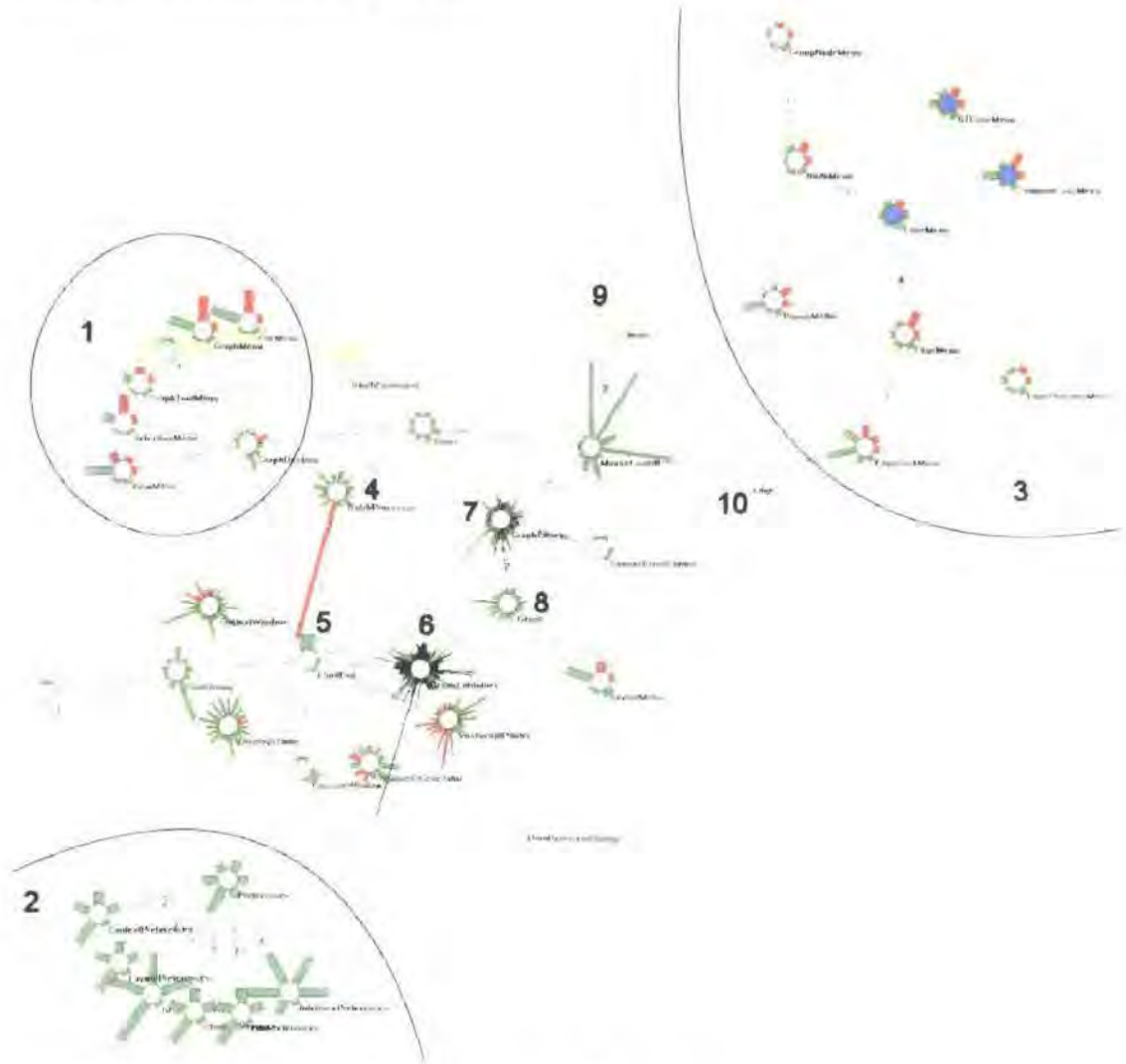
To start the investigation into the structure of the software, it was run from within DJVis. The Class View was used to inspect which classes are used by the application and to investigate the complexity of the class relationships. Figure 8-11 shows the graph as presented in the Class View at the point when the tool had been initialised and is showing its user interface.



**Figure 8-11 GraphTool classes after initialisation as shown in the Class View**

Figure 8-11 shows that there are two sub graphs and a number of unconnected classes. In this view, the edges represent class references, therefore the unconnected classes are not referenced by other classes through the use of field references. These unconnected classes appear to be utility classes with limited functionality. The main focus of the investigation into the software's structure will therefore focus on the two sub graphs. The small sub graph in the bottom left of Figure 8-11 contains five classes. Inspection of

the class names (and optionally the method names and source files) indicates that these classes are used for lexical analysis and for the management of tokens in a linked list. There is no other functionality provided by the classes and they do not seem to be heavily interconnected with the other classes. Therefore, this sub graph does not appear to be of any real interest and can be abstracted or even filtered from the tracing as it produces a large number of method calls for the list and token operations. The large sub graph, therefore, appears to be the main item of interest and this presents a number of interesting features, which are labelled in Figure 8-12.



**Figure 8-12 Identification of interesting features in the main sub graph.**

The features identified in Figure 8-12 are numbered and each is discussed below

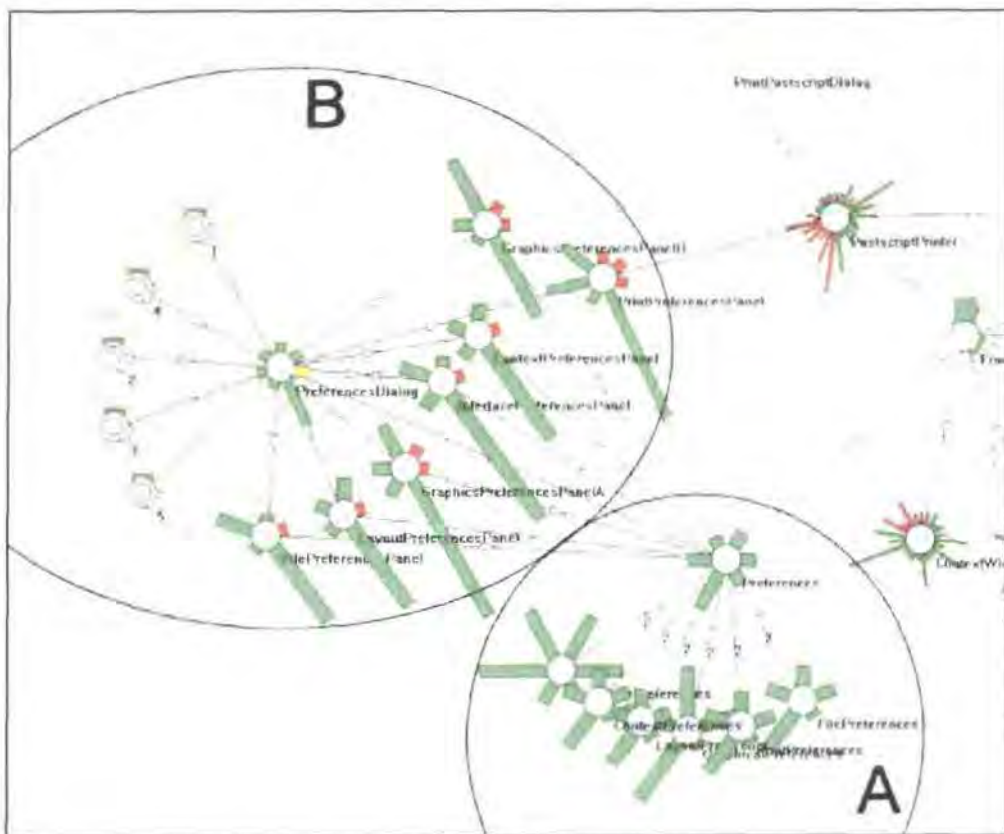
### **Feature 1**

There is a cluster of classes from the central GraphDesktop class. Their names all end in the word "Menu" and closer inspection of the actual GraphTool user interface shows that the names correspond directly to the actual menu headings in the main windows title bar. Therefore, one can hypothesise that these classes implement the main menus for the Tool. Inspection of the method names in Class View also supports this hypothesis and using the Query View the classes can be inspected to show their full inherits and implements structure. This shows that they all inherit from the Java API class "JMenu" and implement "ActionListener". The only exception to this is

the inner class "1" which is also in this cluster.

## Feature 2

Feature 2 is another cluster of classes. This one is centred on the "Preferences" class and this class references six classes, whose names all end in the word "Preference". This naming would therefore suggest that they are responsible for the preference options and closer inspection of GraphTool user interface reveals a preference option under the GraphTool menu. This brings up a dialog box that has six categories that relate to the names of the classes. The classes in the cluster have a similar shape and inspection of the method names shows that they all provide the same methods (load, save, copy and set defaults). However, changing the edges to show inherits and implements relationships shows that this is not enforced through the use of an interface. As GraphTool displays the Preferences dialog box, a number of classes are loaded and these changes are shown in the Class View.



**Figure 8-13** Classes loaded as a result of displaying the Preferences Dialog

Figure 8-13 shows the result of opening the Preferences dialog box on the Class View display. The view shows that new classes have been loaded by the JVM and these are shown in the annotated group "B". The existing preferences classes remain (shown as group "A") but the central Preferences class of the group is now referenced by many of the new classes in group "B". It can be seen from the class naming that these new classes have the same names as the classes in group A except they have "Panel" on the end of the names. These classes therefore handle the user interface panels for the other classes, which actually contain the data for each subset of the preferences. These 'panel' classes also have one method that is significantly longer than the rest

and investigation of this, using the mouse over details, shows that this is the constructor method for each of the classes. This example highlights how runtime information can be used to filter the classes under study, as classes are only loaded and therefore presented at the point they are needed. Therefore, if the user were considering some other aspect of the GraphTool software, they would not have to consider these additional preferences classes. A static analysis of the software would present all these classes, which could add complexity to the resulting visualisation.

### **Feature 3**

This cluster of classes handles menu code and popup menus as suggested by the names of the classes and the method names. The classes contain relatively little code (shown by the short method line lengths) and most of the method names are repeated across the classes as they all implement the ActionListener and ItemListener interfaces of the Java API.

### **Feature 4**

The BatchProcessor class has one very long private method which when inspected using mouse over is called "processCommand". The class appears to support batch processing from its name and the names of its methods. The "processCommand" method has yet to be called, which can be seen by changing the method line colour mapping to represent the number of calls. If the user is interested in this possibly anomalous method they could use the Class view to navigate to the source. This shows that the method is in fact a long sequence of embedded "if" statements, each of which handles a different command type.

### **Feature 5**

Feature 5 is the class "FrontEnd". This is a static class (shown by having no instances (white class node)) and it references many of the main classes in the program. This would appear to be a central class through which the other classes are joined. If the Class View is open while the GraphTool program initialises, then it can be seen that this class is loaded second and then all the classes it references are loaded. Closer investigation shows that the GraphTool class (the initial class containing the main() function) is just a wrapper for FrontEnd. The FrontEnd class' role is to create and initialise the main classes of the software and to act as a central point to reference the other important classes.

### **Feature 6**

Feature 6 is the class GraphContainer. This stands out as a class with a very large number of methods, some of which are long in length. It has ninety-seven methods, however, only nine of these are private suggesting that this large amount of functionality is offered to other parts of the program. Also, the class only has two fields (identifiable by changing the display from method lines to field triangles) suggesting that it operates on data provided by other classes, and in particular, the graph class which it references using one of the fields. The class appears to have numerous methods that operate on the graph, and is therefore, a candidate for further investigation for the layout operations

### Feature 7

The GraphCanvas class also has a large number of methods and would appear from first impressions to be the second most complex class after GraphContainer in terms of the number of methods. However, in the case of GraphCanvas a large number of its methods are short in length and investigation of the method names shows that class is mainly concerned with displaying the graph, as the Canvas part of the name suggests.

### Feature 8

The class Graph would be expected in an application focusing on graph display and editing. One may expect that this would hold a large amount of functionality on managing and updating the graph. However, first impressions suggest that this is not the case for GraphTool as the class has relatively few methods and most of them are short in length, therefore the class does not represent a significant amount of the code base. Inspection of the method names or source shows that the class provides basic addition and deletion of nodes and edges, however no layout or other graph operations are provided. Switching the view to display the variable details of the classes allows the Graph class to be investigated further.

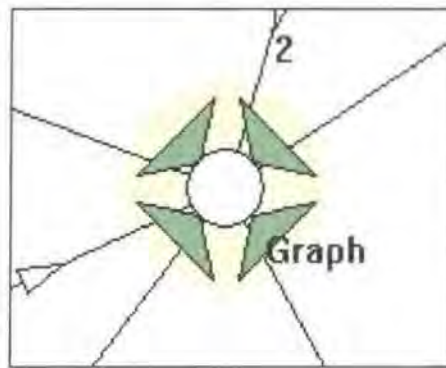


Figure 8-14 Displaying the field information for the Graph class

Figure 8-14 shows that the Graph class has only four fields. These can be investigated by selecting them, or using the mouse over facility as shown in Figure 8-15.

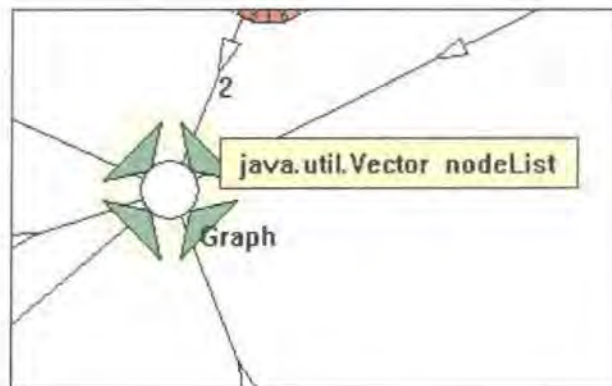


Figure 8-15 Investigating field names and types using mouse over information

Figure 8-15 shows that one field of the Graph class is a vector called nodeList. The class also has

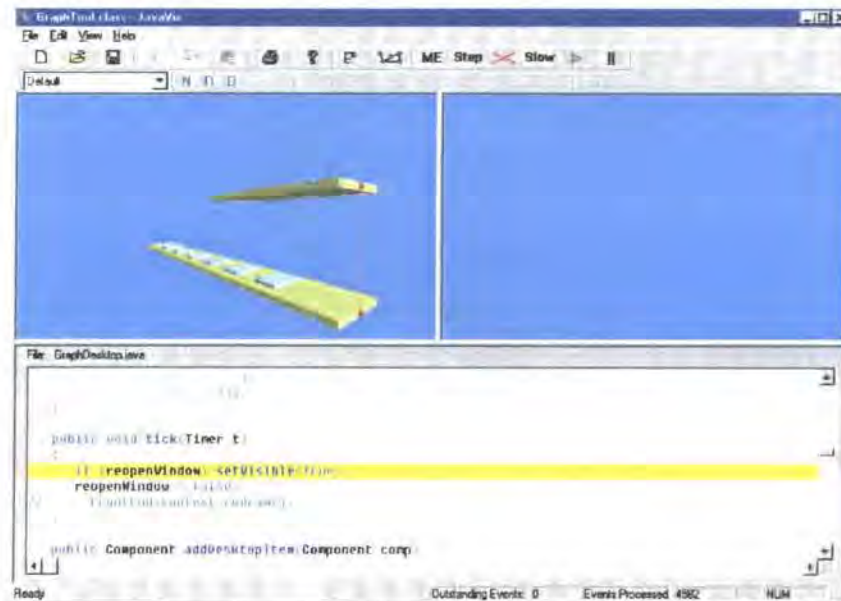
a `edgeList` vector, therefore from these names and the simple `addNode()` and `addEdge()` methods of the class, the user can see that these vectors store the nodes and edges of the graph. However, it can be observed from Figure 8-15 that these fields are public (shown by the green shading of the field triangles) and can therefore be modified by other classes. This could indicate that the graph functionality may be dispersed over a number of classes and this class could be heavily coupled to the other classes due to these public fields. The small amount of functionality provided by this class is obvious from the visualisation. The far greater complexity of the `GraphContainer` and `GraphCanvas` classes provides a cue to the user that it is these classes that need to be focused upon.

### **Features 9 and 10**

There are "Node" and "Edge" classes, as one would expect in a graph application. These classes have yet to be loaded by the JVM, as they have not yet been needed. This is shown by the dotted line of the class node. When these classes are loaded, then the details of the classes will be presented.

The initial view also allows an idea of the programming techniques to be assessed. The user can see that the package circles are all the same colour therefore indicating that the code is all in one package, which in this case is the default package. The references (when selected as the edge type) are all static. This is combined with minimal use of user defined interfaces, in fact, only one "Timed" is used which is implemented by three classes. There is also no inheritance between the user-defined classes. This relatively quick investigation of the program allows the user to gain some idea of its structure, in terms of the classes and their relationships.

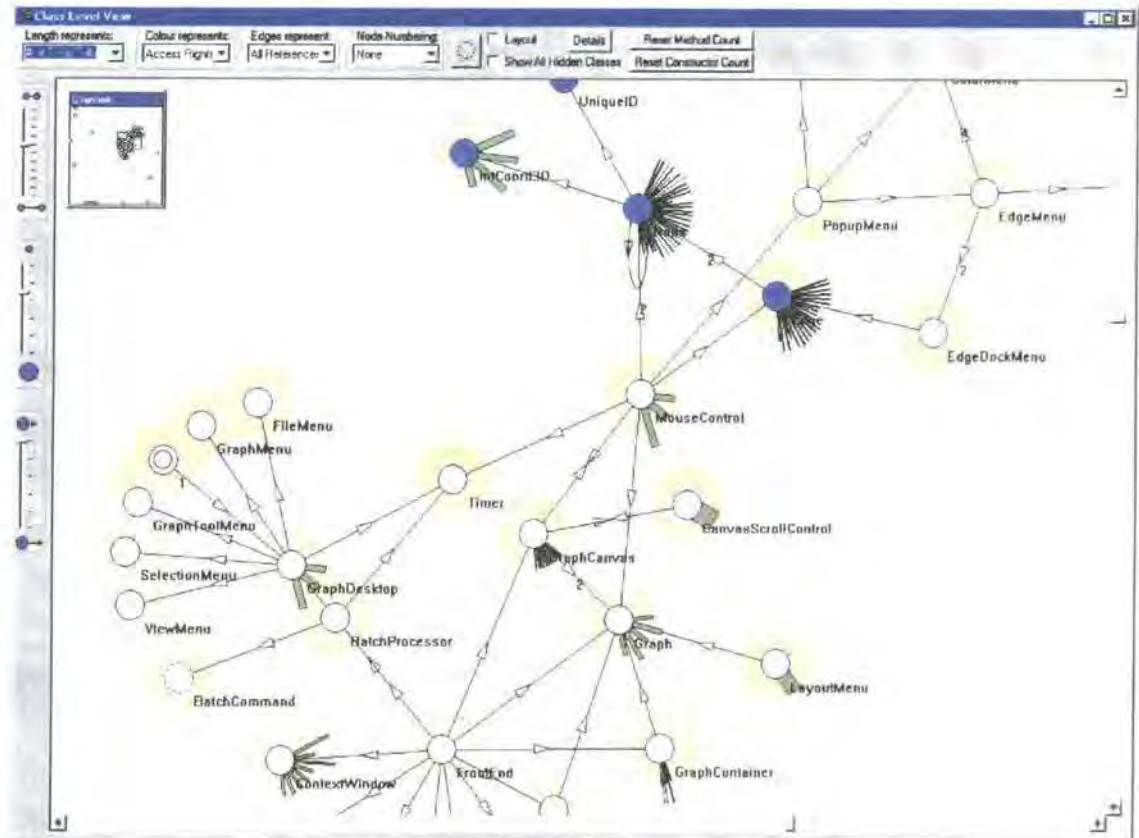
Investigation of the threading aspects of the program can be done through the use of the Runtime View as shown in Figure 8-16 . Using this, it is possible to see that the program does not explicitly use thread groups, however, it does use a number of threads. The main thread exits after initialisation of the software and a number of other threads remain. These include the system threads for handling the user interface aspects of the program, as well as three user created threads. Investigation of these threads, using the Runtime View, shows that they are related to the three classes that implement the Timed interface and these threads are from the Timer class, which uses the Timed interface to call a `tick()` function for these three classes. These classes perform background tasks, such as checking for batch commands and are not related to graph layout.



**Figure 8-16 Investigating the threading structure of GraphTool**

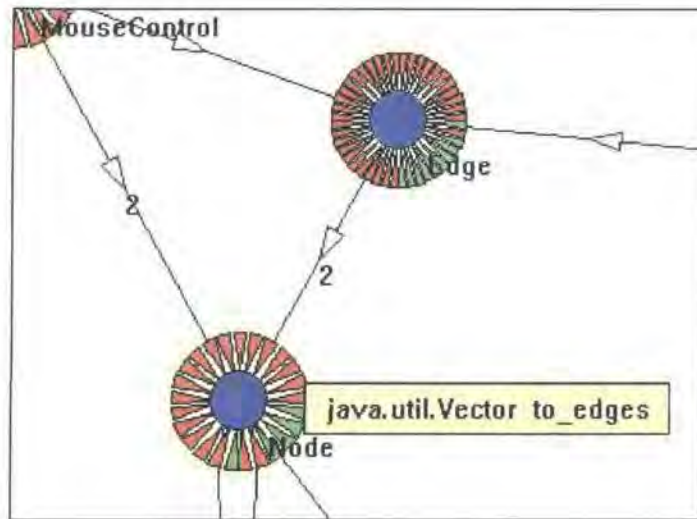
The threading aspects of the program do therefore not seem to be relevant to the addition of new layouts to the tool. The user focuses again on the Class View with the new aim of investigating the details of how the existing layouts are performed and how new layouts could be added. The initial investigation of the Class View highlighted some possible areas for the investigation of the layout functionality. The software offers two graph layouts so the user decides to trace the execution of each of these to observe the classes and methods involved. Using the Class View set to display the method lines as the number of calls and the mapping function set to group the number of calls into three groups, the program is traced for each of the graph layouts on the same graph data. This shows that the first graph layout (grid) was much simpler in its implementation than the second layout (horizontal) in terms of the number of classes and methods involved. This layout implementation therefore offers a good starting point for the user to investigate how they could add additional layouts. The Method Pixel View can be used to investigate the calling structure of the methods identified by showing the calling relationships between the individual methods.





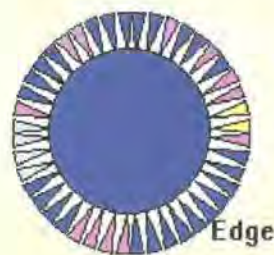
**Figure 8-17 Showing the methods involved in the horizontal layout functionality of GraphTool**

Figure 8-17 shows the calling structure for the more complex second layout. The calling structure of the grid layout functionality required a slightly smaller number of classes and interestingly did not call any methods of the Graph class. It also called a smaller number of methods, particularly of the Node and Edge classes as well as the GraphContainer class. The Node and Edge methods are called extensively and without the custom mapping they would introduce too much visual complexity within the display. The length mapping used in Figure 8-17 categorised to only three levels and therefore only offers an overview of the calling count, however, a more fine grained mapping could be used once the user has grasped the basic pattern. The user can also trace the calling structure of individual methods using the Method Pixel View. Using this, the user can follow the individual calling, for example observing the calling of the layout method from the LayoutMenu actionPerformed() method. This also allows an impression of the interconnectivity of the method to be examined in terms of the number of methods that it calls and the number that calls it. Using this, the user can locate the method that actually performs the layout and this is a member of the GraphContainer class. The Node and Edge classes are heavily used in the layout of the graph, as one would expect. The user can investigate these further by examining their methods, particularly those that are called in the layout and also the fields of the classes.



**Figure 8-18 Investigating the fields of the Node and Edge classes**

Figure 8-18 shows a section of the Class View showing the fields of the Node and Edge classes. The access right mapping is used for the shading of the field triangles. Both classes have a large number of fields and the majority of these fields are private, as shown by the red shading. The user is investigating the public fields of the Node class and as the pop up information shows<sup>5</sup> this is a Java API Vector class called `to_edges`. The other public fields include a string for the name of the class and another vector called `from_edges`. The class therefore allows open access to its edges in the graph. The private fields of the class store the position and visual settings of the node, such as colour and visibility settings. The Edge class has numerous fields, so the user focuses on this and changes the shading of the triangles to represent the type of the field.



**Figure 8-19 Close up of Edge class showing field types**

Figure 8-19 allows the fields of the Edge class to be observed more clearly. The blue represents fields of primitive types, purple represents references to Java API classes and yellow represents references to user classes. It can be seen that the group of public fields (Figure 8-18) are in fact all fields of primitive types (Figure 8-19). Closer investigation shows that these are all static variables that act as a mapping of edge settings to named values. The other fields of the Edge class define its position and appearance along with

<sup>5</sup> The image capture software used to acquire Figure 8-18 has removed the mouse cursor. The mouse cursor would be displayed to the user using the tool, therefore allowing them to relate the popup information to the graphical item being inspected.

any arrows on the edge. The use of public fields for the Nodes edge list and the similar use of public fields for the Graph class result in a highly coupled collection of classes for graph functionality. The user may also want to consider improving this structure, if they plan to use the code and wish to maintain it, or add further functionality in the future. The user can observe the usage of these public fields by watching them using the Variable Watch View. This allows the user to locate any code that uses these public fields. For instance, Figure 8-20 shows the usage of the `from_edges` vector in the Node class in a trace that uses most of the functionality of GraphTool. It shows that five classes access the vector, while the method lines show the number of methods in each class which access the field. It can be seen from Figure 8-20 that the `from_edges` vector is used in particular in a large number of methods of the GraphContainer class.



**Figure 8-20 Watching the `from_edges` vector of the Node class in the Variable Watch View**

The tracing of the layout operations is also a suitable point to introduce a field watch and the user may want to observe the accesses of the node and edge lists of the Graph class using the Variable Watch View. This would allow them to see which code uses these lists through the layout of the graph. Another option the user could have used to locate the layout functionality would be to filter the methods displayed to those with names containing the text “layout” through the use of a user filter. This would offer a quick approach to locate possible code of interest, however, it is obviously highly dependent on the naming used for the methods and does not show other code that is used in the layout operation. It does however, highlight a way in which a user may quickly try to detect “important” methods for some given functionality, without the need to trace the code.

Investigation of the trace allows the user to focus on the pattern used by the basic grid layout. Many of the method calls shown in the trace, particularly those of the Node and Edge classes, are actually called in the graphical update operation after the layout. Once the user has identified the method that performs the layout, they could retrace the operation and restrict the tracing to this method to just see the code used for the layout functionality. This is not needed in this simple example, but it demonstrates how the user can reduce the size of the trace through an iterative process as they gain greater understanding of the code structure.

Using the information gained through the use of DJVis, the user can plan how to add their own layouts to GraphTool. This case study has identified the overall structure of the GraphTool program and highlighted how the existing layout functionality is implemented. This allows the user to draw up a plan of action that would be needed for the addition of new layout functionality. This would involve:

- Addition of new layout options to menu handler (LayoutMenu class). This would involve adding the new layout text to the menu and updating the class' actionPerformed() method to handle the new item, by calling the new layout method.
- The entry point method of the new layout functionality would need to be added (to GraphContainer if the existing pattern is followed). This method will be responsible for performing the actual layout and may use additional private functions to separate the functionality. This may involve the use of the node and edge vectors of the Graph class as well as the from\_edges and to\_edges vectors of the Node class. The resulting changes in position of the nodes should be done using the setPosition() method of the nodes, as this handles the update of the nodes and the positioning of child nodes if the node represents a group.
- User Options relating to the layout will need to be stored in the user preferences of GraphTool. This will involve the update of the preference classes. This includes updating the LayoutPreferences class, which stores user settings, through the addition of new fields to store any settings on the new layout algorithms. This change, will then result in the need to update the setDefaults(), load() and save() methods of the class in order to incorporate the new settings. This will allow the new settings to be stored however, as identified in Figure 8-12 feature 2, these settings are displayed using the PreferencesPanel classes and therefore the LayoutPreferencesPanel will need to be updated to display the new settings in the preferences dialog.

This plan of action for the addition of new layout implementations resulted from an understanding of the GraphTool software, which was gained through the use of DJVis. This case study of a specific maintenance task, on a particular piece of software, has demonstrated how DJVis can be used to aid the comprehension of a real piece of software. It highlighted some interesting features about the overall structure of the GraphTool program and showed how DJVis could be used to investigate these identified features further, as well as helping with a specific task. This case study and the previous scenarios do not aim to show all the features of DJVis or the only way the tool can be used to address these tasks. However, the scenarios and this case study offer an insight into the use of DJVis on a variety of tasks, using some of the core features of the visualisation.

## **8.6 Conclusions**

This chapter has evaluated the ideas presented in this thesis. The evaluation is presented in three distinct parts. Firstly, DJVis and the implementation techniques used for the prototype tool were evaluated informally through a detailed discussion. Secondly, DJVis was evaluated against two feature based evaluation frameworks. Finally, the use of DJVis was demonstrated using five scenarios and one in-depth

case study on a real software system. Whilst these scenarios and the case study do not explicitly evaluate DJVis, they do allow the reader to gain an understanding of DJVis, thus allowing them to assess the merit of the approach on the tasks presented.

The informal discussion of DJVis allowed a number of aspects of the theoretical visualisation and prototype implementation to be discussed in detail. The discussion describes the author's assessment of the visualisation and areas where they felt improvements were needed. A number of proposed improvements and additional features were presented which could improve the current visualisation. While this discussion is obviously subjective, it provided a means of imparting experience of DJVis that had been gained through its development and application to real Java software. Much of this information would not have been evident from the feature-based evaluation alone. This informal evaluation also addressed the implementation techniques used for the prototype implementation of DJVis, and in particular, looked at the effectiveness of the Java Platform Debugger Architecture for software visualisation. The visualisations are not restricted to this particular information extraction method and this could have been combined with other techniques such as class file annotation to trace the method calling structure. The use of JPDA incurred a large performance overhead as it was used to trace method entry and exits. This is obviously not what it was designed to do on a large scale and such functionality would not be used continually in a standard debugger. This overhead could be limited by filtering the method entries and exits, although even filtered methods result in a performance cost, as the JVM must still check the event against the current filters. However, the use of the JPDA has successfully allowed Java programs to be easily visualised, without the need to modify their source code. The informal evaluation discusses such issues in more depth, along with the achievements and issues of each of the views that comprise to make DJVis.

The feature-based evaluation of DJVis highlighted its support for multiple investigation methods, thus allowing it to support a variety of users. DJVis achieved positive results from the frameworks that indicate its suitability for program comprehension tasks. The frameworks did however demonstrate areas for improvement and further evaluation. An empirical study would be useful to evaluate the use of DJVis in an industrial setting and would prove a worthwhile extension to this work.

The scenarios demonstrate how DJVis and the runtime information it presents can be used to aid program comprehension activities. Each scenario, demonstrates how DJVis can be applied to a specific problem and how the views can be used together to allow new understanding to be gained about the software under study. The scenarios conclude with an in depth case study highlighting the application of DJVis to a specific piece of software. This case study provides much more detail than the previous scenarios and shows not only how DJVis can be used to gain an understanding of a piece of software, but also how it can be used to help plan modifications to the software. The scenarios also highlighted the deficiency in relying solely on dynamic information for some tasks. While it is perfectly suited to some tasks, for example scenario 1 (corrective maintenance), other tasks such as scenario 4 (impact analysis) could have benefited from statically parsed information. This is due to the visualisation only presenting the information for the present execution and not for all possible executions. Therefore, in scenario 4 the

impact analysis was restricted to the present execution and other areas of the code may have needed to be changed as well.

This evaluation has highlighted the lack of any specific runtime visualisation evaluations, this is to be expected, given the poor state of software visualisation evaluation. Currently, there is lack of evaluation frameworks for generic software visualisation, let alone specific areas such as runtime visualisation. A number of factors to be considered in the design and evaluation of runtime visualisations were presented in section 8.3 and these could act as a building block for further work in this area.

# Chapter 9 Conclusions

## **9.1 Introduction**

Program comprehension is a significant and time-consuming aspect of maintenance activities. The understanding of the software is typically based on insights gained through studying the source code of the program. Documentation on the software and indeed the software itself may be poorly maintained and structured, due to the pressures of project deadlines and repeated maintenance and modification. Such problems affect the maintenance of any piece of software and when this software is object-oriented in nature, there are other challenges to overcome in gaining an understanding of its structure. This is due to the discrepancies between the static description of classes and the network of interacting objects that actually exists at runtime. Despite the demands of program comprehension on resources, there exists very little in the way of tool support for program comprehension.

Software visualisation offers an approach to aid this activity by allowing maintainers to deal with the large volume of abstract information that program comprehension tasks involve. Research into software visualisation is still in its infancy and a variety of challenges exist, such as finding suitable representations for the abstract information that software presents and dealing with the scale of real world software.

In order to deal with the discrepancies between the static specification and dynamic behaviour of object-oriented software, the visualisation of such systems can benefit from the use of dynamic information on the program. This visualisation of runtime data can be combined with details of the static specification to provide greater insights into the actual operation of the software and aid users in the understanding of its structure. This runtime visualisation presents a number of challenges in terms of how to extract and visualise the vast volume of information involved and how to provide familiarity in the face of constant change.

## **9.2 Summary of Research**

This work has focused on the application of visualisation techniques to runtime information in order to aid program comprehension of object-oriented software. The work focused on object-oriented software due to the particular discrepancies that exist between its static description and runtime structure. The aim was to highlight the dynamic aspects of a piece of software, alongside the information extracted using traditional static analysis, in order to allow a greater understanding of the software to be gained. This was achieved through the development of the DJVis approach, which proposes new representations for different aspects of an object-oriented system at runtime. DJVis and each of its views were fully described in Chapter 5 with particular focus of visualising Java. This work focuses particularly on class level interactions within the software and hence the central role of the Class View. DJVis does not attempt to display all runtime information on the executing software. There are many facets of a program's execution that could still be visualised, and in particular, it would be interesting to extend this work into visualising at the object level and also higher level views of the system, possibly at the architectural level. Runtime visualisation is still a relatively unexplored area and there remains many



research challenges within it. These issues and challenges are discussed throughout the thesis and in particular in sections 4.7 and 8.3.

Chapter 6 provided an overview of the implementation of a proof of concept tool to demonstrate the main concepts of DJVis. This tool provided a means to apply the representations of DJVis to real world Java programs. This allowed additional experience of their benefits and issues to be gained and this was incorporated into the informal evaluation in Chapter 8. The ideas generated in this research were evaluated using a number of techniques in Chapter 8. This included an informal discussion of the theoretical visualisation and the implementation approach taken, as well as a framework-based evaluation. The chapter also demonstrated the use of DJVis through a number of scenarios and through one in depth case study. This case study showed how it could be used for a specific task, based upon a real piece of software.

The issues and research challenges facing software visualisation and in particular software visualisation at runtime have also been discussed in chapters 3,4 and 8.

The main contributions of this work can be summarised as:

- Provided new visualisations of object-oriented software, which highlight the dynamic aspects of the software.
- Illustrated how runtime information can be used for program comprehension activities on object-oriented software.
- Highlighted the need for further work in the visualisation of runtime information.
- Demonstrated the use of debugging information for the automatic visualisation of Java programs.
- Developed a prototype tool to implement the ideas of DJVis, allowing them to be realised on real software systems.
- Assessed the use of JPDA for runtime visualisation of Java.
- Discussed issues that should be considered in the design and evaluation of runtime visualisations.

These contributions tie in with the criteria for success detailed in chapter 1

### **9.3 Criteria for Success**

This thesis defined a number of criteria for success in chapter 1. These are now revisited to contrast the achieved research against the initial criteria. Each of the original criteria is presented, followed by a discussion of the extent to which it has been achieved.

#### **a) Address the visualisation issues of representing an object-oriented language such as Java at runtime.**

The issues involved in representing a software system at runtime have been discussed throughout this thesis. The challenges faced by any software visualisation were initially introduced in section 3.6, before the issues, which are involved in visualising runtime information, were discussed in Chapter 4. These involve new intricacies to the generic software visualisation issues, as well as new

considerations due to the dynamic nature of the data. The runtime issues included a discussion on the various aspects of runtime visualisation, from techniques and issues involved in extracting the necessary information, to overall trends and challenges faced by a runtime visualisation system. For instance the problems of scale and dealing with a constant change in data in an efficient way are discussed in section 4.7. The specific details of visualising an object-oriented language such as Java are discussed in Chapter 5 and in particular in section 5.3. Chapter 6 then highlights the details of the technique used to extract the information required to generate the visualisations detailed in Chapter 5. The evaluation of DJVis highlighted the lack of consideration for runtime visualisation issues in the current frameworks due to their generic software visualisation focus. A number of considerations for the evaluation and design of runtime visualisations were presented and this can be found in section 8.3. This focuses specifically on runtime visualisation of software systems and addressed issues such as whether the visualisation presents the current state of the program, as well as a summary of previous states.

**b) Develop new representations of object-oriented system at runtime.**

This thesis has defined a number of new representations for the visualisation of object-oriented software at runtime. Each visualisation forms a view of some aspect of the runtime behaviour of the software under study and it is these views that makes up DJVis. The main views are the *Runtime View* (shows threading aspects), the *Class View* (class level interactions), the *Variable Watch View* (field variable accesses) and the *Method Pixel View* (shows method calling relationships). These views are described in detail in Chapter 5 and are then evaluated in Chapter 8. These representations were also implemented into a proof of concept prototype tool, which allowed the representations to be investigated on a variety of real world Java programs.

**c) Provide various levels of abstractions in visualisations of runtime information.**

DJVis provides numerous levels of abstraction on a running program. This is achieved through two approaches. Firstly, each view presents a different level of abstraction of the software. For example, the *Class View* represents a class level abstraction, whilst the *Variable Watch View* shows details at the level of an individual field and its accesses. The views are synchronised where applicable to allow the user to coherently change between abstraction levels with the minimum of cognitive overhead. Secondly, each view typically incorporates its own abstraction levels. An example of this is the *Runtime View* which presents both an overview of the threading and thread group structures, whilst allowing the user to zoom in to the level of individual method calls to observe details of the method's structure and its variables. These two approaches provide the user with a variety of abstractions of the program under study.

**d) Develop a proof of concept prototype tool to demonstrate visualisations.**

The visualisations proposed as part of DJVis were implemented as discussed in Chapter 6. This prototype tool allowed the ideas to be applied to real world Java software. The tool uses debugging techniques to extract the necessary information, allowing programs to be visualised without modification. This enabled the feasibility of the representations to be evaluated. The proof of concept tool implemented the main ideas presented within this thesis, however, not all the aspects were

implemented. For instance, the *Variable Watch View* was not implemented and neither were the lowest levels of detail for the *Runtime View*. However, the feasibility of the views was considered in terms of whether the required information could be extracted using the implementation technique chosen. The views could have been generated automatically and the visualisations in this case were hand produced. The implementation of the prototype tool is discussed in the evaluation chapter in section 8.2.

**e) Show the applicability of these concepts to maintenance tasks using this proof of concept tool.**

The ideas presented in this thesis were demonstrated against maintenance tasks in the evaluation in Chapter 8. The scenarios highlighted how DJVis could be used on a variety of maintenance activities, whilst the case study provided a more detailed example. These include corrective and preventative maintenance activities, along with generic program comprehension tasks. For instance, scenario 3 (section 8.4.3) showed how DJVis could be used for preventative maintenance. While the case study (section 8.5) gave a detailed example of how DJVis can be used to gain an understanding of the structure of a piece of software and then aid in the planning of how to extend this software's functionality. The case study showed the application of DJVis, and more specifically, the prototype tool to a real piece of Java software. The ideas were also evaluated against two feature-based frameworks, including one designed for the assessment of software exploration tools for program comprehension tasks. This gave encouraging results, with DJVis doing well against the frameworks. In particular the frameworks highlighted the support for multiple levels of abstraction and the ability for the user to control the navigation and swap between arbitrary and directional navigation. However, the frameworks also highlighted areas of DJVis that need further work. For instance, the framework by Knight [Knig00] highlighted the lack of support for visualising the data architecture of the program. At present the user can investigate the classes and local variables involved, but there is currently no support for visualising the objects that make up such data structures. This area of visualising object level details is described in more detail in section 9.4 on further work.

**f) Demonstrate that the visualisations can be generated automatically with the programmer needing no knowledge of the structure of the software under study.**

The visualisations presented in this thesis can all be generated automatically. The extraction of runtime information for the proof of concept tool is based on the Java Platform Debugger Architecture, which allows details to be extracted with no modifications needed to the class file or Java source code. The only condition is that the code is compiled to include debugging information, although the visualisation is still able to extract some information without this. The use and simplicity of using the proof of concept prototype tool is presented in section 6.5. The user does not need to know anything about the structure of the program under study, instead all they must do is select the Java file (source, class or jar file) to start from within the proof of concept visualisation and the rest of the process is automatic.

## 9.4 Future Work

The visualisation of runtime information offers great scope for further work. This is both in terms of expanding the ideas presented here, as well as generic areas of runtime visualisation that offer scope for future development. This separation is used to highlight the areas for future research.

### DJVis

- The current DJVis ideas presented in the thesis could be expanded to allow greater benefit to prospective users. The aim of the current visualisations is to focus on class level interactions. However, there exists many other levels of abstraction that could be visualised in order to aid program comprehension. One main area for research could be the addition of object level visualisations and the interconnection of these with the existing visualisations. This would allow the user to investigate the program at a lower level of detail than currently possible and allow them to explore the object-level interactions. This addition of new visualisations links in with the ability to add greater abstraction levels as the current ideas only explore certain types of runtime information. New visualisations could be based on existing information, or on new measures such as combining the current information with profiling information and details extracted from a static parse of the source code. There are many possibilities for the incorporation of additional information from varied sources such as revision control information and these would add new intricacies and possibilities to the DJVis visualisation.
- The informal evaluation identified that the Query View was diminished due to the use of mixed 2D and 3D representations. Therefore, there may be benefits if this querying idea was made more central through uniform support by all views. This would involve the convergence of the views on either 2D or 3D representations.
- DJVis currently focuses on the visualisation of Java, however many of the ideas could be applied to other object-oriented languages with little or no changes. Some ideas may need some modification and the information extraction methods would vary between languages. However, benefits may be gained by offering coherent visualisations across different languages allowing users to move between object-oriented languages without having to learn the use of a new visualisation. Issues would need to be addressed when the language implements the OO paradigm in differing way to Java. For example, the visualisation of C++ would introduce new challenges due to the ability to place code outside of classes. This code would therefore need to be visualised differently in the Class View, as it would not have a classes node to be positioned around.
- The discussion of runtime issues in section 8.3 highlighted the lack of support for saving the state of the visualisation to allow comparison to future states. The incorporation of this may benefit some tasks, such as if the user wants to consider the differences caused by running a program in a different set-up.

The issues mentioned so far have been specifically focusing on extensions to the DJVis ideas. However, some of the future work from this research focuses on issues that apply to the whole area of runtime visualisation.

### **Software Visualisation of Runtime Data**

- The use of runtime information offers a breadth of visualisation possibilities. Numerous diverse tasks can be supported through suitable presentation of this information, and therefore, approaches that offer access to this breadth of information are beneficial. This information may be extracted through the use of different methods, but can be presented in a coherent fashion. This is not to say that one large visualisation environment is necessarily the way forward, instead a suite of co-operating tools may be more suitable. There needs to be support for these different levels of runtime information to support tasks such as performance analysis, where a user may want to start looking at higher level performance data before focusing down to the actual objects involved in some operation using multiple levels of abstraction. Currently there is little support for such tasks, unless numerous tools are used. A suite of interconnected tools could allow the user to access information on a piece of software in a coherent fashion, despite having multiple information sources, such as dynamic and static analysis and various metrics from both static and dynamic analysis. This diverse information would be presented in a manner to allow the user to drive the investigation based on their task. The visualisation of such an enormous information source would present a major challenge and great thought would have to go into the representations, so that they could support the changes in abstraction and integration of the different data types.
- The adoption of visualisation techniques could benefit from the integration of visualisation approaches into current development processes. This would reduce the separation between the development process and visualisation activities and may prompt developers to use visualisation in their development process.
- One major challenge for runtime visualisation is the extraction of the necessary information in an efficient and non-obtrusive way. Not only must the visualisation be useful, but it must also be feasible to extract the necessary information from large-scale software systems. This is a not a visualisation issue per se, but one which must be addressed, if runtime visualisation systems are to be considered viable.

This is not a definitive list, but emphasises some of the major areas of future work that could be considered as an extension to this research.

## **9.5 Conclusion**

This thesis has highlighted the area of runtime visualisation and presented its research issues and challenges whilst demonstrating its benefits, particularly for object-oriented software. This thesis has highlighted the current state of research and proposed new representations of object-oriented software through DJVis and its views (Runtime View, Query View, Class View, Method Pixel View and Variable

Watch View). It has also defined issues that should be considered in the design and evaluation of runtime visualisations. This discussion of issues and the proposed new representations aim to guide future research.

## Abbreviations

2D	2 Dimensional
3D	3 Dimensional
GUI	Graphical User Interface
JDI	Java Debug Interface
JDK	Java Development Kit
JDWP	Java Debug Wire Protocol
JPDA	Java Platform Debugger Architecture
JVM	Java Virtual Machine
JVMCI	Java Virtual Machine Debug Interface
OO	Object-Oriented
VM	Virtual Machine

# References

- [Baec97] **R. Baecker, C. DiGiano and A. Marcus**, *Software Visualization for Debugging*, Communications of the ACM, Vol. 40, No. 4, pages 44-54, April 1997.
- [Benn91] **K. Bennett, B. Cornelius, M. Munro and D. Robson**, *Software maintenance*, Chapter in Software Engineer's Reference Book, J. A. McDermid (Ed) Butterworth-Heinemann Ltd, 1991.
- [Broo83] **R. Brooks**, *Towards a Theory of the Comprehension of Computer Programs*, International Journal of Man-Machine Studies, Vol. 18, No. 6, pages 543-554, 1983.
- [Broo87] **F. P. Brooks Jr.**, *No Silver Bullet: Essence and Accidents of Software Engineering*, IEEE Computer, No. 20, pages 10-19, April 1987.
- [Brow85] **M. H Brown and R. Sedgewick**, *Techniques for algorithm animation*, IEEE Software, Vol. 2, No. 1, pages 28-39, 1985.
- [Burk98] **S. K. Burkwald, S. G. Eick, K. D. Rivard and J. D. Pyrcce**, *Visualizing Year 2000 Program Changes*, Proceedings of the 6<sup>th</sup> International Workshop on Program Comprehension (IWPC '98), Ischia, Italy, pages 13-18, June 1998.
- [Chi98] **E. H. Chi, J. Pitkow, J. Mackinlay, P. Pirolli, R. Gossweiler and S. K. Card**, *Visualizing the Evolution of Web Ecologies*, Proceedings of ACM Conference on Human Factors in Computing Systems (CHI '98), ACM Press, Los Angeles, California, pages 400-407, 1998.
- [DDD] GNU Data Display Debugger (DDD).  
<http://www.gnu.org/software/ddd/>  
(Accessed 16/10/02)
- [DePa97] **W. De Pauw, D. Kimelman and J. Vlissides**, *Visualizing Object-Oriented Software Execution*, In Software Visualization, J. T. Stasko, J. B. Domingue, M. H. Brown and B. A. Price (eds.), MIT Press, 1997.
- [dot] dot  
<http://www.research.att.com/sw/tools/graphviz/refs.html>  
(Accessed 19/11/02)
- [Dwyer01] **T. Dwyer**, *Three Dimensional UML Using Force Directed Layout*, Proceedings of the Australian Symposium on Information Visualisation, Sydney, Australia, December 2001.



- [Eick92] **S. G. Eick, J. L. Steffen and E. E. Sumner Jr**, *SeeSoft<sup>TM</sup> --a tool for visualizing line oriented software statistics*, IEEE Transactions on Software Engineering, Vol. 18 No. 11, pages 957-968, November 1992.
- [Feij98] **L. Feijs and R. de Jong**, *3D Visualization of Software Architectures*, Communications of the ACM, Vol. 41, No. 12, pages 72-78, December 1998.
- [Frontier] Frontier Developments  
<http://www.frontier.co.uk/>  
(Accessed 29/11/02)
- [Gamm94] **E. Gamma, R. Helm, R. Johnson and J. Vlissides**, *Design patterns: elements of reusable object-oriented software*, Addison-Wesley, 1994.
- [gdb] GDB: The GNU Project Debugger  
<http://www.gnu.org/software/gdb/gdb.html>  
(Accessed 18/11/02)
- [Geis94] **A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek and V. Sunderam**, *PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing*, MIT Press, Scientific and Engineering Computation, Massachusetts Institute of Technology, 1994.  
Available online at: <http://www.netlib.org/pvm3/book/pvm-book.html> (Accessed 27/11/02)
- [Glob94] **A. Globus and E. Raible**, *Fourteen Ways to Say Nothing With Scientific Visualization*, IEEE Computer, Vol. 27, No. 7, pages 86-88, July 1994.
- [Gogo99] **M. Gogolla, O. Radfelder and M. Richters**, *Towards Three-Dimensional Representation and Animation of UML Diagrams*, Proceedings of UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, Vol. 1723, pages 489-502, October 1999.
- [Grae81] **A. C. Graesser**, *Prose Comprehension Beyond the Word*, New York: Springer-Verlag, 1981.
- [Hans97] **D. R. Hanson and J. L. Korn**, *A Simple and Extensible Graphical Debugger*, Proceedings of the Winter USENIX Technical Conference, Anaheim, CA, pages 173-184, January 1997.

- [Hatch01] **A. S. Hatch, M. P. Smith, C. M. B. Taylor and M. Munro**, *No Silver Bullet for Software Visualisation Evaluation*, Proceedings of the Workshop on Fundamental Issues of Visualization, Proceedings of The International Conference on Imaging Science, Systems and Technology (CISST), Las Vegas, USA, pages 651-657, June 2001.
- [Hubb99] **R. J. Hubbard, J. Cook, M. Keates, S. Gibson, T. Howard, A. Murta, A. West and S. Pettifer**, *GNU/MAVERIK: A mirco-kernel for large-scale virtual environments*, Proceedings of ACM Symposium on Virtual Reality Software and Technology, London, December 1999.
- [IEEE] IEEE Std 610.12-1990, *IEEE Standard Glossary of Software Engineering Terminology*, 1990. In IEEE Standards Software Engineering, 1999 Edition Volume One: Customer and Terminology Standards, page 46, 1999.
- [Imag] Imagix 4D  
<http://www.imagix.com/>  
(Accessed 18/11/02)
- [IntelliJ] IntelliJ IDEA  
<http://www.intellij.com/>  
(Accessed 19/11/02)
- [ISVis] ISVis  
<http://www.cc.gatech.edu/morale/tools/isvis/isvis.html>  
(Accessed 27/11/02)
- [JBuilder] Borland JBuilder  
<http://www.borland.com/jbuilder>  
(Accessed 19/11/02)
- [Jeff99] **C. L. Jeffery**, *Program Monitoring and Visualization: An Exploratory Approach*, Springer-Verlag New York, Inc., 1999.
- [Jerd94] **D. F. Jerding and J. T. Stasko**, *Using Visualization to Foster Object-Oriented Program Understanding*, Graphics, Visualization and Usability Center, Georgia Institute of Technology, Technical report GIT-GVU-94-33, 1994.
- [Jerd96a] **D. F. Jerding and J. T. Stasko**, *The Information Mural: Increasing Information Bandwidth in Visualizations*, Graphics, Visualization and Usability Center, Georgia Institute of Technology, Technical report GIT-GVU-96-25, 1996.

- [Jerd96b] **D. F. Jerding and J. T. Stasko**, *Visualizing Message Patterns in Object-Oriented Program Executions*, Graphics, Visualization and Usability Center, Georgia Institute of Technology, Technical report GIT-GVU-96-15, 1996.
- [Jins] Jinsight  
<http://www-106.ibm.com/developerworks/library/jinsight/>  
(Accessed 19/11/02)
- [JPDA] Java™ Platform Debugger Architecture  
<http://java.sun.com/products/jpda/>  
(Accessed 19/11/02)
- [JTid] JTidy  
<http://sourceforge.net/projects/jtidy>  
(Accessed 19/11/02)
- [Kitc96] **B. Kitchenham and L. Jones**, *Evaluating Software Engineering Methods and Tools. Part 1: The Evaluation Context and Evaluation Methods*, Software Engineering Notes, Vol. 21, No. 1, pages 12-15, January 1996.
- [Knig00] **C. Knight**, *Virtual Software in Reality*, PhD Thesis, Department of Computer Science, University of Durham, June 2000.
- [Knig01] **C. Knight**, *Visualisation Effectiveness*, Proceedings of the Workshop on Fundamental Issues of Visualization, Proceedings of The International Conference on Imaging Science, Systems and Technology (CISST), Las Vegas, USA, pages 639-643, June 2001.
- [Knig99a] **C. Knight and M. Munro**, *Visualising Software – A Key Research Area*, Proceedings of the IEEE International Conference on Software Maintenance, Oxford, England, August – September 1999.
- [Knig99b] **C. Knight and M. Munro**, *Comprehension with[in] Virtual Environment Visualisations*, Proceedings of the IEEE 7th International Workshop on Program Comprehension, pages 4-11, May 1999.
- [Korn98] **J. Korn and A. W. Appel**, *Traversal-based Visualization of Data Structures*, Proceedings IEEE Symposium on Information Visualization, pages 11-18, October 1998.
- [Laff94] **C. Laffra and A. Malhotra**, *HotWire -- A Visual Debugger for C++*, Proceedings of USENIX C++ Technical Conference, USENIX Association, pages 109-122, 1994.

- [Laht98] **S-P. Lahtinen, E. Sutinen and J. Tarhio**, *Automated Animation of Algorithms with Eliot*, Journal of Visual Languages & Computing, Vol. 9, No. 3, pages 337-349, June 1998.
- [Lamp95] **J. Lamping, R. Rao and P. Pirolli**, *A focus+context technique based on hyperbolic geometry for visualizing large hierarchies*, Proceedings on Human factors in computing systems (CHI '95), ACM Press, Denver, Colorado, USA, pages 401-408, 1995.
- [Lang95] **D. B. Lange and Y. Nakamura**, *Program Explorer: A Program Visualizer for C++*, Proceedings of USENIX Conference on Object-Oriented Technologies (COOTS), Monterey, California, pages 39-54, June 1995.
- [Lanz01] **M. Lanza and S. Ducasse**, *A Categorization of Classes based on the Visualization of their Internal Structure: the Class Blueprint*, Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2001), ACM Press, pages 300-311, 2001.
- [Leto86] **S. Letovsky**, *Cognitive Processes in Program Comprehension*, Empirical Studies of Programmers, Albex, Norwood NJ, pages 58-79, 1986.
- [Lewe02] **C. Lewerentz and F. Simon**, *Metrics-based 3D Visualization of Large Object-Oriented Programs*, Proceedings of the IEEE 1st International Workshop on Visualizing Software for Understanding and Analysis, Paris, pages 70-77, June 2002.
- [Lieb95] **H. Lieberman and C. Fry**, *Bridging the Gulf between Code and Behavior in Programming*, Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI '95), ACM Press, pages 480-486, 1995.
- [Lieb97] **H. Lieberman and C. Fry**, *ZStep 95, A Reversible, Animated Source Code Stepper*, In Software Visualization: Programming as a Multimedia Experience, John Stasko, John Domingue, Blaine Price, Marc Brown, Eds., MIT Press, 1997.
- [Litt86] **D.C. Littman, J. Pinto, S. Letovsky and E. Soloway**, *Mental Models and Software Maintenance*, Empirical Studies of Programmers, Albex, Norwood NJ, 1986.
- [Look] Look!  
<http://www.windriver.com/products/look/>  
(Accessed 19/11/02)
- [Male01] **J. I. Maletic, J. Leigh, A. Marcus and G. Dunlap**, *Visualizing Object-Oriented Software in Virtual Reality*, Proceedings of the International Workshop on Program Comprehension (IWPC'01), Toronto, Canada, pages 26-35, May 2001.

- [Mayr93] **A. Von Mayrhauser and A. M. Vans**, *From code understanding needs to reverse engineering tools capabilities*, Proceedings of the International Workshop on Computer-Aided Software Engineering (CASE93), Singapore, pages 230-239, July 1993.
- [Mayr95] **A. Von Mayrhauser and A. M. Vans**, *Program Comprehension During Software Maintenance and Evolution*, IEEE Computer, Vol. 28, No. 8, pages 44-55, August 1995.
- [Mehn01] **K. Mehner and B. Weymann**, *Visualization and Debugging of Concurrent Java Programs with UML*, In W. de Pauw, S. Reiss, J. Stasko (eds.), Proceedings of the Workshop on Software Visualization, International Conference on Software Engineering, Toronto, Canada, May 2001.
- [Mehn02] **K. Mehner**, *JaVis: A UML-Based Visualization and Debugging Environment for Concurrent Java Programs*, In S. Diehl (Ed.): Software Visualization International Seminar, Dagstuhl Castle, Germany, May, 2001. Revised Papers. LNCS 2269, Springer-Verlag, p. 163-175, 2002.
- [Myer90] **B. A. Myers**, *Taxonomies of Visual Programming and Program Visualization*, Journal of Visual Languages and Computing, Vol. 1 No. 1, pages 97-123, 1990.
- [NetBeans] NetBeans  
<http://www.netbeans.org/>  
(Accessed 19/11/02)
- [Oech02] **R. Oechsle and T. Schmitt**, *JAVAVIS: Automatic Program Visualization with Object and Sequence Diagrams Using the Java Debug Interface (JDI)*, In S. Diehl (Ed.): Software Visualization, International Seminar, Dagstuhl Castle, Germany, May 20-25, 2001. Revised Papers, LNCS 2269, Springer-Verlag, pages 176-190, 2002.
- [OED] Oxford English Dictionary  
Oxford University Press  
<http://www.oed.com/>  
(Accessed 10/03/02)
- [Ouds96] **M. J. Oudshoorn, H. W. Widjaja and S. K. Ellershaw**, *Aspects and Taxonomy of Program Visualisation*, In P. Eades and K. Zhang (editors), Software Visualisation, Chapter 1. World Scientific Press, Singapore, 1996.
- [Park98] **G. Parker, G. Franck and C. Ware**, *Visualisation of Large Nested Graphs in 3D: Navigation and Interaction*, Journal of Visual Languages and Computing, Vol. 9, pages 299-317, 1998.

- [Penn87a] **N. Pennington**, *Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs*, *Cognitive Psychology*, Vol. 19, pages 295-341, 1987.
- [Penn87b] **N. Pennington**, *Comprehension Strategies in Programming*, Proceedings of the Second Workshop on Empirical Studies of Programmers, Albex, Norwood NJ, pages 100-112, 1987.
- [Pott93] **C. Potts**, *Software Engineering Research Revisited*, *IEEE Software*, Vol. 10, No. 5, pages 19-28, September 1993.
- [Pric93] **B. A. Price, R. M. Baecker and I. S. Small**, *A Principled Taxonomy of Software Visualisation*, *Journal of Visual Languages and Computing*, Vol. 4, No. 3, pages 211-266, 1993.
- [Quig02] **A. Quigley**, *Experience with FADE for the Visualization and Abstraction of Software Views*, Proceedings of the 10th International Workshop on Program Comprehension, Paris, pages 11-20, June 2002.
- [Rich99] **T. Richner and S. Ducasse**, *Recovering High - Level Views of Object - Oriented Applications from Static and Dynamic Information*, Proceedings of the International Conference on Software Maintenance (ICSM '99), IEEE Computer Society Press, pages 13-22, 1999.
- [Robe91] **G. G. Robertson, J.D. Mackinlay and S. K. Card** *Cone Trees: Animated 3D Visualizations of Hierarchical Information*, Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI '91), pages 189-194, 1991.
- [Roma92] **G. -C. Roman, K. C. Cox, C. D. Wilcox and J. Y. Plun**, *Pavane: a System for declarative Visualization of Concurrent Computations*, *Journal of Visual Languages and Computing*, Vol. 3, No. 2, pages 161-193, 1992.
- [Roma93] **G. -C Roman and K.C. Cox**, *A Taxonomy of Program Visualization Systems*, *IEEE Computer*, Vol. 26, No. 12, pages 11-24, December 1993.
- [Shil92] **J. J. Shilling and J. T. Stasko**, *Using Animation to Design, Document and Trace Object-Oriented Systems*, Graphics, Visualization and Usability Center, Georgia Institute of Technology, Technical Report GIT-GVU-92-12, 1992.
- [Shne79] **B. Shneiderman and R. Mayer**, *Syntactic Semantic Interactions in Programmer Behavior: A Model and Experimental Results*. *International Journal of Computers and Information Sciences*, Vol. 8, No. 3, pages 219-238, June 1979.

- [Shne80] **B. Shneiderman**, *Software Psychology*, Cambridge, USA, Winthrop Publishers Inc., 1980.
- [Shne96] **B. Shneiderman**, *The Eyes Have It: A Task by Data Type Taxonomy for Information Visualization*, Proceedings of IEEE Conference on Visual Languages, Boulder, Colorado, USA, pages 336-343, September 1996.
- [Smit02] **M. Smith and M. Munro**, *Runtime Visualisation of Object Oriented Software*, Proceedings of the IEEE 1<sup>st</sup> International Workshop on Visualizing Software for Understanding and Analysis, Paris, pages 81-89, June 2002.
- [Solo84] **E. Soloway and K. Ehrlich**, *Empirical Studies of Programming Knowledge*, IEEE Transactions on Software Engineering, Vol. 10, No. 5, pages 595-609, September 1984.
- [Solo88] **E. Soloway, B. Adelson and K. Ehrlich**, *Knowledge and Processes in the Comprehension of Computer Program*, The Nature of Expertise, Chi, M., Glaser, R. and Farr, M. eds., A. Lawrence Erlbaum Associates Hillsdale, N.J., pages 129-152, 1988.
- [Stan84] **T. A. Standish**, *An essay on software reuse*, IEEE Transactions on Software Engineering, Vol. 10, No. 5, pages 494-497, 1984.
- [Stas90] **J. T. Stasko**, *Tango: A framework and system for algorithm animation*, IEEE Computer Vol. 23 No. 9, pages 27-39, 1990.
- [Stas92b] **J. T. Stasko and J. F. Wehrli**, *Three-Dimensional Computation Visualization*, Graphics, Visualization and Usability Center, Georgia Institute of Technology, Technical Report GIT-GVU-92-20, 1992.
- [Stor97a] **M-A. D. Storey, F. D. Fracchia and H. A. Müller**, *Cognitive Design Elements to Support the Construction of a Mental Model during Software Visualization*, Proceedings of the 5<sup>th</sup> International Workshop on Program Comprehension, Dearborn, Michigan, U.S.A., pages 17-28, May 1997.
- [Stor97b] **M-A. D. Storey, K. Wong, F. D. Fracchia and H. A. Müller**, *On Integrating Visualization Techniques for Effective Software Exploration*, Proceedings of IEEE Symposium on Information Visualization (InfoVis'97), Phoenix, Arizona, U.S.A. pages 38-45, October 1997.
- [Stor97c] **M-A. D. Storey, K. Wong and H. A. Müller**, *How Do Program Understanding Tools Affect How Programmers Understand Programs?*, Proceedings of Working Conference on Reverse Engineering (WCRE'97), Amsterdam, Holland, pages 12-21, October 1997.

- [Stor98] **M-A. D. Storey**, *A Cognitive Framework For Describing and Evaluating Software Exploration Tools*, PhD Thesis, School of Computing Science, Simon Fraser University, December 1998.
- [Syst00] **T. Systä**, *Static and Dynamic Reverse Engineering Techniques for Java Software Systems*, PhD Thesis, Department of Computer and Information Sciences, University of Tampere, Report A-2000-4, 2000.
- [Tayl02] **C. Taylor and M. Munro**, *Revision Towers*, Proceedings of the IEEE 1st International Workshop on Visualizing Software for Understanding and Analysis, Paris, pages 43-50, June 2002.
- [Vans99] **A. M. Vans, A. Von Mayrhauser and G. Somlo**, *Program understanding behaviour during corrective maintenance of large-scale software*, International Journal of Human Computer Studies, Vol. 51, No. 1, pages 31-70, 1998.
- [Vion94] **J-Y. Vion-Dury and M. Santana**, *Virtual Images: Interactive Visualization of Distributed Object-Oriented Systems*, Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA '94), Vol. 29, No. 10, pages. 65-84, October 1994.
- [Visi] VisiVue  
<http://www.visicomp.com/product/visivue.html>  
(Accessed 19/11/02)
- [Walk95] **G. Walker**, *Challenges in information visualisation*, British Telecommunications Engineering Journal, Vol. 14, pages 17-25, April 1995.
- [Walk98] **R. J. Walker, G. C. Murphy, B. Freeman-Benson, D. Wright, D. Swanson and J. Isaak**, *Visualizing Dynamic Software System Information through High-level Models*, Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications, (OOPSLA '98), October 1998.
- [Ware96] **C. Ware and G. Franck**, *Evaluating Stereo and Motion Cues for Visualizing Information Nets in Three Dimensions*, ACM Transactions on Graphics, Vol. 15, No. 2, pages 121-140, 1996.
- [Wied86a] **S. Wiedenbeck**, *Processes in computer program comprehension*, In E. Soloway & S. Iyengar, Ed. Empirical Studies of Programmers, Norwood, NJ: Ablex. 1986.
- [Wied86b] **S. Wiedenbeck**, *Beacons in computer program comprehension*, International Journal of Man-Machine Studies, Vol. 25, pages 697-709, 1986.



- [Wied91] **S. Wiedenbeck**, *The initial stage of program comprehension*, International Journal of Man-Machine Studies, Vol. 35, No. 4, pages 517-540, 1991.
- [Wiss99] **U. Wiss and D. Carr**, *An Empirical Study of Task Support in 3D Information Visualizations*, Proceedings of IEEE Conference on Information Visualization, London, England, pages 392-399, July 1999.
- [Youn97] **P. Young and M. Munro**, *A New View of Call Graphs for Visualising Code Structures*, University of Durham, Computer Science Technical Report 03/97, April 1997.
- [Youn98] **P. Young and M. Munro**, *Visualising Software in Virtual Reality*, Proceedings of the IEEE 6<sup>th</sup> International Workshop on Program Comprehension, Ischia, Italy, pages 19-26, June 1998.
- [Youn99] **P. Young**, *Visualising Software in Cyberspace*, PhD Thesis, Department of Computer Science, University of Durham, 1999.

