Runtime Visualisation of Object Oriented Software

Michael P. Smith and Malcolm Munro
Visualisation Research Group
Research Institute in Software Evolution
Department of Computer Science
University of Durham
Durham, DH1 3LE, UK
{M.P.Smith, Malcolm.Munro}@durham.ac.uk

Abstract

Software is inherently dynamic, yet much of the analysis and comprehension processes focus entirely on the static source code of the software. This paper looks at how software visualisation offers a way to aid comprehension by displaying both static and dynamic aspects of a piece of software. A new visualisation is presented with specific focus on a class level summary view.

1. Introduction

The need to understand a piece of software, in terms of its source code and how it works, is key to maintenance and development tasks. Whether it is the need to add new functionality, fix an error, or simply help a new developer join an existing project, gaining an understanding of the software is an essential task. However, despite its commonality it has limited tool support. Documentation on the software can help, but this can be incorrect or out-of-date, especially for maintenance activities. This results in the source code being the main source of information for software engineers, yet simply studying the source code in its entirety is unfeasible, given the size of the real software systems. Even trying to find out the areas of the code of interest is a time-consuming and error prone task. In the case of object-oriented software, a number of other difficulties arise when trying to gain an understanding of the software. This is due to the discrepancies between the static class descriptions and runtime behaviour as networks of communicating objects [1]. The understanding of software systems, and in particular, object-oriented systems, could therefore benefit from tools to aid in highlighting this runtime behaviour, as well as the static aspects of the software. Software Visualisation offers an approach to aid program understanding by presenting the relationships within a piece of software through visual means.

This paper presents a background to software visualisation support for program comprehension tasks and looks at a proposed visualisation for Java programs. This attempts to aid comprehension by presenting runtime information alongside static information.

2. Background

Program comprehension is a major and time-consuming task. Understanding existing programs, accounts for the majority of time which is spent on maintenance, debugging, and code re-use processes [2]. A number of program comprehension theories exist, and a good summary can be found in Mayrhauser and Vans [3]. These suggest different methods for program comprehension, which can be categorised into top-down, bottom-up and opportunistic. The top-down approach is goal driven, and consists of using domain knowledge along with higher level information in order to identify areas of functionality. Bottom-up comprehension occurs when the code is studied and chunked together into higher level abstractions. However, it is a combination of these two approaches that is used, and this is the opportunistic approach. Here, both top-down and bottom-up comprehension processes are switched between, depending on the cues available and the knowledge of the person doing the comprehension in terms of their existing program and domain knowledge.

Software Visualisation offers a way to support these comprehension strategies, and many different visualisations exist. In order to be useful, it is beneficial for the visualisations to support the variations in comprehension by allowing easy changing between top-down and bottom-up strategies and user customisation. The variations arise from differences in cognition, comprehension strategies and knowledge between users, as well as differences in tasks. Storey et al. offer a

Proceedings of the First International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT'02)
0-7695-1662-9/02 $17.00 © 2002 IEEE

framework to guide tool support for program comprehension [2]. Jerding and Stasko [4] have called for the need to visualise object-oriented systems, suggesting that the object-oriented paradigm is a 'double-edged sword'. The code provides the static descriptions of classes and their relationships, however, at runtime there exists a network of communicating objects, each an instance of a certain class. Polymorphism and inheritance can also complicate program understanding.

A number of visualisation tools exist that focus on runtime information and a few of the more notable recent examples are Jinsight [5], Look [6] and Visvue [7].

Jinsight [5] is a tool for visualising the execution of Java programs developed by IBM. The Java program to be analysed is run on a modified version of the Java Virtual Machine that is supplied with Jinsight. This produces a trace file of the execution, and user options allow specification of which events to record. The trace file is then loaded into Jinsight to be analysed. The system offers a number of views, and is designed with object-oriented and multithreaded programs in mind. It shows information on the program, such as calling information, object references and creations. It also has a pattern extraction facility, which extracts patterns related to a given method and shows how the method typically executes. This pattern extraction aims to overcome the massive numbers of invocations of a method, by being able to show the cases when the methods execution diverges from the general pattern. While the different views allow for customisation they only show higher level calling information and no support for linking to actual lower level details such as the source code exists.

Look! [6] is a tool for visualising and debugging C++ programs. It offers a more comprehensive set of views showing both low-level details, such as source code and variable values, along with views of the inheritance structure and referencing between objects using a graph representation. The system also allows classes to be clustered. This allows the message interaction between clusters, and the objects which exist in a particular cluster, to be viewed at a higher level of abstraction.

VisiVue [7] is a runtime visualisation tool for Java. It provides an object reference view using a graph representation and supports user filtering of classes from the view. Trace events can also be output to a file, recording method entry, and exits along with variable changes. However, the system does not allow direct browsing of source code, or show visually the method usage or inheritance structures present in the program. The graph of the object references is also subject to problems of scale when a large number of objects exist, especially if the objects have numerous fields as this affects the size of the node in the graph.

These systems offer a number of types of views. However, they lack support for user annotation and can offer little help when it comes to trying to identify areas for investigation in terms of which areas of code are responsible for specific functionality. They can also suffer from information overload as the size of the program increases and a huge number of objects exist. For example, simply representing the object references using a graph is not adequate, because it fails to scale up when a large number of objects of a particular class exist. Whilst facilities such as zooming and overview windows help, they cannot prevent this problem. More advanced filtering along with higher level representations are needed.

Visualisation of the dynamic aspects of software faces many challenges. Traditional issues, such as how to represent an intangible system, are still faced, along with the layout of these items and how to provide abstraction mechanisms alongside detail views of the huge amount of information that software involves. However, at runtime there can be even more information. Not only is there the static source code, and class structures to highlight, but each of theses classes can have many instances, whose interrelationships may be key to understanding how the software works. Combined with this is the constant change in information, such as object reference relationships, which introduces further challenges for representation and layout.

3. Visualising Object Oriented Software At Runtime

In order to aid the comprehension of Object Oriented software, this approach uses dynamic information alongside the static class descriptions to provide greater information. This aims to overcome the problems caused by polymorphism and inheritance to static analysis. The aim is to allow Java software to be visualised without modification to its source code in an environment where the user can investigate multiple relationships on realistically sized programs.

3.1. Why Dynamic Analysis

The decision to use dynamic analysis of Java programs was driven by a number of factors. The main reason is that static analysis, while beneficial, can only go so far especially in the case of Object Oriented Software. [8]. The Object Oriented paradigm offers a number of problems to static analysis, due to its use of dynamic binding and polymorphism. In order to extract the

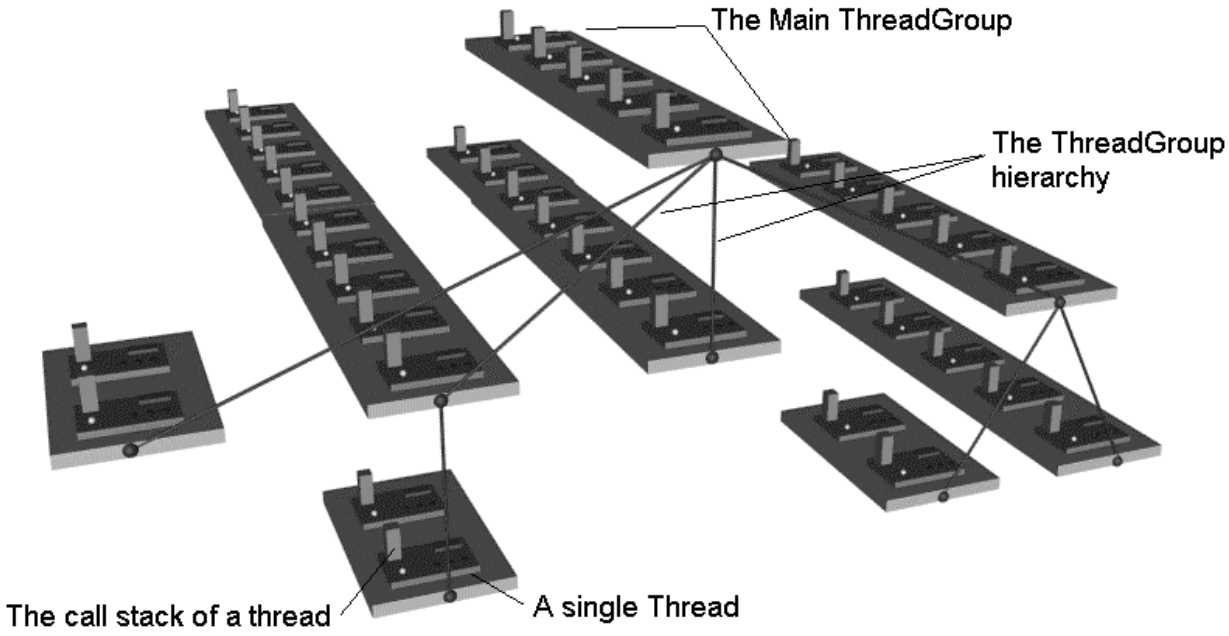

Figure 1 The Runtime View

required information, a dynamic analysis method has been chosen. This also offers the ability to allow user filtering, based on knowledge gained from runtime information. For example, by aiding the user in identifying class usage patterns, thereby, helping them identify classes involved in some specific functionality. This then allows them to filter out classes of no consequence to their task. However, the increased depth of information available from dynamic analysis does not come without a cost. Dynamic analysis, can result in huge information spaces that can be difficult to store and process, never mind visualise! With hundreds of thousands of events to monitor, even for a small program, issues such as user filtering and providing multiple levels of abstraction are particularly important.

3.2. Viewing a running System

A visualisation tool, called DJVis, has been implemented in order to evaluate ideas and allow their application to large-scale software. The runtime information is extracted using the Java Platform Debug Architecture (JPDA) [9]. This allows information to be extracted without the need to modify the source of the program under study. DJVis allows the program to be stepped through using a typical debugger interface (e.g. step to next line, run and break). As the program runs, information is extracted on the classes involved, and the methods being called. This information is presented in a number of views, the main ones are: the Runtime View; the Query View; and the Class View. These views focus on different aspects of runtime information. The Runtime View focuses on thread level events, such as calling information and arguments. Whilst the Class View focuses on class level interactions, such as class reference relationships and class creation details. Combined with this are a number of helper views, which are linked to the main views, and provide another way to navigate the information. All the views use colour extensively, however, for the purposes of this paper, the different features are identified by the differences in the grey scale shading.

3.2.1. The Runtime View

This View presents runtime aspects of a Java program, in terms of the threads, and the thread groups involved, along with the call stack of each thread. The thread groups form a hierarchical relationship, containing other thread groups or threads. A tree layout is used to visualise this relationship.

In Figure 1 the main platforms represent thread groups with the lines showing the hierarchical relationship between them. The smaller platforms upon them represent the threads that exist in each thread group. Upon this platform, details of each thread are presented, such as if the thread is blocked by another thread. The call stack of each thread can be seen, and at this level of detail it simply gives an indication of the number of methods on the stack. However, each call stack can be zoomed in on to see details of the call stack in terms of the individual methods. Figure 1 shows an example with a large number of thread groups (eight in total) and the Main ThreadGroup containing 5 threads. The view can be

customised to show the calling of particular classes. Thus making it easy to see which threads are executing user code and not just Java API code.

3.2.2. The Class View

The Class View presents the classes used in the software, and the relationships between them. The visualisation is based on a graph representation, with the nodes representing the classes, and the edges representing the relationships. The view is customisable, and can present a number of information types dependent on the users requirements. The edges of the graph can be used to show different inter-class relationships, such as references and creates. The use of dynamic analysis allows additional information to be shown, on top of the usual static information. For example, dynamic class references through a common interface can be shown. Thus, making it easier to see the actual class referencing at any one point in the execution. The nodes represent the classes, and their colouring and numbering can present information on the number of objects of that class that have been created (since the program started or from a particular point), to the load order of the class. Around the circular node, the methods of a class are represented as lines coming out from the node. This allows the number of methods each class has to be easily seen. The 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 2 shows the GraphDesktop class, with the method line length representing the method length, and the shading representing the number of calls for that method. The user can easily configure these settings, by using drop down list options for each mapping on the top of the Class View window. As can be seen from Figure 2, the class has 11 methods (4 that haven't been called (shown as white), 4 that have been called a few times (lightly shaded), 2 that have been called slightly more (medium shading) and one method that has been called many times (shown as dark shading)). All of the methods, apart from two, are short in length.

Figure 2 Viewing the GraphDesktop Class in Class View

The Class View provides a good starting point for investigating a program. Not only can the main classes and their relationships be presented, but it is also easy to spot classes with a large amount of functionality, both in terms of a large number of methods or long methods. By observing calling information, it is possible to spot which classes are called within a particular span of the execution. This could be used for highlighting areas for investigation, for example, in order to understand a certain piece of functionality.

The method line idea can also be applied to the fields of a class. In this case, the length or shading would indicate the number of accesses. This can be filtered on whether they are read or write accesses, and on whether they are from the object itself or, from another class if the field is not private. This, combined with the ability to reset the count of field accesses, allows the user to see which fields are used, and in what way for certain functionality. It could also help in performance tuning by showing unexpectedly high field access that could be optimised.

The Class View can also be used to display additional information, for example, the length or shading of the method can be mapped to a measure of its complexity. As well as showing information that could be extracted by the tool itself, it could also be combined with other information sources. Such as using Revision Control Information, to show the number of changes in a method by changing the shading 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.

Graph representations offer an intuitive representation for software engineers, however, they suffer from problems of scale and complexity as the number of nodes and edges increase [10]. In order to aid the scalability of the representation, user filtering is available, and the user can group nodes in order to simplify known or unrelated subsystems. This trade off between familiarity vs. scalability was felt to be acceptable in this case. As in general, the number of classes in a system tends to be small, compared to, say, the number of nodes in a call graph of the system or an object reference graph. The graph also allows interactive scaling of the nodes and the graph itself through slider controls. This allows for greater control and the ability to zoom in and out quickly on the details of the class nodes.

3.2.3. The Query View

The Query View, as its name suggests, is a view dedicated to user queries. It allows items to be dragged from existing views and dropped into a query. These queries can easily be created and named and provide a means for the user to group information that is of interest

to them for a specific task. Here the user controls the placement and inclusion of objects, so for example, they could easily focus on two specific threads or look at certain classes. This helps alleviate the problem of the required information being spread out or in different views. It also provides more flexibility, as information can be accessed in more than one way, and grouped in a way that has meaning to the user. This query view has been implemented with support for a number of views (Runtime and helper views). A full summary of the Query View and the issues involved is beyond the scope of this paper, whose aim is to focus on the visualisation of class level interactions.

3.2.4. Example Scenario

In order to demonstrate some of the concepts described above, an example scenario will be presented which focuses on using the Class View. In this scenario, a piece of software is to be inspected, in order to discover how a piece of functionality is implemented. The software is a tool for displaying and editing graphs, and for this scenario the user has no other knowledge of the software or its structure. The user is interested in finding out how the different layout algorithms are implemented, as they wish to add their own. The first task is, therefore, to gain an overview of the software in terms of key classes and data structures, in order to identify where the functionality of interest is, so its detailed operation can be inspected. As a first step, the program is run from within DJVis using the Class View as the main focus. Using the Class View, the user can observe the order in which the classes are loaded, and how they reference each other. The class naming, number of instances created, and referencing patterns act as higher level cues that can drive further investigation. The method lines allow an impression of which classes have the main functionality.

Figure 3 shows the creation relationships for the classes involved in the graph program. The nodes have small scaling in order to allow an overview to be seen. From this view, a number of things can be noticed. Firstly, a large amount of the functionality is clustered into a small number of classes. For example, the GraphContainer class has a large number of methods, some of which are long in length. (Figure 4 shows a 'zoomed in' view of the class showing its 97 methods more clearly, all the methods are public except for nine, shown as darker shading). Other classes with a significant number of methods are the Node and Edge classes, along with the GraphCanvas class. It can be seen that the

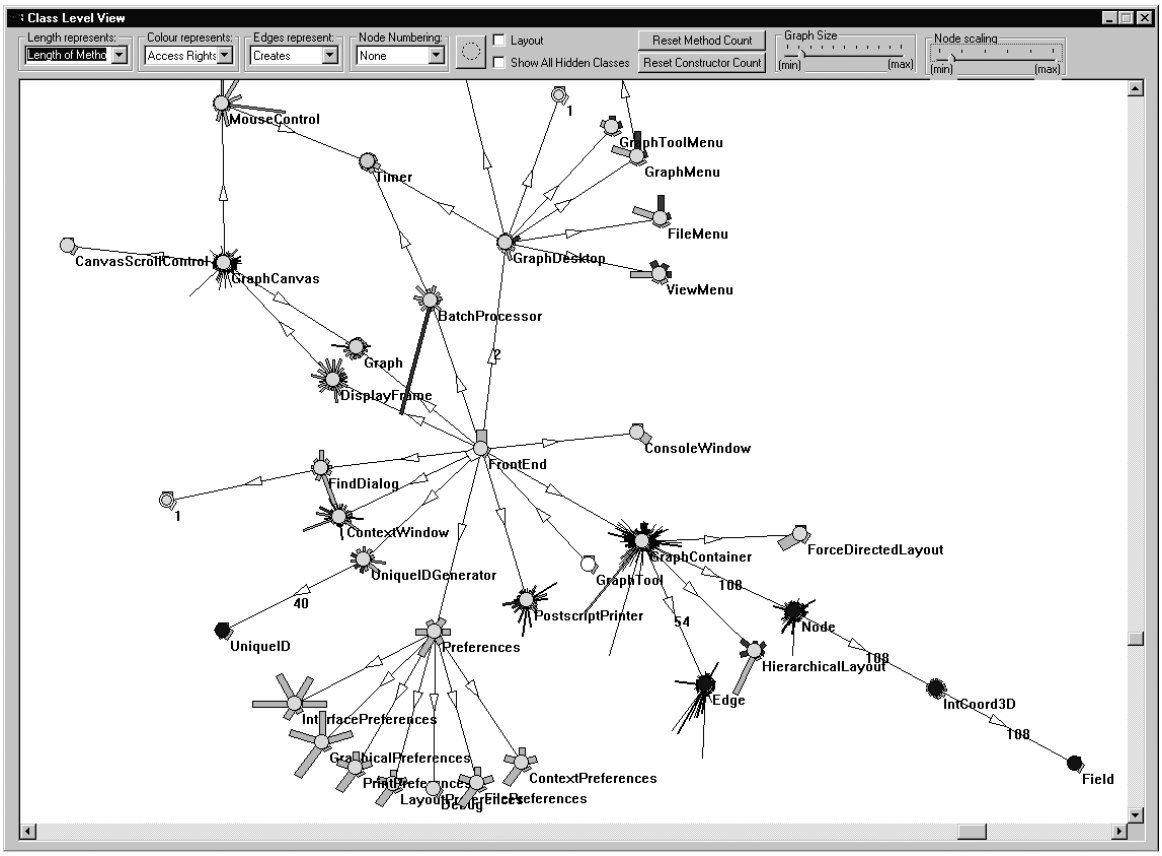

Figure 3 An overview of the class structure using the Class View

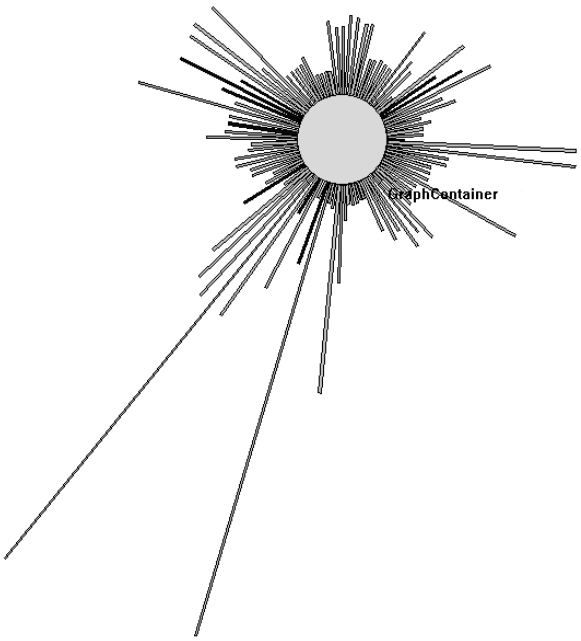

Figure 4 The GraphContainer Class

FrontEnd class is responsible for creating several of the important classes, which it also references (this is observable by changing the edges to show reference relationships using the drop down list at the top of the window). The shading of the nodes represents the number of instances created of that class. As one would expect there has been a large number of Node and Edge objects created (at this point a graph had been loaded). The main managing classes (FrontEnd, GraphContainer, Preferences) have a small number of instances (typically one), shown by the light shading. Static classes are also shown as white nodes. For more detailed information on instances created, the nodes can be numbered with the exact number using the drop down list. The BatchProcessor Class also stands out for having one very long method.

Once an overview of the main classes has been gained, it is possible to focus on the specific area of functionality of interest. From the information presented so far, and possibly any domain knowledge, there may be expectations where, and how, the specific functionality is implemented in the code. However, the visualisation can support this by changing the length of the method lines in the Class View to represent the number of method calls. This can show the total number of method calls, or the number of calls from a user specified point in the execution. Setting such a point, can be easy when the functionality is initiated by a GUI method. However, when it is within a large amount of other functionality, iterative investigation maybe needed to find a suitable point to reset the method call count. Once the functionality has finished, the program can be suspended again and the results of the method calling shown. In the case of this scenario, it can be already be seen from Figure 3 that the GraphContainer class creates a ForceDirectedLayout object. This would therefore be the obvious place to look for information on the layout.

Figure 5 shows the referencing of GraphContainer of the interface Layout. Through this reference, the ForceDirectedLayout class is referenced dynamically (shown by the lighter coloured edge). The interface Layout has one method namely DoLayout(Graph g) and this is therefore the method by which new layouts can be added to the tool. However, it is not always as easy, to know where to focus investigation when trying to understand a piece of functionality. In these cases, the displaying of the method length as the number of calls acts as a guide. It allows classes that aren't involved to be easily spotted, and gives an impression on the localisation of the functionality by showing the number of classes involved.

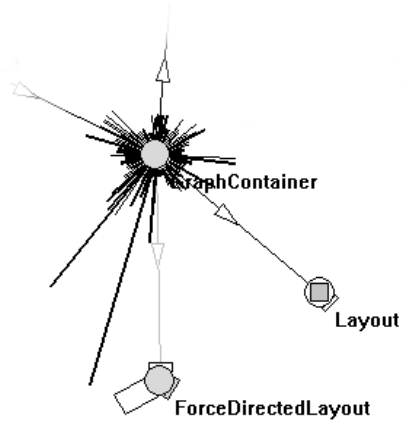

Figure 5 GraphContainer showing reference relationships

Figure 6 clearly shows which, and how many times each method was called. Using this information, areas for further investigation can be identified. From Figure 6, it can be seen that methods of the Node and Edge classes are called in particular, as would be expected along with methods of the ForceDirectedLayout class. Other classes involved include the GraphDesktop and GraphCanvas as well as MouseControl and ContextWindow. It can be hypothesised by their names, that these classes are involved in the graphical update side of the operation. Investigation of the method names, and/or source, may confirm this, allowing them to be removed from the investigation. This leaves the ForceDirectedLayout and Node and Edge classes for further investigation. This could, therefore be a good point to start looking at the source code in the syntax highlighted source view, using

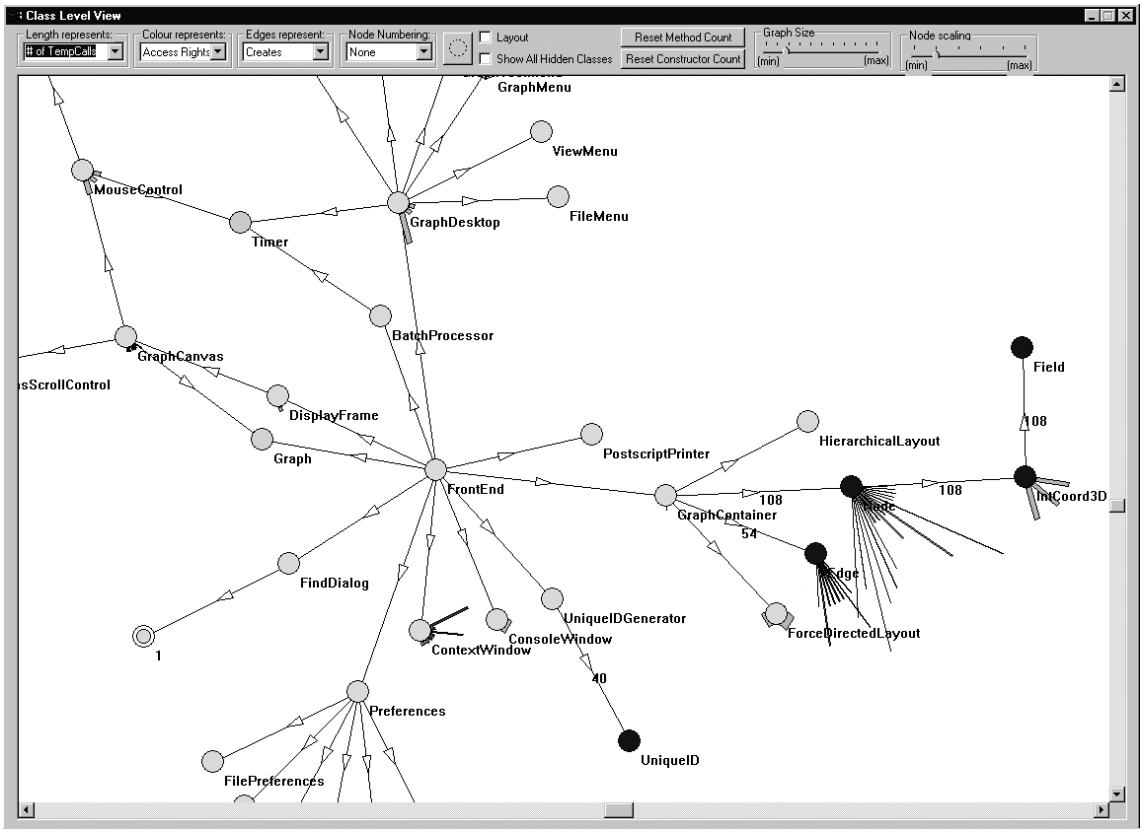

Figure 6 Class View showing method calling counts

the Class View to navigate the source. In this scenario, the class naming and functionality of interest allowed the involved classes to be easily apparent. However, the same technique could be used in more complex examples. This technique can also be used to spot unexpected methods, which the user may not have otherwise expected and possibly ignored.

The Class View aims to aid top down comprehension and allow quick checking of expectations in terms of classes that one would expect in a given situation. It can also provide an easy way to navigate views of lower level details such as finding the source for particular methods.

4. Further Work

Many issues remain within the scope of visualisation of runtime information, both in terms of the visualisation presented in this paper and the problem in total.

The Class View is based on a graph representation, which as previously mentioned, can suffer from problems of scalability. In order to aid this, further development of the view is needed in terms of user filtering and abstractions. At present, the view supports basic user grouping, filtering, and abstraction. These features could be expanded along with greater overview and zoom facilities to help navigation within the graph. Without such aids, the view would become overly complex and difficult to understand, for example, when the number of classes in the system is large or the inter-connection between them is very high. This would result in increased complexity of the view, which could act as an indication that the software needs restructuring, for example, if the coupling between the classes is excessively high. However, it would mean the view was of limited use in understanding the system.

Another issue that could be investigated within the visualisation is the support for the evolution of Java software under study. It would be beneficial if the visualisation could incorporate new information with minimal changes to the existing structure and highlight to the user what has changed. Presently the layout of the Class View uses a force directed layout algorithm. Therefore, the picture presented will not necessarily be the same through the different executions of the same program. The scope for continuous change also poses problems in that the graph can be modified at any time through the execution, therefore the nodes in the graph can change position. Ideally, a layout with minimal

changes between executions and different versions is desirable.

There is scope for greater user customisation, and one such improvement would be to allow the user to easily define the shading scheme and method length mapping function for the Class View so they can customise it for their task. For example, if they are looking for very small changes within a small scale of numbers, they will want to make these more prominent. On the other hand, if they are looking at very large scales, they may only want to give an indication of the order of magnitude.

When monitoring a program there must be a balance met between the amount of information that is extracted and the speed of the execution. For example, the Class View style of method counting could be done for individual objects as well as possible recording all field values for the object. This could aid identification of problems with a specific instance, such as checking if that instance was initialised before use. However, such detailed monitoring will slow the execution of the program and require significantly greater storage space.

The visualisation focuses on the class level interactions, and thread based events such as method calls. Greater work is needed on unifying these, and providing additional visualisations for details, such as object level relationships. As previously mentioned, simply showing this as a graph can lead to problems of scale. Therefore, alternative representations need to be investigated, along with methods of restricting the volume of information into a more manageable form.

The Query View has been implemented with support for a number of views, however, there is currently no support between the Query View and the Class view. This is due to the fact the Query View is currently 3D to support objects from the Runtime View, while the Class View uses a 2D representation. Many issues arise when considering the use of these views together, and further work is needed on assessing how to provide a consistent querying method across all views. Links back from the Query View are also needed in order for the user to ascertain where they dragged the information from, thus allowing them to see it in its context. Queries could also be generated automatically based on some user constraint, such as showing the details of all classes that inherit from a particular class or interface.

The current visualisation only considers programs running on a single Virtual Machine. The tool supports networked connections, so, for example, a client and server on different machines could both be visualised using two instances of the tool. However, there is no support for multiple connections using a single instance of the tool at present. Such an approach would raise many issues in terms of how to amalgamate the different information sources.

5. Summary

This paper has presented a visualisation designed to aid the comprehension of object oriented software with specific focus on showing class level details. This gives an overview of the program's structure in terms of its classes and their relationships, whilst also allowing an overview of method and metric information to be shown. This view allows easy user customisation, and could easily be adapted to show additional information, provided that it can be mapped to a length or colour shading scale.

A prototype tool has been implemented to demonstrate the concepts described in this paper. This tool extracts runtime information on a piece of Java Software using the Java Platform Debug Architecture. The tool has allowed the application of the ideas to real world Java programs. However, the techniques proposed are not restricted to Java and could be applied to other object oriented languages, although new issues would be raised if the language allowed code outside the scope of classes. The Runtime View would also need to be modified, if it was applied to another language, as the thread group hierarchy of the Runtime View relies on the thread group idea of Java.

Further work is being done on improving the current visualisation and increasing interaction between the different elements within the visualisation, as well as looking at different representations for other runtime aspects such as object level relationships.

Acknowledgements

Michael Smith is supported by an EPSRC studentship.

References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides., *Design patterns: elements of reusable object-oriented software*, Addison-Wesley, 1994

[2] M. 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 5th International Workshop on Program Comprehension,* Dearborn, Michigan, U.S.A., pp 17-28, May 28-30, 1997.

[3] A. Von Mayrhauser and A.M. Vans, "Program Comprehension During Software Maintenance and Evolution", *IEEE Computer*, vol.28, no. 8, August 1995, pp 44-55.

[4] 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

[5] Jinsight, IBM Research, Available: http://www.research.ibm.com/jinsight/

[6] Look!, Wind River Systems, Inc., Available: http://www.wrs.com/products/html/look.html

[7] VisiVue, VisiComp, Inc., Available: http://www.visicomp.com/index.html

[8] C. Knight "Smell the Coffee! Uncovering Java Analysis Issues", *Proceedings of First International Workshop on Source Code Analysis and Manipulation (SCAM 2001)* , November 10, 2001

[9] Java™ Platform Debugger Architecture (JPDA). Available: http://java.sun.com/products/jpda/

[10] C. Knight and M.Munro, “Comprehension with[in] Virtual Environment Visualisations”, *Proceedings of the IEEE 7th International Workshop on Program Comprehensio*n, pp4-11, May 5-7, 1999.