

Identifying Structural Features of Java Programs by Analysing the Interaction of Classes at Runtime

Michael P. Smith and Malcolm Munro

Visualisation Research Group

Department of Computer Science, University of Durham

Durham, DH1 3LE, UK.

{m.p.smith, malcolm.munro}@durham.ac.uk

Abstract

This paper describes research on visualising Java software at runtime in order to enable the identification of structural features. The aim is to highlight both the static and dynamic structure of the software and aid software engineers in tasks requiring program comprehension of the code. The paper takes the position that this type of analysis and visualisation for object oriented languages must be carried out with dynamic runtime information and that it cannot, in general, be obtained by static analysis alone. A case study is worked through to demonstrate the approach.

1. Introduction

Knight and Munro define software visualisation as "software visualisation is 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." [4]. It is this definition that this paper build on to investigate the structural features of Java programs.

The process of understanding software is fundamental to the majority of, if not all, software engineering tasks. Tasks such as development, testing, debugging, maintenance and performance tuning all require some understanding of the software at the source code level. The code defines the static structure of the software and thus is essential for understanding; however, it can be difficult to get a true understanding from only this static description. This is particularly so for object-oriented software as the paradigm introduces new language features such as polymorphism and dynamic binding that makes analysis and comprehension more difficult. Object-oriented software has many advantages, however Jerding and Stasko suggest it is "a double-edged sword" [1]. This is partly

due to the discrepancies between the static class descriptions and runtime behaviour as networks of communicating objects [2] [3]. 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." [3]. Because of this De Pauw et al. state that "Insight into dynamic aspects is critical for understanding, tuning and debugging object-oriented software" [3]. This discrepancy is the motivation behind this research, which aims to improve program comprehension of object-oriented systems by analysing and visualising both their static and dynamic structure through the use of a number of visualisations.




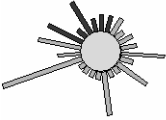



2. DJVis Visualisations

DJVis is a visualisation tool designed to show details of Java software as it executes [5][6][7]. It connects to an executing Java program through the Java Platform Debugger Architecture [8] to extract program events. The tool has a number of different views, each of which shows some aspect of a Java program. The main views are: the Runtime View (shows threading and call stack details); the Query View (supports the Runtime View by allowing user controllable grouping and exploration of information); the Class View (provides class level details of the software); the Method Pixel View (provides details of method calling relationships); and the Variable Watch View (provides a history of read and write access to a variable). This paper will use the features of the Class View.

The Class View in DJVis is designed to show the structure and relationships between classes. The view

uses an augmented graph representation, with the nodes representing types in the software and the edges representing relationships between the types. The nodes are circular and are augmented with additional details about the type. Coming out from the node by default are 'method lines'. Each line represents a method defined by the class and the length and colour of the line represents metrics for that method. Table 1 shows the representations used in the Class View. A full explanation is given in the papers referenced above.

Table 1 Class View Representations

Representation	Meaning
	Class (shading represents metric (number of instances created by default))
	Interface
	Inner Class (inner shading represents metric)
	A class and its methods (length and shading represent user selectable metrics)
	A class and its fields (shading represents user selectable metrics)
	Type yet to be loaded
	Package type belongs to (Colour coded by package)

The Class View uses colour extensively to convey information, however these colours have been mapped onto different grey scales where possible to facilitate printing in black and white.

3. Case Study

The case study 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 19,000 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. It is

made up of 86 classes defined in 66 source files. The authors had no experience of the source code before applying DJVis to it. 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 Class View of DJVis was used to inspect which classes are used by the application and to investigate the complexity of the class relationships. Figure 1 shows the graph as presented in the Class View at the point when GraphTool had been initialised and is showing its user interface. The nodes represent types (for example a class and its methods on row four of Table 1), and the arcs represent references.

Figure 1 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 1 contains five classes. Inspection of the class names (and optionally the method names and source files using a pop up browser) 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 2.

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 GraphTool. Inspection of the classes using the pop-up browser supports this and 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.

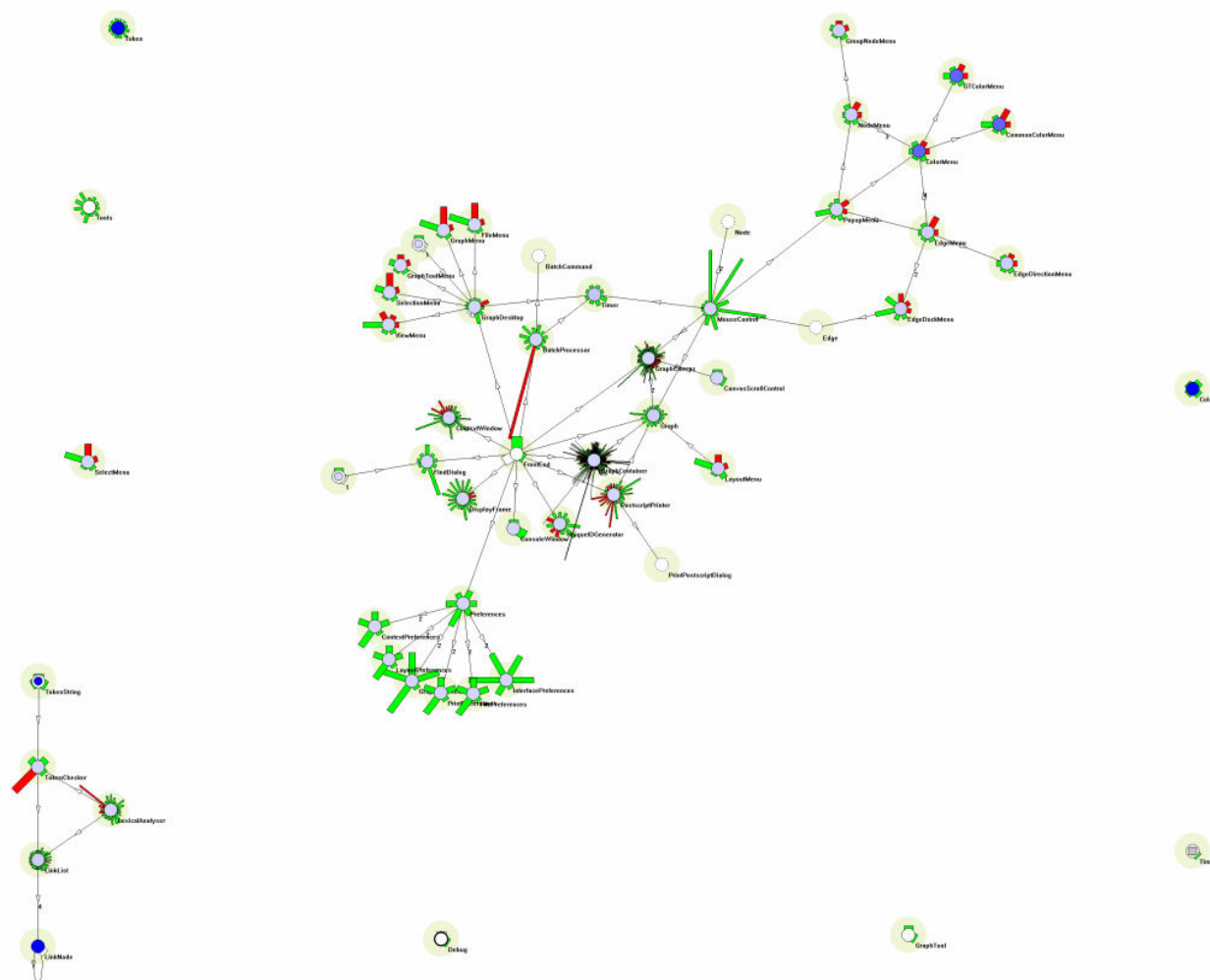


Figure 1 GraphTool classes after initialisation as shown in the Class View

Feature 2

This cluster is centred on the "Preferences" class which 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 then instantiated and these changes are reflected in the Class View

Figure 3 shows the result of opening the Preferences dialog box on the Class View display. The view shows that new classes have been loaded 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

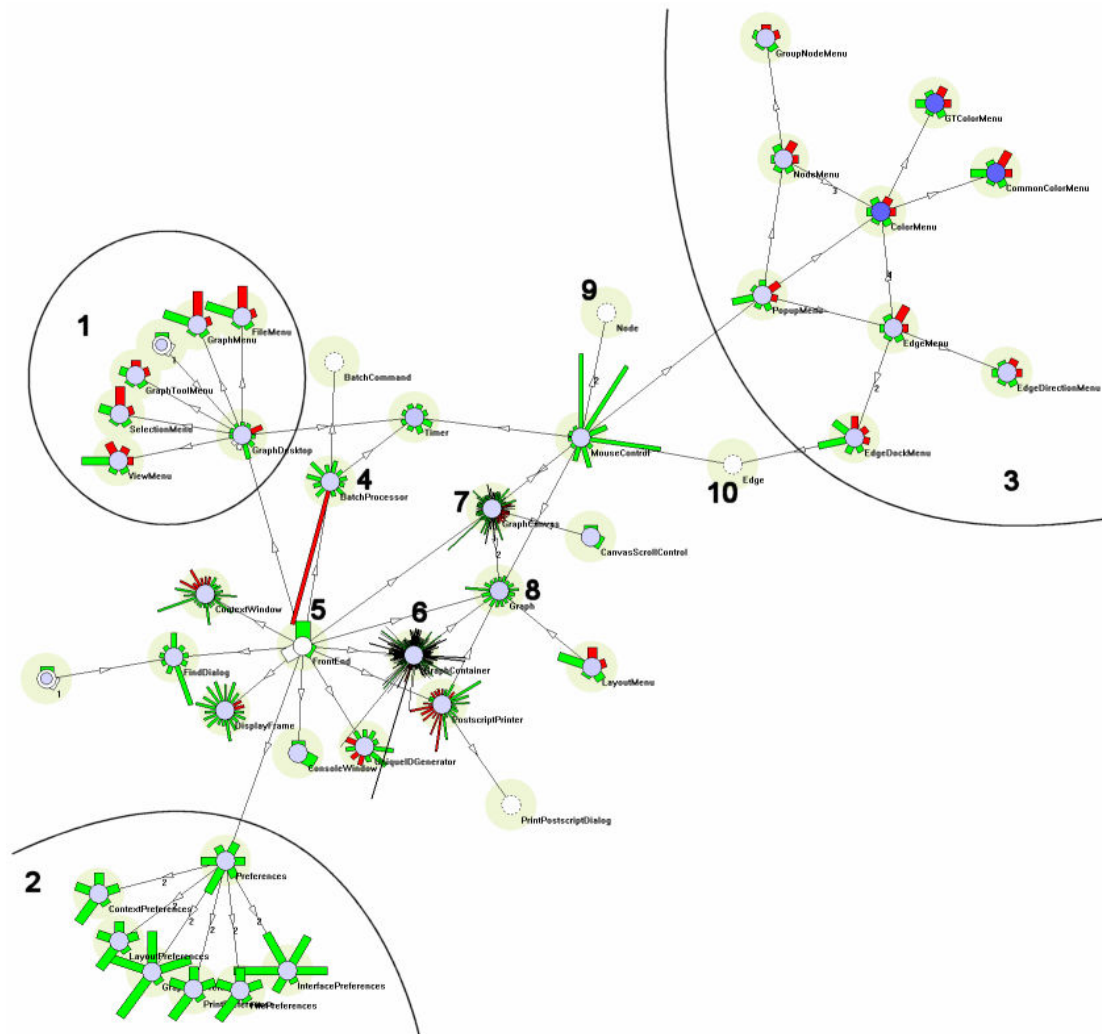


Figure 2 Identification of interesting features in the main sub graph.

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 pop-up browser. This provides detailed information on the class and its methods including the source code and the calling summary.

Feature 5

This 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

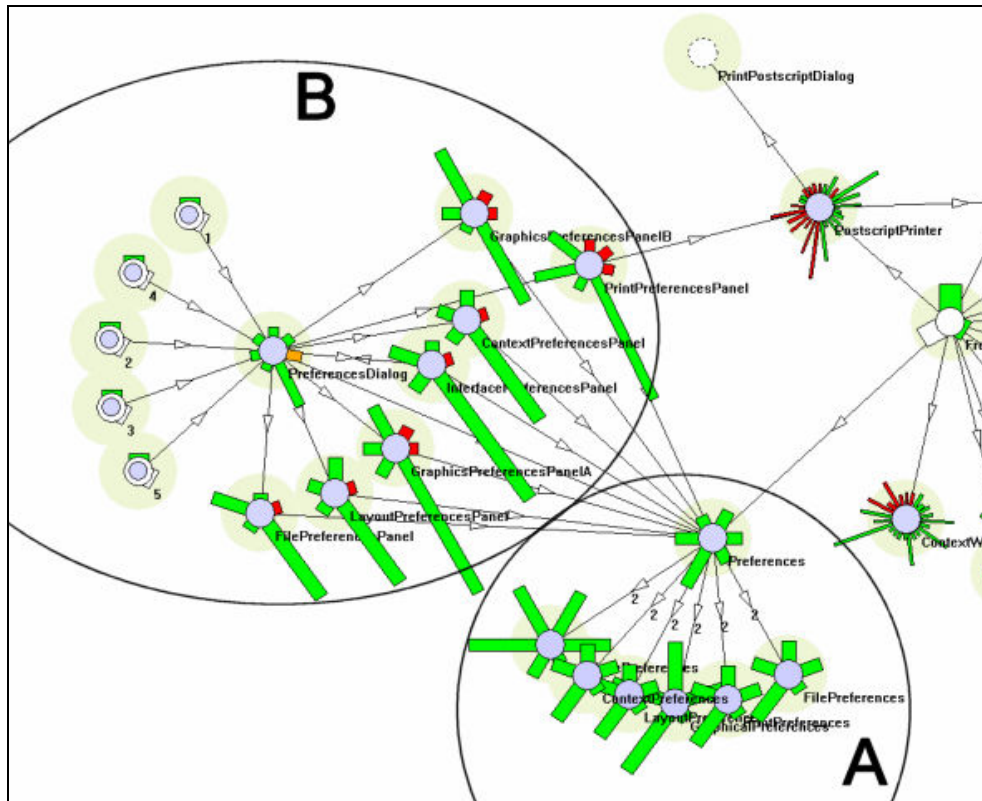


Figure 3 Classes loaded as a result of displaying the Preferences Dialog

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. Changing to the field representation also shows that these references are all public.

Feature 6

This is the class GraphContainer and 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 an important class to comprehend in terms of how the software implements its functionality.

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.

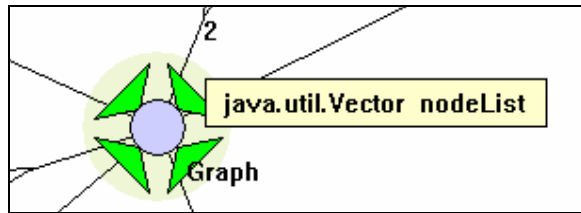


Figure 4 Investigating field names and types using mouse over information

Figure 4 shows that the Graph class has only four fields and that one field of the Graph class is a vector called nodeList. The class also has an 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 4 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.

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. It can also be seen that there is limited encapsulation with many key data structures being public. The functionality of GraphTool is also heavily clustered into a limited number of 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.

From this initial overview the user can identify structural features of interest to their task. They can abstract or filter features of little interest to simplify the visualisation. They can also use the integrated views to investigate the features further. For example, by looking at the field accesses and at the calling relationships and histories, and through specifying custom mappings to highlight metric values of interest.

4. Conclusions

The paper has shown the application of the DJVis visualisation tool in identifying the structure of a Java program. The program is executed in conjunction with the DJVis tool and the Class View visualisation is used. This visualisation shows the interaction between classes at runtime and helps in identifying structural features by easily showing partitions of the class graph. This type of analysis cannot be carried out in general through static analysis.

5. References

- [1] 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.
- [2] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design patterns: elements of reusable object-oriented software, Addison-Wesley, 1994.
- [3] 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.
- [4] 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.
- [5] M. Smith, "Runtime Visualisation of Object-Oriented Software", PhD Thesis, Department of Computer Science, University of Durham, 2003.
- [6] M. Smith and M. Munro, "Runtime Visualisation of Object Oriented Software", Proceedings of the IEEE 1st International Workshop on Visualizing Software for Understanding and Analysis, Paris, June 2002, pp. 81-89.
- [7] M. Smith and M. Munro, "Providing a user customisable tool for software visualisation at runtime", IASTED International Conference on Visualization, Imaging, and Image Processing (VIIP 2004), 2004
- [8] Java™ Platform Debugger Architecture, <http://java.sun.com/products/jpda>

