Stacksplorer: Understanding Dynamic Program Behavior

Jan-Peter Krämer, Thorsten Karrer, Jonathan Diehl, Jan Borchers
RWTH Aachen University
52056 Aachen, Germany
{kraemer, karrer, diehl, borchers}@cs.rwth-aachen.de

ABSTRACT
To thoroughly comprehend application behavior, programmers need to understand the interactions of objects at runtime. Today, these interactions are often poorly visualized in common IDEs except during debugging. Stacksplorer allows visualizing and traversing potential call stacks in an application even when it is not running by showing callers and called methods in two columns next to the code editor. The relevant information is gathered from the source code automatically.

ACM Classification: H5.2 [Information interfaces and presentation]: User Interfaces. - Graphical user interfaces.

General terms: Design, Human Factors, Languages

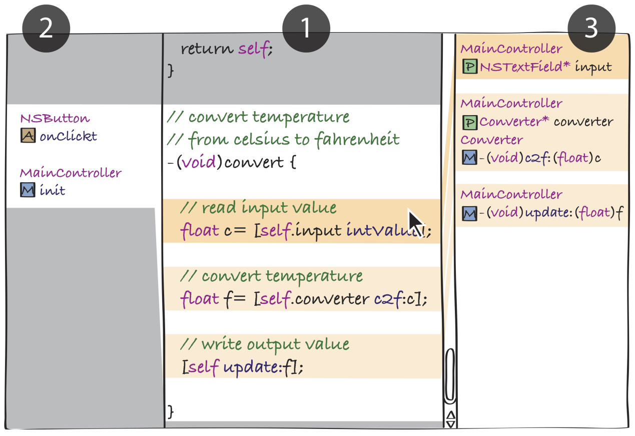
Keywords: Programming, navigation, IDE.

INTRODUCTION
In object-oriented programming languages, source code is typically distributed across multiple files each containing a number of classes and their methods. Interactions of methods at runtime such as the call stack, however, are equally important to help the developer understand how an application works [4]. The file-based structure of source code does, in most cases, not reflect these interactions between methods appropriately.

Current IDEs primarily rely on support for navigation between different source code files. Information about possible call stacks can often be revealed even without a debugger when the application is not running, but visualizations for this information are far from perfect. Exploring how a method is used in the context of a call stack still needs lots of time [3]. In a preliminary study, that we conducted ourselves, we additionally found that many programmers subjectively consider existing tools offered in *Xcode*[1], Apple's standard IDE, unsatisfying for their tasks.

Stacksplorer helps programmers to see and navigate through an application's method calls that are automatically extracted from the code. After determining which kinds of interactions between objects at runtime, apart from the call stack, are important for programmers, we designed an appropriate visualization and interaction for this information and for navigating through it. By implementing these within an IDE, we want

[1]http://developer.apple.com/technologies/tools/xcode.html



```
    return self;
}

// convert temperature
// from celsius to fahrenheit
-(void)convert {

  // read input value
  float c= [self.input intValue];

  // convert temperature
  float f= [self.converter c2f:c];

  // write output value
  [self update:f];

}
```

Figure 1: Stacksplorer utilizes horizontal navigation to explore potential call stacks of an application.

to prevent programmers from getting lost in their code, help them find relevant information more easily, and increase their awareness of potential side effects of changes.

RELATED WORK
By observing programmers working on software maintenance tasks, Robillard [4] identified two basic strategies to explore the runtime behavior of source code: Structured traversal of the call stack, and skimming through the source code in an opportunistic way. Task success was higher for users applying the first strategy.

In a similar study, Ko [3] found that programmers mostly traverse the call stack by searching for a method name globally in all source code files of their project. However, the results then have to be inspected manually, because method names can be ambiguous. Alternatively, developers use a tool that allows navigating to the implementation of a method. Such a tool is included in most common IDEs. Navigation from the occurrence of a method name to its implementation, however, represents only one direction of a potential call stack. Navigation in the other direction, i.e., to methods calling a particular method, is not widely supported. Both approaches do not visually support understanding how methods interact at runtime. Ko also frequently observed navigation back and forth multiple times between two files, because the previously visible code disappears when navigating.

Numerous systems to improve this understanding have been proposed: Singer and Kersten [5, 2] propose filtering the files shown during navigation by a *degree of interest* determined by the developer's previous activities. Code Bubbles [1] presents a user-selected set of related source code information. However, to add new methods to this set of relevant information, techniques similar to those required in regular IDEs have to be applied.

DESIGN

We designed our prototype as a plugin for Xcode. Therefore, we ran a short preliminary study confirming that developers using Xcode and Objective-C exhibit the same navigation behavior and tool usage as found in previous studies.

From previous work and our own study, we can conclude several implications for our design. Firstly, we have to support two different kinds of source code browsing strategies. For users with structured browsing behavior, our system should make the call stack accessible more easily and the navigation through the call stack faster. If opportunistic strategies are applied, our system provides less targeted support. However, the displayed information will still be relevant.

Secondly, we can conclude from user preferences that supporting visualization of and navigation along the call stack is crucial in order to make the system beneficial to developers. The call stack also includes information about access to instance variables, which usually takes place through accessor methods.

A paper prototype of our tool is shown in Fig. 1. The central editor (1) is equivalent to Xcode's standard editor, retaining all its features and functionality. The cursor in this window marks our *focus method*. The side columns (2,3) show other methods with which our focus method interacts at runtime. The left hand column (2) shows methods *calling* the focus method, the right column (3) shows methods that *are called* by it. The information in both side columns is gathered and updated automatically with no user interaction required at any point.

In addition to navigating through a class by scrolling vertically in the editor, our design allows navigating horizontally through the call stack, by clicking a method in one of the side columns. For example, navigating to a method that calls the focus method, will cause all 3 columns to shift to the right. The method that was selected moves to the center and opens in the central editor (1), the previous focus method appears in the list of called methods to the right (3), and the left column is updated with new information (2). Important paths through the code may also be stored for later reference.

Because our design occupies additional screen space horizontally, it works best on high-resolution, wide-screen displays. We allow collapsing the side columns in case they are not needed, to accomodate for smaller screens.

IMPLEMENTATION

Implementing the prototype as a plug-in for Xcode is possible through a private plugin API. To replace the standard editor in Xcode with our interface, we use Objective-C's reflection capabilities to change Xcode's view classes accordingly. Additionally, we can reuse existing parsing engines in Xcode to extract possible call stacks from the source code. Xcode contains a lexicographic source code scanner (lexer) that operates on a single file. A project indexer is also included that extracts the static object hierarchy, including information about all methods and instance variables, from all source code files in a project.

These parsing engines allowed us to quickly build a first prototype, without having to use a static code analyzer. To determine the methods called from the focus method, we use the source lexer to find all method calls in the implementation of the focus method, and search the appropriate methods in the project index. To find the methods calling the focus method, we run a project-wide search for the focus method's name in the background. Because method names can be ambiguous, we then filter out all method calls that are performed on an object having a different type than the class containing the focus method. In our first prototype, these naïve algorithms work reasonably fast and produce very reliable results. However, we are not able to determine if a method call we found is unreachable due to preconditions that can never be met during real program executions. This would be possible using a static code analyzer.

SUMMARY AND FUTURE WORK

Stacksplorer provides programmers with a convenient way to view and traverse potential call stacks in their source code. This can make developers more effective when implementing changes and may help them to gain code comprehension more quickly. Because the prototype is implemented as an Xcode plugin, developers can still use all existing features of that IDE.

Our prototype is currently in a beta status, but already fully functional for Objective-C source code. We intend to test this software prototype with developers to evaluate its effectiveness. We will also investigate how other interactions of objects at runtime, for example through an instance variable, can be appropriately visualized and explored.

REFERENCES

1. A. Bragdon, R. Zeleznik, S. P. Reiss, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola. Code bubbles: a working set-based interface for code understanding and maintenance. In *Proc. CHI*, New York, USA, 2010. ACM.

2. M. Kersten and G. C. Murphy. Mylar: a degree-of-interest model for IDEs. In *Proc. AOSD*, volume 05, pages 159–168, Chicago, Illinois, 2005. ACM.

3. 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 TSE*, 32(12):971–987, Dec. 2006.

4. M. P. Robillard, W. Coelho, and G. C. Murphy. How Effective Developers Investigate Source Code:An Exploratory Study. *IEEE TSE*, 30(12), 2004.

5. J. Singer, R. Elves, and M.-A. Storey. NavTracks: Supporting Navigation in Software Maintenance. In *Proc. ICSM*, pages 325–334. IEEE, 2005.