

Answering Common Questions about Code

Thomas D. LaToza

Institute for Software Research

School of Computer Science, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213

tlatoza@cs.cmu.edu www.cs.cmu.edu/~tlatoza

ABSTRACT

Difficulties understanding update paths while understanding code cause developers to waste time and insert bugs. A detailed investigation of these difficulties suggests that a wide variety of problems could be addressed by more easily answering questions about update paths that existing tools do not answer. We are designing a feasible update path static analysis to compute these paths and a visualization for asking questions and displaying results. In addition to grounding the questions we answer and tailoring the program analysis in data, we will also evaluate the usefulness of our tool using lab and field studies.

Categories and Subject Descriptors

D.2.6 [Software Engineering]: Programming Environments; D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement.

General Terms

Human Factors, Verification

Keywords

Code navigation, callgraph, program comprehension, empirical study, science of design, feasible paths

1. INTRODUCTION

Before successfully editing code to make a change, developers try to understand relevant unfamiliar code. Tools to support this activity have long been designed from many perspectives. Tools have been built to traverse dependencies in code (slicing [10]), recommend relevant code to be inspected [11], check method contracts [4], and reverse engineering diagrams from code [9].

The key insight of this dissertation is that this activity can be more effectively supported by better models of typical developer activity during code change tasks. Most existing

tools rely on intuition or anecdotes to identify challenges developers face understanding unfamiliar code. A central tenet of modern human computer interaction is that gathering data about the context and nature of user activity is essential for designing useful tools [2]. We have followed this approach by conducting exploratory studies of developer activity, finding mismatches between tools and developer needs, and designing new interactions that directly support developer needs.

Our key finding to emerge to date is that developers ask questions about update paths linking application state changes to effects. Based on this finding, we are designing a static analysis to compute feasible update paths and a visualization to display these paths annotated with other relevant information. This thesis will contribute 1) improved models of fact finding activity, 2) a feasible update path analysis, 3) a visualization of update paths, and 4) an evaluation of the tool's usefulness.

2. PROGRAM COMPREHENSION AS FACT FINDING

To understand developer activity while changing code, we observed 13 developers in the lab make two complicated 1.5 hour changes to an open source application [6]. To explain our results, we formulated a theory of program comprehension as *fact* finding. A fact is a belief a developer holds about code (e.g., a method has effects). Facts unify constructs across several areas - hypotheses from program comprehension; information from code navigation; design decisions, dependencies, concerns, and design rationale from software design; and constraints from verification - to provide a unified account of the change process. Developers seek facts by reading methods, use their knowledge to learn facts describing code, critique facts they believe are poor design, explain facts to understand the consequences of changing facts, and propose and implement changes.

- Facts were useful both for suggesting proposed changes and for generating constraints to rule out proposals that should not be implemented. In order to understand constraints on proposed changes and accurately reject infeasible proposals, participants spent much of their time trying to explain why facts were true.
- Developers made *path choice decisions*, choosing between further investigating a proposal, abandoning it in

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICSE'08, May 10-18, 2008, Leipzig, Saxony, Germany.

Copyright 2008 ACM 1-58113-000-0/00/0004...\$5.00.

favor of another, and implementing a proposal to see if it works. Poor path choice decisions led to developers abandoning better proposals for inferior proposals.

- Developers explored code poorly, introducing false facts. Developers failed to ask questions they never realized were relevant, failed to test assumptions, failed to find answers they were unable to locate, and found incorrect answers.
- Several developers reported being overwhelmed trying to remember facts they discovered. We hypothesize this limited the number of proposals they could consider and facts they could find.

A key design implication of our results is that several seemingly diverse problems arise from challenges in answering questions about code. Weighing the time and distraction of answering a question against the likelihood it would reveal useful facts, developers often chose to stay with their assumptions or attempted less risky, but lower quality changes. Some questions were hard to answer, leading developers to make assumptions and implement changes, only to realize after testing that the proposal would never work, wasting time investigating, implementing, and testing infeasible changes. These underlying problems resulted in abandoned changes, hacks, bugs, wasted time, and frustration.

3. UPDATE PATH QUESTIONS

To find questions a tool should answer, we analyzed breakdowns (e.g., questions participants took a long time to answer, bugs they inserted) to identify 12 hard-to-answer questions. Though the tasks had been designed only to be challenging and to require understanding the design, we discovered a common theme in both tasks - update paths. An *update path* is code connecting *triggers*, a change in application state, with *effects*. For example, participants in one task were concerned with connections between opening a file in a text editor and 7 status bar updates.

A variety of evidence supports the case that the problems we observed arose from understanding update paths. Developers verbalized facts they learned as links between triggers and effects they expected to occur:

“Buffer tells the text area, hey I just inserted something. The text area then asks the buffer what his fold level is, at which point we then are recalculating the fold level and that's going on in buffer.”

Developers asked questions about update paths:

“I'm concerned that I won't get all of the events that cause this guy to get updated. And I'm not sure, with the existing tools in Eclipse, how to find out all the places that can cause this thing to be called.”

Developers stated false beliefs about update paths:

“All right, right, so I think it's safe to say that when a transaction completes and we do the computation we can update all the fold state. Then, from that point on, any calls from getFoldLevel doesn't require computation.”

Resulting in wasted time – this is after 24 minutes of investigation and debugging:

“And we know that the fold update is ok because we're doing it from the first invalid line to this line. So we're good. Excellent. Let's make sure that switch buffer doesn't do anything. OHHHH. No! No!!!”

In addition to recording and transcribing think-aloud data, we also manually transcribed every navigation event where developers changed the method visible on screen. Developers navigated through update paths, but also followed infeasible update paths, wasting time understanding task irrelevant code. Finally, we analyzed the changes to find bugs. Many of these were also attributable to false beliefs about update paths.

Specific challenges answering update path questions underlie these problems:

- Developers could not distinguish infeasible from feasible paths. Existing call graph views, such as Eclipse's call hierarchy, allow incremental expansion of all feasible and infeasible paths from a target. If A calls B and B calls C, the path A, B, C is infeasible if guards on the call to C are never true in the context of the call from A. Our preliminary investigations suggest showing only feasible paths often significantly reduces the fan-out that makes conventional call-graphs difficult to use, particularly for methods dispatching on events.
- Developers could not follow paths to specific trigger methods. Using the call hierarchy to traverse backwards from effects to triggers requires traversing all paths until either the trigger or the beginning of the path is found. For nontrivial programs, traversing all paths is usually impractical. In practice, developers guessed incorrectly which paths came from the trigger.
- Developers could not understand how update paths interacted. Some developers proposed changes with unchecked assumptions about paths they had not investigated.
- Developers could not understand the class structure of paths. Developers made false assumptions about object ownership, object relationships, and objects affected by path changes.

While our results were limited to two lab tasks, a study [8] of 25 developers across 27 field and lab tasks also observed update path questions (e.g., “how is control getting (from here to) here?”).

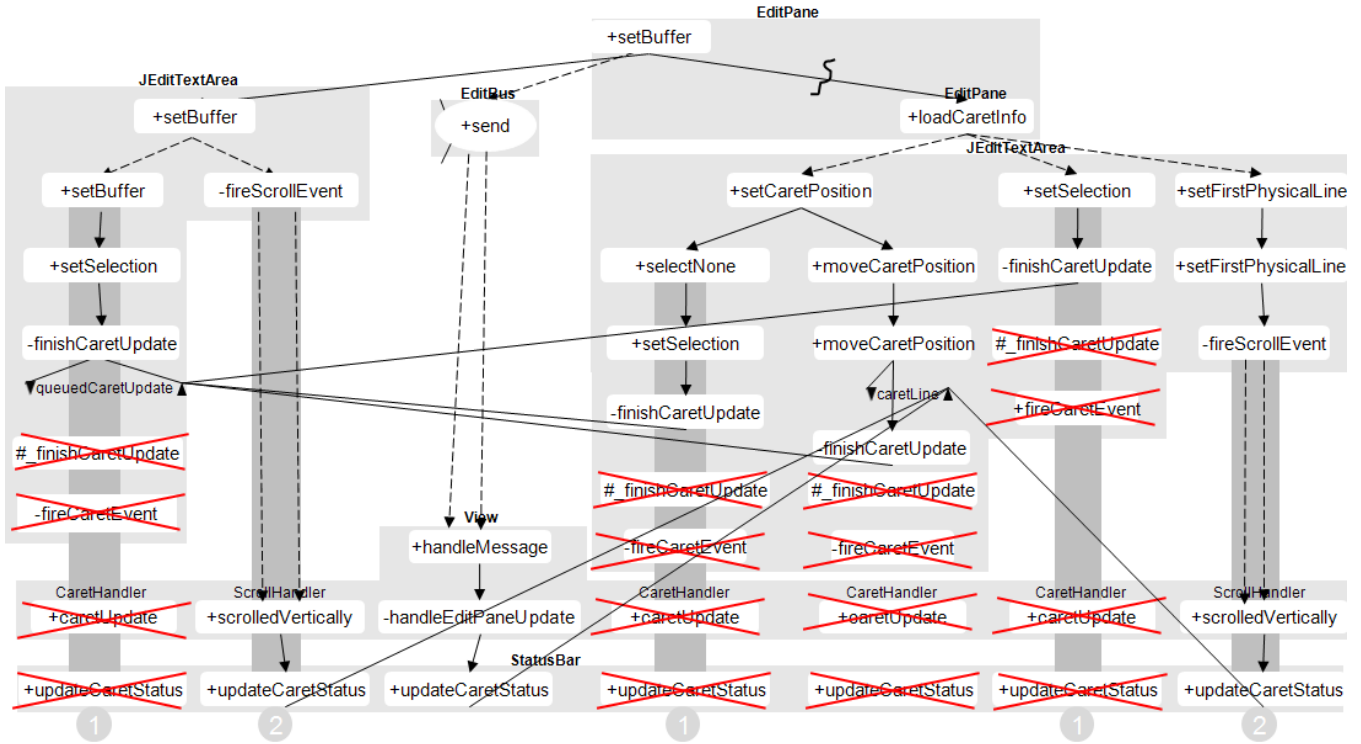


Figure 1. A mockup of a preliminary WhatTree visualization. Method contexts (white boxes) are shown in containing classes (grey boxes) and grouped by recurring sequences (numbered shaded regions). Calls (lines) either may (dashed) or must (solid) happen, may happen more than once (two lines), and sometimes are spawned on new threads (thread icon). Fields (text) are read (up arrow) and written (down arrow), depicting problems such as stale data (first read of caretLine). Editing code may cause visible contexts to no longer be on feasible paths (crossed out), revealing unexpected changes.

Existing program analyses for program understanding have been dominated by slicing and chopping. A slice is the set of program points affected by a statement [10], not feasible paths. While useful for debugging tasks where the objective is finding a divergence between actual and expected program values to locate a bug, slicing tools cannot compute feasible paths. In contrast, documentation tools can describe feasible paths overlaid on class structure - UML sequence diagrams [7], use case maps [3] - but lack analyses for reverse engineering them from code.

4. FEASIBLE UPDATE PATH ANALYSIS

To build a tool for answering update path questions, this thesis will develop a static analysis to compute these paths. Propagating constraints on variables (e.g., `queuedCaretUpdate = false`) along paths determines if a path through a guard is feasible (e.g., `if (queuedCaretUpdate) finishCaretUpdate()`). While a model checker computes such paths, current model checkers [1] are specialized for finding low frequency bugs and therefore may compute very precise abstractions. This results in generating many paths and being too slow for an interactive tool. Inter-procedural dataflow analyses compute the set of dataflow facts that hold at a program point but cannot directly determine which paths are feasible.

We are designing an analysis for these queries that is more precise than existing inter-procedural dataflow analyses but less precise and faster than existing model checkers. Intra-procedurally, a path-sensitive dataflow analysis propagates constraints to create a tree of feasible paths summarizing each method. Each interior node in the tree is a test instruction (e.g., `instanceOf`, `==`, `!=`) with children for the test failing, succeeding, and being indeterminate. Method call nodes along the path have constraints on their parameters. Interprocedurally, we propagate constraints from actual to formals to pick the appropriate path. We do not track constraints on arithmetic variables, preventing an explosion of paths through loops. Examples from our tasks have few calls intraprocedurally control dependent on arithmetic constraints, and we will measure how generally this holds.

Many update paths from our study involved constraints on fields - for example, a Boolean field set to false later guards a sequence of methods leading to several effects. We will investigate designs for propagating constraints through fields. Some key challenges we may address include modeling separate instances of classes and tracking aliasing. We will use real corpora of code to make decisions on what precision / performance tradeoffs to make.

5. THE WHATTREE

To use our analysis to directly answer questions about update paths, we are designing the WhatTree. A WhatTree allows developers to explore update paths to answer questions about the effects of a method (e.g., what does this do? why do I need this call?) and triggers causing a particular effect (e.g., when does this happen?). A WhatTree visualizes *feasible* update paths as trees of method contexts, field reads, and field writes. We are designing a visualization of these paths that supports both answering questions about update paths and keeping track of the answers. The visualization will show methods, fields, and paths as developers ask questions expressing interest and hide them when developers decide they are no longer interesting. Methods are laid out so the in-order traversal of the tree is the execution order. The WhatTree annotates update paths to answer some questions without having to view the source code. We hope this will better support generating and evaluating change proposals from visible information. Our prototype displays information that we observed some developers make use of in our tasks. We will iterate the design to show other information we observe being used in our evaluations.

We will also explore interactions for using the WhatTree. Filtering paths from assertions (e.g., `assert queuedCaretUpdate = false`) may help explore what-if scenarios. Specifying sets of triggers, rather than a single trigger, may help compare update paths.

6. EVALUATION

To evaluate the utility of the WhatTree, we will conduct lab and field studies to determine the ways in which WhatTree helps developers ask questions more effectively and to quantify the magnitude of these effects. We hope that developers will be able to more quickly answer questions, backtrack off of false exploration paths more easily, propose more and better design changes reflecting a more global understanding of design problems, more rapidly and accurately reject proposals, and more accurately and rapidly recover from interruptions by recovering the “working set” [5] of the methods and fields that the developers were working on. Developers will make changes to Java code either using the WhatTree or Eclipse, and we will use a combination of time spent on activities, bugs, think-aloud, and post-task questionnaires to understand how the WhatTree influences how developers work. We will also give the WhatTree to developers in the field and observe what questions they are able to answer directly with the WhatTree and what questions they still look at the code to answer.

7. CONCLUSIONS

Careful observations of developers have revealed update path questions developers ask that are hard and error prone

to answer but that existing tools do not answer. A static analysis designed to balance precision / performance tradeoffs will compute these paths. The WhatTree will answer these and other common questions about code.

8. ACKNOWLEDGEMENTS

This research was supported by NSF grant IIS-0329090, the EUSES consortium under NSF ITR CCR-0324770, and an NSF fellowship.

9. REFERENCES

- [1] T. Ball and S. K. Rajamani. Automatically Validating Temporal Safety Properties of Interfaces. In *SPIN 2001, Workshop on Model Checking of Software*, LNCS 2057, 103-122, 2001.
- [2] H. Beyer and K. Holtzblatt. *Contextual Design*. Academic Press, 1998.
- [3] R. J. Buhr and R. S. Casselman. *Use CASE Maps for Object-Oriented Systems*. Prentice Hall, 1996.
- [4] L. Burdy, Y. Cheon, D. Cok, M. D. Ernst, J. Kiniry, G. T. Leavens, K. Leino, and E. Poll. An overview of JML tools and applications. In *Software Tools for Technology Transfer*, 7(3), 212-232, 2005.
- [5] A. J. Ko, B. A. Myers, M. Coblenz, and H. H. Aung. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. In *IEEE Transactions on Software Engineering (TSE)*, 32(12), 971-987, 2006.
- [6] T. LaToza, D. Garlan, J. Herblseb, and B. Myers. Program Comprehension as Fact Finding. In *European Software Engineering Conference and Foundations of Software Engineering (ESEC/FSE)*, 2007.
- [7] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.
- [8] J. Sillito, G. C. Murphy, and K. De Volder. Questions programmers ask during software evolution tasks. In *Foundations of Software Engineering (FSE)*, 23-34, 2006.
- [9] M.-A. D. Storey and H. A. Müller. Manipulating and documenting software structures using shrimp views. In *International Conference on Software Maintenance (ICSM)*, 1995.
- [10] M. Weiser. Program slicing. In *International Conference on Software Engineering (ICSE)*, 1981.
- [11] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining Version Histories to Guide Software Changes. In *IEEE Transactions on Software Engineering (TSE)*, 31, 429-445, June 2005.