Static Stack Visualization: Supporting Programmer Navigation and Understanding by Providing Method Context via Possible Stacks Visualization

Proposal

Joachim Kurz
Media Computing Group
RWTH Aachen University
joachim.kurz@rwth-aachen.de

June 7, 2011

0.1 Introduction

Maintenance tasks are common.

Software development does not stop when a product is deployed, instead it is usually necessary to adapt the software product to changing requirements [Sommerville, 2007, p. 489]. This is called maintenance and costs to do so often make up the majority of the total costs during the software lifecycle [Sommerville, 2007, p. 489]. Thus it is important to support programmers during maintenance tasks.

Understanding the complete program is not required for a maintenance task.

Since such maintenance tasks usually involve changes in the program code it is important to understand the program to estimate the effects a change will have and implement the change in the most effective way. Often it is not necessary to understand the complete program but just a part of it [Erdös and Sneed, 1998]. One of the basic questions to answer to understand enough of a program according to Erdös and Sneed [1998] is "where is a particular subroutine or procedure invoked?". Also, programmers spent a lot of time navigating in source code [Sherwood, 2008] and tools making navigation in code easier should thus be welcome.

Stacksplorer gives only little support for searching across paths.

Stacksplorer [Krämer, 2011] already answers the question identified by Erdös and Sneed [1998] and supports navigation along the call graph. The call graph is the graph that is constructed from program source code by interpreting each method/function as a node and adding a directed edge between method a and method b iff a calls b. But Stacksplorer only displays the direct successors and predecessors in the call graph but developers also have questions that they try to answer by searching across paths. Also, they do not only search upstream looking for callers of a method (as implied by Erdös). They also search downstream (following method calls), for example, to better understand why a method is called [LaToza and Myers, 2010] by identifying methods deeper in the call graph that cause (expected) side effects. Stacksplorer does support up- and downstream navigation but only one level at a time.

To support developers in searching across paths we would like to extend Stacksplorer during this thesis to support the visualization of possible call stacks thus providing a nar-

row look into the depth of the call graph in contrast to Stacksplorer's direct-neighbor-focused breadth-first view of the call stack. In the following we will first describe the related work we were inspired by and based our work on to then describe what we are planning to work on during this bachelor's thesis.

0.2 Related Work

There are many different ways to visualize a program's structure in general (not just by displaying different kinds of program structure graphs like call graphs or control flow graphs[1]). Since we do not plan to provide an overview over the complete code but only over the surroundings of a specific part (in our case the method in the call graph) we will not focus on complete program visualization approaches but instead look for work which tries to visualize only a part of the program structure. Edge cases in that way are projects like Code Bubbles [Bragdon et al., 2010] which allows the developer to choose which parts of the code they want to focus on and displays the relations between the chosen code segments. However, we do not want to create a whole new Integrated Development Environment (IDE) but integrate the visualization with Xcode[b] , an existing IDE, more like an accessory view in the same way Stacksplorer works.

We will focus our research on partial visualization of program structure, not complete visualization.

Another edge case are projects that use degree of interest functions to select which context to display for a currently viewed code segment like [Jakobsen and Hornbæk, 2009] but are not limited to call relations. We will have a look at those projects to see whether they have useful visualization techniques but will not use their degree of interest functions

[1] The control flow graph is similar to the call graph, but its nodes are single statements in the code (for example assembler instructions or single source code lines) and there is an edge between node a and node b iff b can be executed in some scenario directly after a has been executed. For example, a line before an if-else-statement would be connected by one edge to the first line of the if-clause and by another edge to the first line of the else-clause.

[b] http://developer.apple.com/Xcode/

and instead focus on visualizing related methods by using the call graph.

0.2.1 Graph and Tree Visualizations in General

A call graph is usually a proper[2] directed graph not just a tree – not even acyclic (because of recursion). But by constraining the visualization to one focus method plus its descendants and ancestors and replacing recursion (=cycles) by special nodes the part of the call graph to visualize can be transformed into a directed acyclic graph (DAG) or maybe even a tree. Thus we will mainly look at DAG and tree visualization.

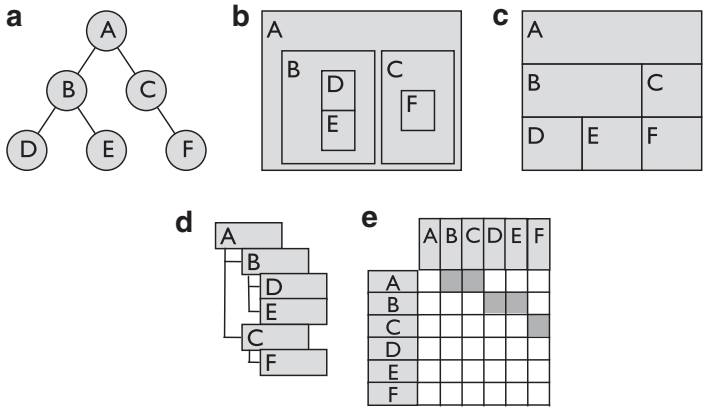

Figure 1: Five different tree representation styles. Redrawn from [Graham and Kennedy, 2010].

Graham and Kennedy [2010] identify 5 types of basic tree visualizations (see Figure 1):

a) node-link diagram

b) nested diagram

c) adjacency diagram

d) indented list, as used in the Microsoft Windows Explorer and several IDEs (see 2.4)

[2] as in non-acyclic and possibly more than one predecessor per node

e) matrix representation

But they do not consider the matrix representation in their evaluation of tree-visualizations because they disregard it as too complicated and space-inefficient for trees. There are studies showing that the indented list is subjectively preferred by users and other studies implying that this is only due to the familiarity of the users caused by the lists usage in Windows [Graham and Kennedy, 2010]. Andrews and Kasanicka [2007] compare four hierarchy browsers, one of them an indented list (which they call tree view), two nested diagrams and one node-link diagram. They could not find a significant difference in completion time when letting the users explorer different hierarchies except for one pair of browsers for one out of 8 different tasks. Since there is no clearly favorable visualization we think there is room for improvement and hope a more focused visualization will perform better for our use case.

There seem to be no big differences between different basic tree visualizations.

0.2.2 Fisheye Views

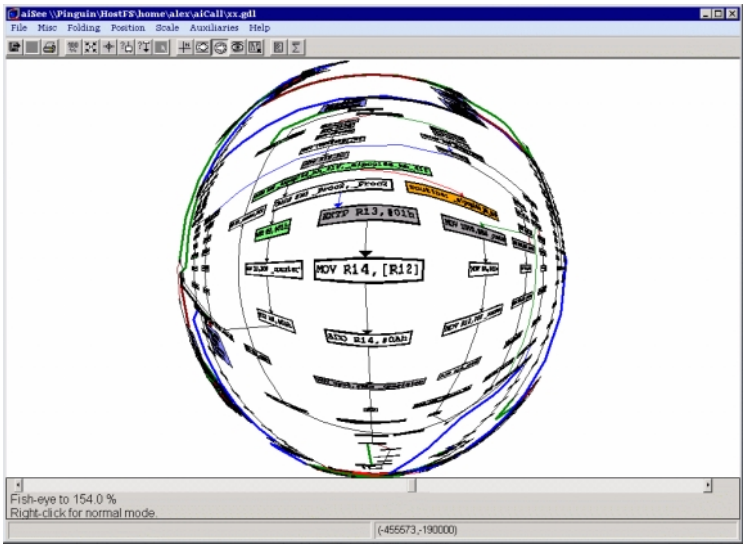

Figure 2: A fisheye view of a control-flow graph. Redrawn from [Evstiougov-Babaev, 2002].

Fisheye views highlight a focus point and distort distant features.

One graph visualization method respecting the idea of having a focus point in a graph and displaying context of different importance is the concept of fisheye views described by [Furnas, 1986]. Often, they are just applied as an optical effect to existing graph drawings by zooming in on a focus point and distorting nodes that are far away from the focus point (see Figure 2). But fisheye views are not lim-

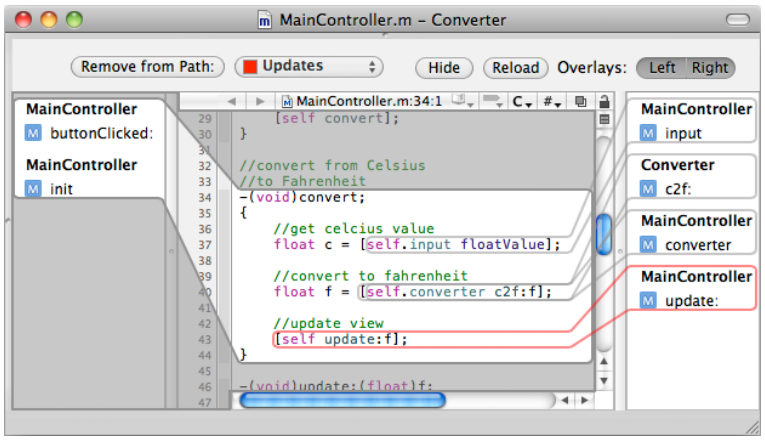

Figure 3: Stacksplorer displays the callees of the current focus method on the left of the editor view and the called methods on the right. Overlays are used to show where the methods on the right are called in the code. Picture from the Stacksplorer Website[b].

ited to optical effects there are also semantic ones. [Jakobsen and Hornbæk, 2009] modified the Eclipse IDE Editor to display relevant lines from the same file as the currently viewed code below and above the editor view and could show that their modified editor was adopted and actually used in real-life work.

0.2.3 Strahler Numbers

Strahler numbers[3] have been developed in hydrology to measure stream networks but have recently been used in more general graph applications as well. They measure the

[3] Wikipedia gives a nice overview[c] over Strahler numbers on binary trees, for other trees there are several generalizations

[b] http://hci.rwth-aachen.de/stacksplorer
[c] http://en.wikipedia.org/wiki/Strahler_number

complexity of subtrees and are generally higher if a node has more branches. Although they were originally defined on binary trees but have been generalized to n-ary trees and DAGs. In this case they have the property that they are equal to the minimum number of registers needed to compute an n-ary expression encoded by such an n-ary tree[4]. They have even been generalized to special kinds of non-acyclic graphs. [Auber, 2002]

Strahler numbers are a measure of the complexity of subtrees.

Herman et al. [1998] used Strahler numbers to highlight paths that lead to bigger/more complex subtrees giving hints to the user which paths are worth further exploration, while exploring the graph in a zoomed-in state. Auber [2002] used these numbers to prioritize edge rendering of large graphs so a bare skeleton could already give an impression of the overall graph before the rendering finished.

Strahler numbers have been used for graph navigation and prioritizing graph rendering.

0.2.4 Current IDEs

Several of the more popular IDEs already have call graph visualization features but they all use different kinds of tree views/indented lists to visualize them (see Figure 4 to 7). Eclipse, NetBeans and IntelliJ also only allow you to view just one side of the call graph (either callees or callers) at a time.

Most big IDEs use tree views to show the call hierarchy. But they are often not implemented very well.

Microsoft Visual Studio displays a node for callees and one for callers in the same view but one has to expand this collection of callees/callers before being able to see them and has to expand a similar collection on each level down the call hierarchy. Even worse: One can interleave callee and caller relationships in this tree view thus at one point the child relation might mean ”callee” and further down the tree it might mean ”caller” (see Figure 7 for an example).

Also none of the mentioned tree views updates automatically when another method is selected. The user has to invoke it explicitly for a method they find interesting. Thus there seems to be room for improvement.

[4] An n-ary arithmetic expression can be encoded as an n-ary tree and vice-versa

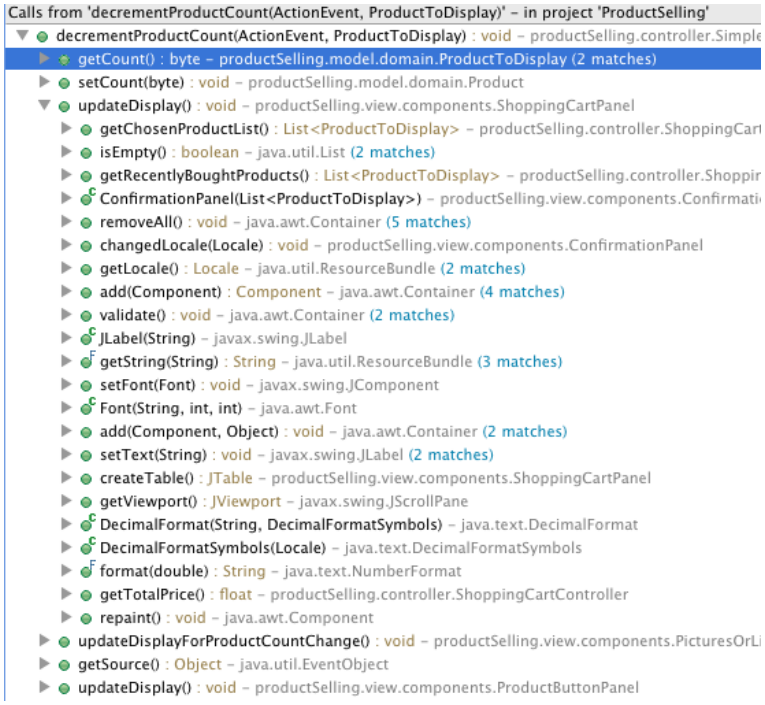

Figure 4: Eclipse's call hierarchy view: Only one direction (callees or callers) is viewable at a time.

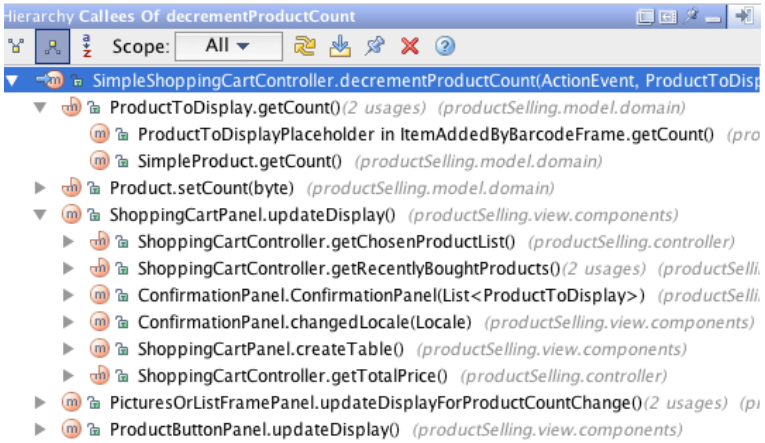

Figure 5: IntelliJ's call hierarchy view: Only one direction (callees or callers) is viewable at a time.

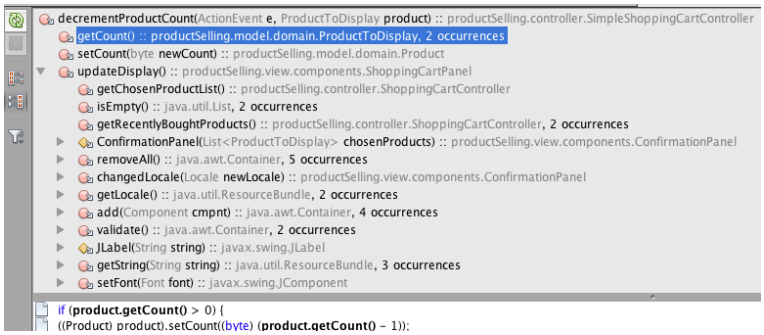

Figure 6: NetBeans' call hierarchy view: Only one direction (callees or callers) is viewable at a time.

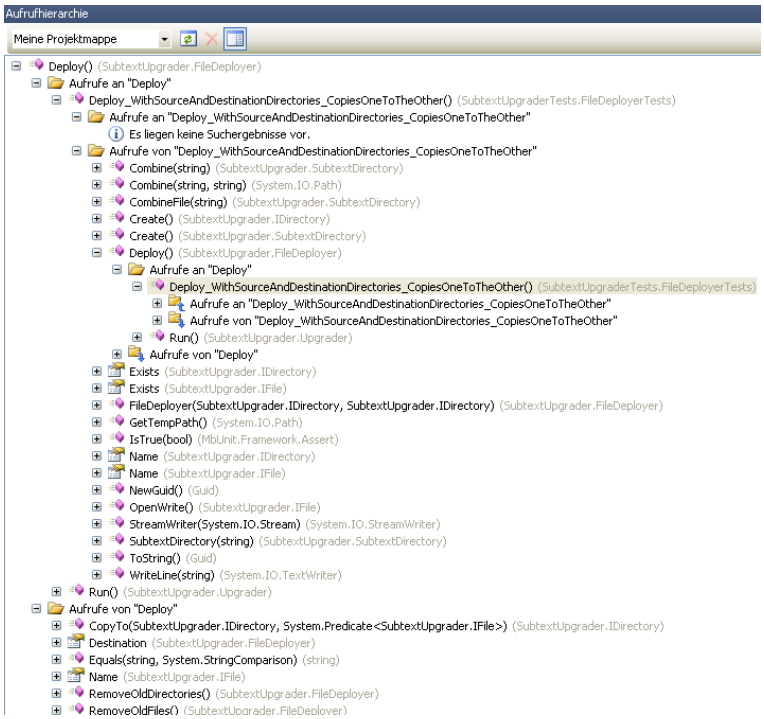

Figure 7: Visual Studio's call hierarchy view: Both directions can be see at a time but directions can be interleaved as well (Caller → Callee → Caller visualized as children).

0.3 Our Work

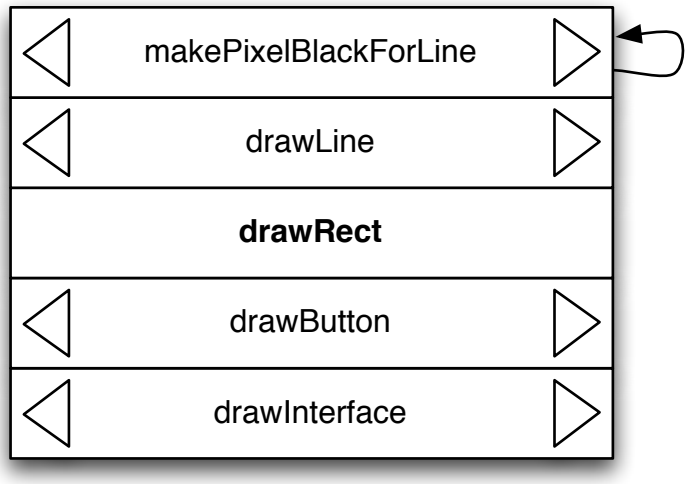

Figure 8: A sample visualization of a potential call stack. The arrow buttons left and right of a method can be used to exchange the methods by some call sibling methods. The arrow at the side of the top method symbolizes that this method is called recursively. The drawRect method is the current focus method and thus there are no other choices and thus no arrow buttons for it.

We would like to implement the depth-view as an accessory view similar to Stacksplorer.

As mentioned in the introduction developers ask questions that can be answered by searching across paths. Stacksplorer already provides some support to do so but we think this support can be improved since Stacksplorer does not provide the depth-view of the call graph. Thus, we would like to implement a view on the call graph providing this perspective. It should still support navigation along the call graph as Stacksplorer does. We also think the approach of Stacksplorer to implement this view as an accessory view that can be used while writing code is good because this way the programmer can still work on the code and and does not have to switch modes. Thus we would like to keep the visualization compact and small to fit beside the editor view.

As an accessory view we do not have a lot of screen space. However, as Stacksplorer has shown it is possible to narrow down the editor view a bit in the horizontal dimension without affecting the code readability too much, especially on todays wide-screen displays. If we have a part of the width of the display to ourselves we can use the complete height, thus our view will probably be higher than wide.

A tree view would fit these space requirements but similar to Stacksplorer its focus is on all of the direct successors and to see a complete path one usually has to expand all the children on the path manually. And then it still shows a lot of unnecessary information if all one is interested in is one specific path.

A treeview is to cumbersome to use.

One visualization that is already used in many debuggers and has the property of being usually higher than wide is that of a stack of methods. We would like to extend this concept by providing a mechanism to exchange a method by one of its siblings to customize the stack that is shown. We imagine this to work like a combination lock where each ring (=set of method siblings) can be turned independently. Although in our case turning one ring would affect the descending methods. A very basic drawing of this concept can be seen in Figure 8.

We would like to extend the stack visualization to a combination lock like visualization.

Reducing the number of shown methods to just one stack at a time causes the problem of which stack to show by default when the user has not chosen anything yet. To find a decent initial stack we plan to order the siblings by their Strahler number and selecting the ones with the highest value. This way we have a least a consistent order of the siblings. The Strahler number of a method might correlate with the relevance of the method to the developer but we could not find any research about this relationship and proving or disproving this is out of the scope of this thesis. We will also make use of the tags that can be assigned to methods with the current version of Stacksplorer to find more relevant methods.

Possibilities to choose a default stack include order in the graph, tagged methods and Strahler numbers.

We will implement a tree view as well.

To evaluate our new call graph view we also plan to implement a basic tree view (which Xcode does not provide so far) and then compare our implementation to this basic tree view and the original Stacksplorer implementation in a user test.

0.3.1 Preliminary Time Schedule

The scheduled time for a bachelor's thesis is 4 months. Since we already started to work on it a few weeks ago we will take the 9th of May as the start date. That would mean that we have to be finished on the 9th of September. That makes 17 full weeks. Using this calculation we will probably have a few days buffer in the end in case something goes wrong. We have already read a few papers and skimmed a lot more but we would like to read some more to start with a solid understanding of programmer navigation behavior, program understanding and different graph visualizations. After that we will come up with a design of the stack visualization and requirements specification for the plugin as a whole. We plan to test the design as a low-fidelity prototype in some quick qualitative user tests. Then we spend a lot of time implementing the actual prototype and evaluating it. In the end we will write up all the results in the thesis.

what	until when	how long
read (more) papers	May 30th	3 weeks
design & test Lo-Fi prototype & write down requirements	June 13th	2 weeks
implement prototype	July 18th	5 weeks
design user study	July 25th	1 week
do user study	August 8th	2 weeks
write thesis	Sept. 5th	4(.5) weeks
		17 weeks

Bibliography

Keith Andrews and Janka Kasanicka. A Comparative Study of Four Hierarchy Browsers using the Hierarchical Visualisation Testing Environment (HVTE). In *2007 11th International Conference Information Visualization (IV '07)*, pages 81–86. IEEE, 2007.

David Auber. Using Strahler numbers for real time visual exploration of huge graphs. In *International Conference on Computer Vision and Graphics*, pages 56–69, sep 2002.

Andrew Bragdon, Robert Zeleznik, Steven P. Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J LaViola Jr. Code bubbles: a working set-based interface for code understanding and maintenance. In *Proceedings of the 28th international conference on Human factors in computing systems*, pages 2503–2512, New York, NY, USA, 2010. ACM.

Katalin Erdös and Harry M. Sneed. Partial comprehension of complex programs (enough to perform maintenance). In *6th International Workshop on Program Comprehension. IWPC'98*, pages 98–105, SES Software Eng. Service, Budapest, jun 1998. SES Software Eng. Service, Budapest, IEEE Comput. Soc.

Alexander A. Evstiougov-Babaev. Call graph and control flow graph visualization for developers of embedded applications. In *Graph Drawing*, pages 337–346, AbsInt Angew Informat GmbH, Saarbrucken, Germany, 2002. AbsInt Angew Informat GmbH, Saarbrucken, Germany.

G. W. Furnas. Generalized fisheye views. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 16–23, New York, NY, USA, 1986. ACM.

Martin Graham and Jessie Kennedy. A Survey of Multiple Tree Visualisation. *Information Visualization*, 9(4):235–252, dec 2010.

Ivan Herman, Maylis Delest, and Guy Melancon. Tree visualisation and navigation clues for information visualisation. *Computer Graphics Forum*, 17(2):153–165, jun 1998.

Mikkel Rønne Jakobsen and Kasper Hornbæk. Fisheyes in the field: using method triangulation to study the adoption and use of a source code visualization. In *Proceedings of the 27th international conference on Human factors in computing systems*, pages 1579–1588. ACM, apr 2009.

Jan-Peter Krämer. Stacksplorer Understanding Dynamic Program Behavior. Master's thesis, RWTH Aachen University, jan 2011.

Thomas D. LaToza and Brad A Myers. *Searching across paths*. ACM, may 2010.

Kaitlin Duck Sherwood. Path exploration during code navigation. Master's thesis, The University of British Columbia (Vancouver), aug 2008.

Ian Sommerville. *Software Engineering*. Pearson Education Limited, Essex, England, 8 edition, 2007.