VisCG: Creating an Eclipse Call Graph Visualization Plug-in

Kenta Hasui, Undergraduate Student at Vassar College Class of 2015

Abstract

Call graphs are a useful tool for understanding software; however, these graphs can be very large and tend to overwhelm users. To remedy this problem, we created VisCG, an Eclipse plug-in for visualizing the call graphs of Java projects. VisCG uses hierarchies to prune and collapse parts of the call graph. It also helps users focus on the most important parts of the graph by displaying call weights, which are computed statically. VisCG is open source and available for others to use and improve.

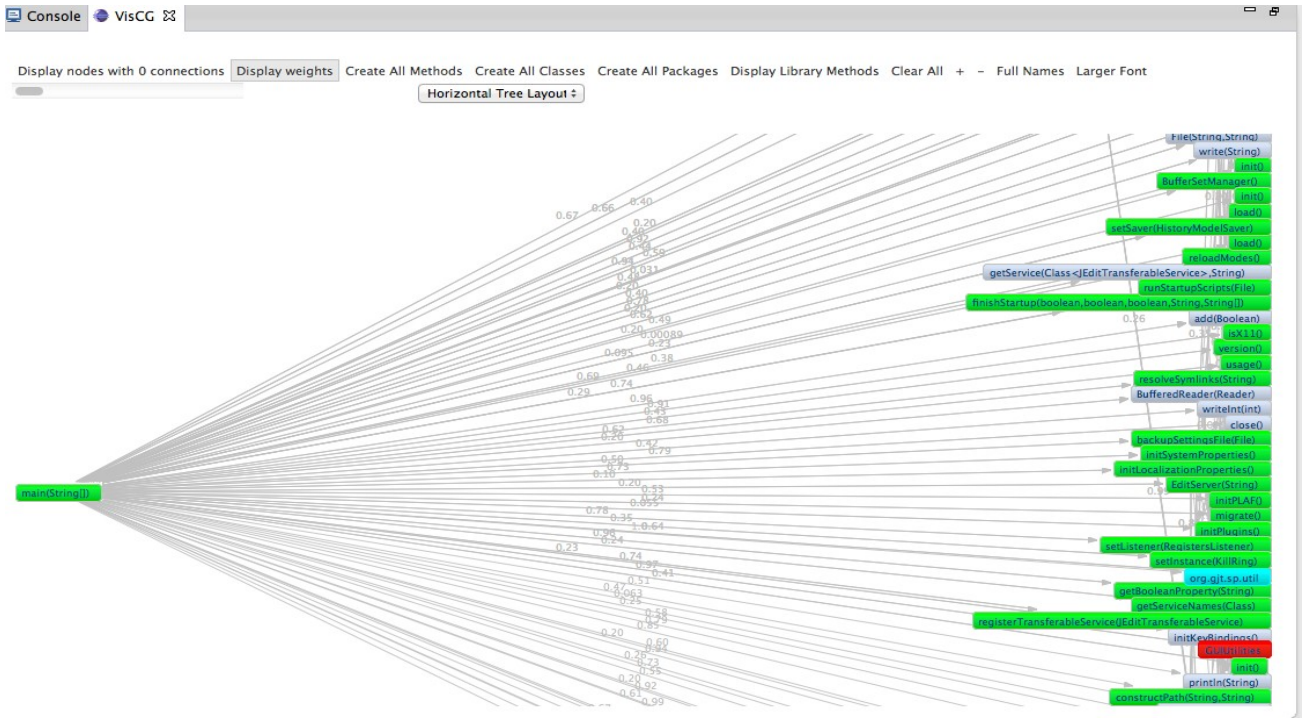

Image 1: VisCG with weights. Mixed levels of abstraction (green, red and blue nodes).

Introduction

As more and more developers learn how to code, programs have grown in number and in complexity. There is a growing need for methods to understand algorithms and to find errors in code. A large part of a software developer's job is software maintenance: fixing bugs, adding new features and modifying code after software is shipped. It can account for up to 70% of total expenses in a given software project [3]. Due to this, analyzing and improving how developers understand code has recently become a large area of research [1].

One of the key tasks in software maintenance is navigating through source code, with *call*

graph based navigation especially important in determining places to implement changes [4]. A call graph represents all methods as nodes. Each method node has outgoing edges connected to its callees and outgoing edges connected to its callers.

We present VisCG, a plug-in for the popular Java Eclipse IDE. It visualizes the *static* call graph of a Java project, which is based on an analysis of the source code without executing it. Unlike most other call graph plugins on the market today, VisCG also has the option to display the "weights" of each call – given a certain method call, how likely it is that the method will indeed be called. We believe that this feature will help developers better understand and visualize their code by allowing them to focus on the more important method calls with the larger weights, rather than the less significant method calls.

Background

Most IDEs today provide ways to find, for a given method M, its *callers* (methods that call M) and its *callees* (methods that are called by M). There are other tools on the market that build on this functionality, such as CallGraph Viewer [7], which draws call graphs incrementally upon user requests. Another example is REACHER [2], which offers upstream and downstream search and shows only methods of interest. These tools, however, suffer from crucial limitations.

The main issue for Eclipse's built-in tools lie in its lack of information. When inspecting an element, Eclipse only displays information directly relevant as a user traverses a program. It only shows direct relationships between immediate neighbor methods. In other words, it cannot display the transitive relationship between two methods which are separated by another method. (For example, if method A calls method B and method B calls method C, Eclipse will not display the relationship between methods A and C.)

In an informal experimental session with a Java/Eclipse developer who used CallGraph Viewer to understand a small library, we found that users can get overwhelmed when seeing large call graphs.

One of the limitations of CallGraph Viewer is that it cannot display the entire call graph. A user has to add callers and callees to the graph, over and over again, which is frustrating and impractical. The plug-in also does not offer different abstraction levels or hierarchical views. Furthermore, it automatically re-adjusts existing nodes in a graph if a new node is added. Finally CallGraph Viewer is not open source, with its last version published in 2008. Others therefore cannot add new features or fix existing bugs.

VisCG, on the other hand, provides more features than CallGraph Viewer and Eclipse to display and analyze call graphs. VisCG remedies the problem of hierarchy and focus by displaying *weighted* call graphs, by offering hierarchical visualizations at the method/class/package levels, and by offering three different forms of navigation. It is also open-source, meaning it is free and open for others to add functionality and improve upon its design.

VisCG

A) Underlying Mechanisms

VisCG is made up of two distinct parts that work in conjunction to create a final graph. First, the program uses an *ASTVisitor,* which is a built-in class in Eclipse. ASTVisitors step through each line of code in a program and perform a certain task or function there. In VisCG, the ASTVisitor visits every method declaration and method invocation. From each one, ASTVisitor extracts the package, class and method names, the number of parameters, and the callee methods. For method invocations, it also calculates the weight of the method call. This is a static trace of the project.

When all of the information is gathered, VisCG uses *Zest* to construct a visual graph and displays it on the screen. *Zest* is a graphical interface for Eclipse based on the built-in JFace and SWT libraries [5]. The buildGraph method initializes the graph. The createPartControl method updates the graph based on user input, rather than creating a new graph each time. This is due to the dynamic nature of this call graph, which provides options for users to change at runtime. The nodes are laid out

according to Zest's layout algorithms. These algorithms maintain optimal spacing between nodes and prevent node overlap to create a visually pleasing graph that is easy to navigate.

B) Display and Node Hierarchies

VisCG provides a hierarchy of three different level of abstraction to display nodes: methods, classes, and packages. Methods are the most specific, lowest-level abstraction, while packages are the most general, highest-level abstraction.

The default view is the method-level abstraction, in which each method in the source code is represented by a square node in the graph. This node contains a label representing the method's method name and the parameter types. This representation is based off of the existing REACHER plug-in [2], and it serves to distinguish overloaded methods with the same identifier while keeping the visualization compact. We have also included an option to display the full method name, including the package and class names. This offers more information than REACHER and leads to a more accurate map. Each method is classified as either a user created method or a library method. User methods, which are declared in the given project, are green. Library methods, on the other hand, are gray.

Each method call is represented by a directed edge between the nodes. The weights of each method call is displayed over each edge. Each node and edge can be highlighted by a user click, to help distinguish a certain method call in a cluttered area. When the plug-in is run, the whole graph is initially displayed. Every method and method call is represented on the graph.

Users can interact with the view to collapse or expand selected nodes into different levels. If a method node is *collapsed* into its class, all method nodes belonging to the same class are disposed and a new class node is created. If a class node is *expanded*, all method nodes for that class are created and the class node is disposed. The same functions exist for packages. Furthermore, there is an option to collapse/expand ALL nodes on the graph to their respective classes and packages.

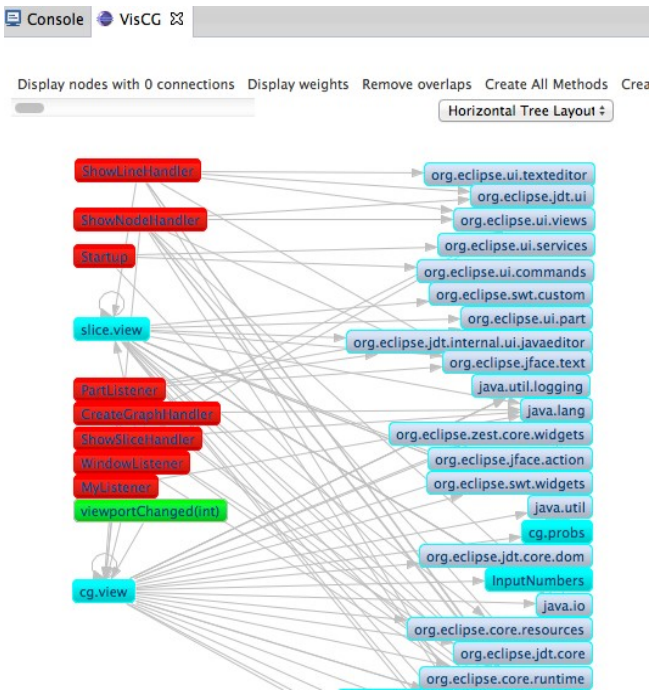

Image 2: Mixed levels of abstraction

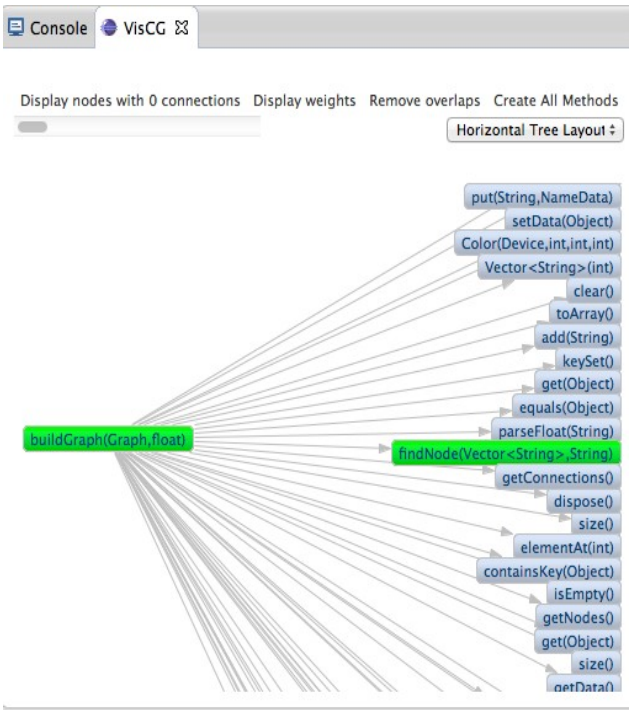

Image 3: Local view of project

C) Edge Weights

VisCG computes probabilities for call edges and displays them as numbers in the call graph (see image 1). Given a source node, the weight represents the likelihood that the source node will call a target node upon execution. These probabilities are computed using a static analysis which has been found to be a good predictor of runtime behavior [8].

We implemented a slider to adjust the weight threshold of the graph; in other words, it allows the user to only see the node connections that have a weight above a certain value. Any edges with lower weights are hidden from view. The further down the slider is, the fewer edges are displayed. All nodes reachable only via hidden edges are also hidden. This feature allows the graph to present only the relevant information to the user. It keeps users from getting overwhelmed and allows for focused study of the code.

D) Global and Local Views

VisCG supports both global and local call graph views. The *global call graph* is the call graph

of an entire program, and the *local call graph* is any portion of a global call graph (See image 3 for an example of the local call graph).

Karrer et al. [1] describe a high level strategy that most developers use to find a certain location in the code to implement a change. This strategy consists of an *exploration phase* and a *traversal phase.* In the exploration phase, developers search for an anchor point to start deeper investigations. Ko et al. [6] also observed this behavior, but referred to it as a *search phase*. During the subsequent traversal phase, users navigate along a specific outgoing path in the call graph until they either find the correct location for a change or notice a dead end and start a new exploration phase. VisCG's *global call graph view* is suited for this exploration phase, as it provides a high-level overview of a given project. By having an option to display all nodes, it allows users to understand the underlying structure of their project and better determine a strong anchor point.

However, the lack of focus on such a view can make it difficult to perform user-centric, method-specific tasks on a large project. Thus VisCG also offers a *local call graph view,* in which the user chooses one method to create, and expands the graph by adding callers or callees to the existing node. This view is especially suited for the *traversal phase.* This makes traversing the graph and understanding the method hierarchy simple and intuitive.

E) Code-View Navigation and Other User Options

VisCG also supports double-click and right-click functionality. By double-clicking on a method node, the Eclipse Java source editor opens the class that contains the specified method. By double-clicking on an edge, the user is directed to the method's invocation. Thus if a user finds a node that requires further examination, they can easily look into the source code for the method's declaration or use. This navigation further supports the *traversal phase* by integrating the code and the graph and allowing for streamlined interactions between the two. Furthermore, the user VisCG provides navigation from the source code to the graph. In the editor, the user can right-click inside a method and choose an option to “show node in call graph.” If the node is already in the graph, the visualization

centers on the chosen node. If not, the node is added to the graph.

VisCG offers five different layouts for the graph: Horizontal Tree layout, Radial Layout, Grid Layout, Spring Layout, and Sugiyama Layout. These help the user see the graph in different forms. Our plug-in also gives users the option to hide or show all library methods.

Future Research

As of August 2014, VisCG has just been completed. Therefore there have not yet been any rigorous studies to find the effectiveness of the plugin. The plugin is still a prototype based on the idea that weighted call graphs would be more effective than a non-weighted one.

Our lab plans to implement a user study to examine how quickly users can solve tasks when given different tools. The participants will be given a program and a feature to add to the program. They will then be split up into three groups. The first group (control) will only be given the source code. The second group will be given the source code and a simple, unweighted call graph. The third group will be given the source code and VisCG's weighted call graph. We will then measure the average time it took each group to implement the changes. This should give us a quantitative measure of VisCG's usefulness.

Conclusion

In this paper we presented VisCG, a new Eclipse plug-in that visualizes the call graph of Java projects. It is more useful than existing call graph visualization tools due to its ability to focus the information it displays. The plug-in helps users hone in on the most important parts of the call graph (and therefore the project code) by using weights and hierarchies to prune and collapse areas of the call graph. VisCG supports different views and layouts of the graph, is open source and integrated into Eclipse. It is a free, advanced call graph navigation tool for Java.

References

1. T. Karrer, J. Krämer, J. Diehl, B. Hartmann, and J. Borchers. “Stacksplorer: Call Graph Navigation Helps Increasing Code Maintenance Efficiency”, in *Proc. UIST'11 Symposium on User Interface Software and Technology.* New York: ACM, pp. 217-224.
2. T. LaToza, B. Myers, and J. Aldrich. “Answering Reachability Questions”, *TOSEM 13’.* In submission.
3. R. S. Pressman. *Software Engineering: A Practitioner’s Approach.* McGraw-Hill, 7th edition, 2010.
4. J. Krämer, J. Kurz, T. Karrer and J.Borchers. “Blaze: Supporting Two-Phased Call Graph Navigation in Source Code”, in *Proc. CHI EA '12 Extended Abstracts on Human Factors in Computing Systems.* New York: ACM, pp. 2195-2200.
5. Eclipse Foundation, “Zest,” [online] 2014, http://www.eclipse.org/gef/zest/ (Accessed: 17 June 2014).
6. A. Ko, B. Myers, M. Coblenz, and H. Aung. “An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks.” *IEEE Transactions on Software Engineering, 32(*12):971–987, 2006.
7. “CallGraph Viewer,” http://marketplace.eclipse.org/content/ callgraph- viewer.
8. Y. Zhang and R. Santelices, “Predicting Data Dependences for Slice Inspection Prioritization,” in Proceedings of IEEE International Workshop on Program Debugging, Nov. 2012, pp. 177–182.