Answering Reachability Questions

Thomas David LaToza

May 2012

CMU-ISR-12-104

Institute for Software Research
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements
for the Degree of Doctor of Philosophy

This work was supported by the National Science Foundation, under NSF grants IIS-0329090, IIS-0534656, CCF-0811610, and IIS-1116724, by the EUSES Consortium under NSF grant ITR CCR-0324770, and by an NSF Graduate Research Fellowship; and by Microsoft Research. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author and do not necessarily reflect those of the National Science Foundation.

Keywords: call graphs, program comprehension, code exploration, reverse engineering, productivity, Java, Eclipse, static analysis, program visualization

ABSTRACT

Software developers understanding and exploring code spend much of their time asking questions and searching for answers. Yet little has been known about the questions developers ask, the strategies used to answer them, and the challenges developers face. Through interviews, surveys, and observations, a series of 7 studies were conducted that begin to address this gap, contributing a better understanding of developers' tools, practices, problems, questions, and strategies, and a model of how developers reconstruct design decisions from code. A design process is described for using studies of developers' work to design more useful tools for developers.

These studies reveal that *reachability questions* are a central part of understanding and exploring code. A reachability question is a search along paths through code. Developers ask reachability questions when reasoning about causality, ordering, type membership, repetition, and choice. For example, to debug a deadlock, a developer searched downstream for calls acquiring resources to reconstruct how and why resources were acquired. Existing tools make these questions challenging to answer by forcing developers to guess which paths through the call graph lead to what they are looking for and which paths are feasible and execute. These challenges cause bugs and waste time. In one study, half of the inserted bugs were caused by challenges answering reachability questions; other developers simply gave up. In observations of professional developers at work, nine of the ten longest debugging and investigation activities involved answering a single reachability question, each requiring tens of minutes of developers' time.

To help developers more easily understand and explore code, REACHER lets developers search along call graphs and find matching statements. An interactive call graph encodes causality, ordering, type membership, repetition and choice and helps developers to remain oriented while navigating through code. REACHER is implemented as an Eclipse plugin for Java. REACHER uses a novel fast feasible path analysis to eliminate some of the most common types of infeasible paths. In a controlled experiment, developers with REACHER were 5.6 times more successful answering reachability questions in significantly less time and reported that REACHER helped them to think more visually.

ACKNOWLEDGEMENTS

First, I'd like to thank all of the faculty with whom I was fortunate enough to work. Jim Herbsleb helped introduce me to research in empirical software engineering. I am deeply indebted to my advisors, Brad Myers and Jonathan Aldrich. Brad Myers was always there to provide lots of feedback and make my work better. Jonathan Aldrich helped bring a software engineering and programming languages perspective to my work. And my committee members – Tom Ball and Niki Kittur – provided many useful insights.

I'd like to thank Andy Ko for being an awesome academic older brother. As an undergad at the University of Illinois, my first discovery of user-centered design of tools for developers was by reading his papers. Over the years at Carnegie Mellon that I was fortunate enough to spend with Andy, I picked up some of Andy's extensive knowledge of HCI research methods and time management. I always looked forward to talking with Andy, as I always come away feeling more inspired about research and having learned something important.

I'd like to thank Rob DeLine, Gina Venolia, and Andrew Begel for being great hosts and collaborators at Microsoft Research. The Human Interactions in Programming team is the perfect place for studying human aspects of software development.

I'd like to thank Brad Myers' other students for many great paper discussions and debates about research – Jeff Stylos, Stephen Oney, Kerry Chang, and YoungSeok Yoon. I'd like to thank my officemates for many fun conversations about life and, occasionally, research – Marwan Abi-Antoun, Ciera Jaspan, Vishal Dwivedi, and Ben Towne. And I'd like to thank all of my fellow SE Ph.D. students whom I've been fortunate enough to get to know – George Fairbanks, Owen Chang, Uri Dekel, Greg Hartman, Kevin Bierhoff, Chris Scaffidi, Nels Beckman, Sven Stork, and Josh Sunshine.

Thanks also to all of my awesome friends who made my time in Pittsburgh fun! From Settlers of Catan to traveling the world to afternoon volleyball to rock climbing to starting a band to running two marathons to two musical productions (!!!!) – you helped me do amazing things I never thought possible until I met you.

Thanks to the people that helped support my research. Thanks to the National Science Foundation for providing funding. And thanks to the many participants in my surveys, interviews, and observations for providing hours of your time!

Finally, I'd like to thank my parents, whose boundless knowledge about everything and attention to detail helped inspire me to do research, and who are always supportive of whatever I do!

TABLE OF CONTENTS

FIGURES

TABLES

1. INTRODUCTION

Exploring code is a central part of constructing software. Developers work in large codebases ranging from hundreds of thousands to tens of millions of lines of code. As developers implement features, fix bugs, and make code more maintainable, they explore the code. Performing these tasks in large codebases requires achieving task-specific understanding of small pieces of code immersed in a huge codebase. Developers explore code to achieve this understanding.

But what exactly does exploring code entail? What questions do developers ask? What strategies do developers use to answer these questions, and what challenges do they face? While software engineering has long speculated about developers' questions and strategies for answering them, there has usually been little evidence behind this speculation. Instead, anecdotes, toy examples, and accepted wisdom inspire and motivate tools. When this speculation is wrong and does not describe real challenges that developers face, tools produced from this speculation are unlikely to impact software development practice (see Sections 3.2 and 3.3). Indeed, one of the few studies to evaluate code exploration tools through a user study found **no effects** of any of three recent tools [AMR07].

1.1. Designing code exploration tools from data

This dissertation begins by describing a process for using data about how developers work to design tools that solve important problems developers face. To inform the design of more useful code exploration tools and gather data about how developers understand and explore code, this dissertation then describes a series of exploratory studies. A series of two surveys and interviews of developers in the field examines the context of exploring code – its relationship to activities, tools, practices and problems that developers themselves perceive when dealing with large and complex codebases. The next study investigates how developers explore and understand design decisions in code and the effects of design knowledge in shaping and influencing this process. Finally, the space of questions developers ask during coding tasks is examined through a survey examining hard-to-answer questions and the relationship of these questions to current software development tools and practices.

1.2. The problem

From these open-ended exploratory studies, a specific, hard-to-answer, frequent, error-prone, time-consuming, and tool-supportable class of questions emerged – *reachability questions*. Many investigation and debugging tasks are, in essence, search tasks – developers seek statements which answer a question. Traditional searches find sets of statements in a program which match the search criteria. In contrast, a reachability question is a search along paths through a program for statements matching the search criteria. Developers often wish to find statements with specific control flow connections. For example, to debug a deadlock, a developer wished to find methods downstream from an event that acquired resources (see Table 7.2). To debug a null pointer exception in an object, a developer tried to understand what paths might exist in which the object is used before first being initialized (see Section 7.4).

More formally, a reachability question consists of two parts: the paths to search and the search criteria specifying the statements to find (see Tables 7.1 and 7.2 for examples of reachability questions). Reachability questions represent feasible paths as a set of concrete traces *TR.* A concrete trace *tr* is a list of *<s, env>* tuples, where *s* is a statement and *env* maps every variable in *s* to a value. *traces(p, O, D, C)* is the set of all concrete traces in a program *p* from an origin statement *o* in the set *O* to a destination statement *d* in the set *D* which satisfy all the filtering constraints *c* in *C*. *O, D,* and *C* can be left unspecified by using a *?* (although at least one of *O* or a *D* must be specified). Questions without an origin are called *upstream* reachability questions while questions with an origin (and optionally a destination) are *downstream* reachability questions. *C* is a set of filtering constraints *c*, where *c* is a tuple *<s, x, const>* specifying a value for a variable *x* in *s*. *x* and *const* can be left unspecified (?) to find only the traces containing *s*. A trace *tr* satisfies a constraint *<s, x, const>* when ∃*<s, env>* in *tr* s.t. *env(x) = const*. If a trace contains multiple copies of a statement in different contexts, it satisfies the constraint if the constraint is true in any of the contexts.

There are two types of reachability questions: *find* and *compare*. *find SC in TR* finds the portion of each *tr* in the set of traces *TR* that matches search criteria *SC.* A search criteria SC is a set of statements. A search criteria *function*, given attributes describing a set of statements, generates a *SC*. Table 1.1 lists search criteria functions we observed in our studies, each of which produce a set of statements *SC*. A reachability question then matches the statements s_{sc} in a *SC* against each *<s, env>* tuple in a trace *tr,* selecting those traces in *tr* where ∃*<s, env>* in *tr* s.t. $s_{sc} = s$.

Note that the search criteria functions are defined independently of traces through the program and any other notion of feasibility. For example, for each method *m* in *M*, *callers(M)* will find all call sites with the method name *m*. Similarly, *dDepend(s, x)* finds all of the reaching definitions of *x* at *s* (statements that might be the previous assignment of *x* at *s*). Determining which of these statements are feasible and relevant happens through the

Function	**Finds the set of statements that:**
grep(str)	include text matching the string *str*
reads(F), writes(F)	read / write a field *f* in the set of fields *F. FIELDS* is the set of all fields in the program.
stmts(T)	are in a type *t* in the set of types *T*
stmts(M)	are in a method *m* in the set of methods *M*
callers(M)	are callsites of a method *m* in the set of methods *M*
callees(M)	are method declaration statements of methods invoked by a method *m* in the set of methods *M*
ends	are method calls to framework methods without source or are method declaration statements with no callers
dDepend(x)	are data dependent on a variable *x*. *dDepend(x)** finds the transitive closure including transitive data dependencies.

Table 1.1. Search criteria functions describing statements for which developers searched (see Tables 7.1. and 7.2.).

reachability questions, which intersects these statements against the statements in traces through a program.

compare$(TR_a, TR_b) : TR_{common}, TR_1, TR_2$ compares sets of traces. Compare first attempts to match, by some method, each tr_a in TR_a to a corresponding trace tr_b in TR_b. When such a match is found, compare then attempts to match (by an unspecified method) tuples $<s_a, env_a>$ in tr_a to corresponding tuples $<s_b, env_b>$ in tr_b. This generates three new lists: tr_{common} which contains an ordered list of tuples that matched, and tr_1 and tr_2 which contain an ordered list of tuples in tr_a and tr_b that did not match. TR_1 and TR_2 are the set of all tr_1 and tr_2, respectively, and also contain traces in TR_a and TR_b for which no match could be found. TR_{common} is the set of all tr_{common}. Note that compare is not implemented by REACHER (see Section 11.1.2 for a discussion of how REACHER might be extended to support *compare* questions).

Reachability questions reflect how developers use control flow to reason about programs. The ubiquity of reachability questions stems in large part to the expressiveness of control flow, and its resulting ability to answer many questions (see Table 1.2). Control flow is often represented as a *control flow graph* which contains an edge from statement *a* to *b* when there exists an execution in which *b* executes immediately after *a*. In imperative programs, control flow expresses *causality* between a call site statement and a method. Calling a method causes statements in it (and statements in methods it transitively calls) to execute. Determining when something happens requires finding the control flow by which it may be reached. And control flow expresses the *order* in which statements execute.

Question	**Related downstream search**
What parts of this data structure are accessed in this code?	Search downstream for accesses to the data structure
What parts of this data structure are modified by this code?	Search downstream for writes to the data structure
What data is being modified in this code?	Search downstream for writes to any field
What exceptions or errors can this method generate?	Search downstream for throws or error calls
How do calls flow across process boundaries?	Search downstream for out of process messages
How is control getting from *a* to *b*?	Search downstream from *a* for *b*
What resources is this code using?	Search downstream for calls accessing or acquiring resources
What are the possible actual methods called by dynamic dispatch here?	Search downstream across feasible paths
How does this code interact with libraries?	Search downstream for calls to libraries
What is the difference between these similar parts of the code (e.g., between sets of methods)?	Compare statements downstream from each method
Question	**Related upstream search**
Is this tested?	Search upstream for unit test methods
What threads can reach this code or data structure?	Search upstream for thread creation calls
What is the "correct" way to use or access this data structure?	Search upstream for paths along which data structure is used
What is responsible for updating this field?	Compare paths along which field is written
What in this structure distinguishes these cases?	Search for reads from the structure upstream from each case
In what situations or user scenarios is this called?	Search upstream for framework callbacks denoting user actions
When during the execution is this method called?	Search upstream for framework callbacks or main methods
What parameter values does each situation pass to this method?	Search upstream for values which flow into parameters
Is this method or code path called frequently, or is it dead?	Search upstream from method or code path

Table 1.2. Many of developers' questions are related to reachability questions.

Developers work to understand a program's control flow throughout investigation and debugging activities as they mentally model, reason, and navigate. For example, when investigating an unfamiliar codebase, developers first mentally construct a control flow representation of connections between its parts [P87]. And their knowledge of a method's part of the call graph increases as they interact with its code [FMH07]. Information foraging theory predicts that developers traverse control flow and search for "prey" – locations in code – by using "scent" – the similarity of the information which labels the control flow edges to their knowledge of their prey – to rank the potential of edges to traverse [LBB08].

Ensuring control flow is easily understandable has been an important goal of language design. Following Dijkstra's observation that `goto`s obfuscate control flow, making reasoning difficult [D68], language designers introduced structured programming languages that simplify control flow within methods [DDH72]. But in order to promote reuse and modularity, modern languages obfuscate interprocedural control flow between methods with features such as dynamic dispatch and indirection. For example, an analysis of code in Adobe's desktop applications found that one third of the codebase is devoted to event handling logic which in turn caused half of the reported bugs [P06]. Successfully coordinating dependencies among effects in loosely connected modules can be very challenging [E09].

Interprocedural control flow is often visualized using a *call graph*. Modern Integrated Development Environments (IDEs) such as Eclipse and Visual Studio let developers see and navigate call graphs with commands ranging from *go to definition* (from a call site) to providing a tree view for exploring call paths. Unfortunately, developers report that understanding control flow remains difficult [DKC05]. As a result, developers exploring code often lose track of control flow relationships between where they are and where they have been, becoming disoriented and lost [AM06].

To better understand the challenges developers face answering reachability questions and requirements for tools to make answering these questions easier, this dissertation describes a series of studies into how developers ask and answer reachability questions while they are understanding and exploring code. The first study revealed that problems answering reachability questions cause bugs. Due to the difficulties when trying to definitively answer reachability questions, developers often guess and make unchecked assumptions. These problems answering reachability questions were responsible for half of the bugs inserted during the study. A second study investigated the frequency and difficulty of reachability questions and found that, as developers gain experience or learn a codebase, answering these questions becomes neither less frequent nor easier. Finally, the third study observed 17 professional software developers at work in their day-to-day coding activities. 9 of the 10 longest investigation and debugging tasks involved answering a single reachability question, each requiring tens of minutes of investigation.

1.3. Addressing the problem – an approach

With existing tools, developers answer reachability questions by manually traversing control flow paths, in search of statements. What if this work was instead performed by a tool? Instead of traversing, developers would simply enter a search, and the tool would find the matching statements. To help understand control flow among methods, developers could see a visualization of control flow among the methods they are investigating. And to stay oriented, developers could use this visualization as a navigation aid.

An additional challenge developers face when exploring static call graphs is infeasible paths. An infeasible path is a path that can never execute. It is caused by correlations between conditional statements in code – certain combinations of branch choices at conditionals will never be taken. As developers traverse paths, developers must track and determine which of these paths are feasible. Infeasible paths will seem to connect portions of the call graph that cannot actually be connected. Manually traversing these paths to check for their feasibility forces developers to think about details of the path to which they could otherwise be oblivious. But an automated tool that followed infeasible and feasible paths of any length might further bury the developer in irrelevant information, by showing many long and complex paths that are infeasible.

To reduce the number of infeasible paths developers face, this dissertation presents a new fast feasible path analysis. While existing tools such as model checkers can eliminate a large class of infeasible paths, their great precision also makes them slow, often taking hours or days, and they are often unscalable to large programs. For an interactive tool being used on a constantly changing codebase, this is impractical. However, examples of common infeasible path idioms suggests a simpler approach. In many situations, conditionals are controlled by constants such as flags, the runtime type of an object, or messages sent on a bus. Intuitively, knowing only where the path begins (e.g., a method handling a user input event) and tracking constant values through assignments to variables along the path may be sufficient to determine the branch taken at a conditional. To further reduce the latency developers face when using the tool, fast feasible path analysis builds a summary of possible paths through each method in the codebase and then uses these summaries to generate paths when a user invokes a search. As a result, fast feasible path analysis is usually able to generate call graphs in under 2 seconds of analysis time while eliminating many frequent types of infeasible paths.

In its final sections, this dissertation describes REACHER – a tool for searching along control flow. REACHER lets developers search along paths and find statements using a variety of attributes. REACHER helps developers answer some of the most frequent types of reachability questions we observed – searches along control flow – but does not support filtering or comparing paths or searching along data flow. REACHER helps developers understand control flow by depicting causality, ordering, type membership, repetition, recursion, and conditionality. REACHER is fully integrated within Eclipse as a plugin, allowing developers to open methods in an Eclipse editor and use REACHER to remain oriented as they explore code.

To evaluate REACHER, we conducted a lab study in which developers used REACHER to answer simple reachability questions. Compared to Eclipse, developers using REACHER were over **5 times** more successful in significantly less time.

This dissertation investigates this thesis:

> Reachability questions are a frequent, time-consuming, hard-to-answer, and error-prone part of understanding and exploring code. A tool that eliminates common infeasible paths and helps developers to search, visualize, and navigate call graphs allows developers to understand and explore code significantly faster and more successfully.

1.4. Contributions

This dissertation contributes the following:

- 10 studies of understanding and exploring code
  - A survey of developers' use of time and problems (**Activities Survey**)(Chapter 4)
  - Interviews of developers about time and problems (**Activities Interviews**)(Chapter 4)
  - A survey of developers' practices (**Follow-up Survey**)(Chapter 4)
  - Lab study of code exploration (**Exploration Lab Study**)(Chapter 5 and Section 7.1)
  - A survey of developers' reachability questions (**Reachability Survey**)(Chapter 7.2)
  - A survey of developers' hard-to-answer questions about code (**Questions survey**)(Chapter 6)
  - Field observations of developers' reachability questions (**Reachability Observations)**(Sections 7.3 and 7.4)
  - A paper prototype study of REACHER (**Paper Prototype Study**)(Section 10.1)
  - Two lab studies of REACHER (**REACHER Lab Study 1**)(Section 10.2) & (**REACHER Lab Study 2**)(Section 10.3)
- A model of how developers explore and reverse-engineer design in code and the effects of expertise on this process.
- Evidence that answering reachability questions is error-prone, frequent, and time-consuming.
- A new approach to exploring code and answering reachability questions by searching along control flow paths.
- A Fast Feasible Path Analysis that eliminates some of the infeasible paths caused by constant-controlled conditionals and that constructs static traces from source code.
- A new, interactive visualization of call graphs.

- Evidence that searching along control flow and visualizing call graphs helps developers explore code more quickly and successfully.

1.5. Outline

The rest of this dissertation begins with a discussion of related work, which spans several areas – studies of developers, tools for exploring code, and path-sensitive static analysis. Next, a user-centered process is described for designing useful tools for developers by gathering data about how developers work. Chapter 4 presents three studies that examine the context of code exploration, investigating its frequency, relationship to other activities, and associated tools. Chapter 5 investigates the role of expertise and uncertainty in reconstructing software design from code. Chapter 6 investigates developers' information needs in coding activities in detail. Chapter 7 presents a lab study, survey, and field observations into a specific challenge exploring code – answering reachability questions. To help developers more effectively answer reachability questions using a tool, a fast feasible path analysis is then described for producing call graphs. Using this analysis, Chapter 9 describes a tool for helping developers more easily answer reachability questions. This tool is then evaluated and refined through a small, pilot paper prototyping study and two larger lab study evaluations. Finally, Chapter 11 concludes and surveys potential future work.

2.

RELATED WORK

Related work for this dissertation falls into three broad areas.

Researchers have long studied how professional software developers understand and explore code. The oldest studies, in the psychology of programming literature, focused primarily on mental representations of programs and the influence of these representations on comprehension. These studies also show how this knowledge influences how experts work. More recently, studies have shifted focus to the activities in coding tasks, including work on questions developers ask, how developers traverse relationships, and the effects of disorientation during coding tasks.

Researchers have designed a wide variety of tools and techniques for helping developers explore code. One popular approach is to ask developers to write a description of important aspects of a method – such as pre and post conditions – in a contract. Developers might then be able to read the contract of a method and never need to explore the implementation of a method's callers or callees. Specification, verification, and testing systems have investigated a wide range of approaches for contracts. Other tools help developers explore code by helping them to identify methods by providing recommendations, a query system, or supporting traversals. Still other tools embody a variety of designs for visualizing call graphs.

REACHER's fast feasible path analysis builds on a long and diverse set of techniques for analyzing code. Many existing techniques have explored approaches for attempting to eliminate infeasible paths through the program.

2.1 Studies of developers

2.1.1. Studies of program comprehension and the effects of knowledge

For many decades, researchers have studied the activity of programming through the lens of psychology [D02]. Applying ideas from the study of cognition, much research focused on how developers mentally represent computer programs and the effects of these representations.

Studies have shown that control flow is an important part of mental representations of programs. Using a priming technique to examine the mental representation of programs, one study found that professional developers learning an unfamiliar program first used a control flow representation [P87]. Another study asked developers in the field to answer questions about their code as they worked and found that developers' knowledge of a method's portion of the call graph increases as they work with it [FMH07].

Numerous studies have found that developers do not mentally represent source code literally but instead recognize instances of schemas. *Schemas* are templates with situation-specific slots. For example, a developer might have a `for`-loop schema with slots for the iteration condition and the operation done to the collection. Using this schema to recall code, the developer might forget situation specific information and, for example, recall the loop index variable to be `i` instead of `j` [D90]. Studies of expertise in other domains have found that many of the advantages of experts arise from their large library of schemas. For example, while chess experts remember realistic boards better than novices, their advantages vanish for random boards [CS73]. This and other results suggest that experts "chunk" what they perceive to mentally represent it in memory as schema instantiations. While there is much evidence for the existence and importance of schemas, studies of schemas in programming have been limited to highly localized code idioms (e.g., `for` loops) and have not investigated schemas at the level of design (e.g., design patterns).

Several studies have found differences between experienced and inexperienced developers working with code. Experts debug faster by generating better hypotheses while studying less code [GO86]. Experts write down low-level information while novices write down higher-level information, perhaps due to differences in how experts and novices work [D93]. Experts better understand code before changing it and better choose when to instantiate schemas. Experts select from multiple strategies for accomplishing tasks, are capable of generating multiple alternatives before making a choice, and design top-down more from high level ideas to low level ideas for familiar and simple problems [D02].

2.1.2. Studies of code exploration

More recently, attention has shifted to studies designed to elicit design recommendations for better tools or practices by identifying information needs and questions associated with different software engineering activities. A number of these studies are relevant to how developers explore code.

At the most general level, several studies have characterized how developers spend their time. In one study [PSV94], thirteen developers on a large software project logged every hour for a year which of 13 activities they were engaged in. The categories distinguished different life cycle activities such as estimation, requirements, high level & low level design, test planning, coding, inspections, and high level & low level testing. Most developers reported being in a coding stage; developers in this stage spent about half their time in coding activities. In a second study [SLV97], developers were surveyed, observed, and interviewed to count the number of times they switched between one of fourteen activities (unrelated to those in [PSV94]). Observing each of eight developers for an hour revealed that developers most frequently executed UNIX commands, followed by reading the source, loading or running software, and reading or editing notes. Yet is not clear how activity switches translate to time spent on activities as activities may be frequent and brief or long and infrequent. A third study examined student developers working on a short, lab task and found that they spent 22% of their time reading code, 20% editing code, 16% navigating dependencies, 13% searching, and 13% testing [KAM05]. These studies demonstrate that coding activities are a significant part of software development and that understanding code – by searching, reading, and navigating – is a significant part of coding activities.

Several recent studies have characterized the information necessary during coding activities by listing questions developers ask. At the most general level, developers ask "why" (rationale), "how" (implementation), "what" (meaning of variables), "whether" (if code exhibits behavior), and "discrepancy" (observations do not match expectations) questions [L87].

Another study characterized developers' work through development activities, spanning interactions with code, artifacts, and teammates [KDV07]. When writing code, developers seek functionality to reuse and information about how to reuse it. Developers submitting a change ask if it is correct, whether it follows team conventions, and what changes it should include. Triaging a bug determines if it is a legitimate problem worth fixing. When receiving a new bug, developers reproduce it to determine what it looks like and when it occurs before asking about its cause. Developers also ask design questions about code's rationale and the implications of a change.

At a finer granularity, developers ask a variety of questions about code, ranging from very local questions to global questions [SMV08]. As developers begin, they first need to locate "focus" points in code, and attempt to map domain concepts to points in code. From focus points, developers ask about relationships between elements at focus points and other elements – e.g., types, methods, access statements, creation statements. More globally, developers reason about properties spanning a whole subgraph, including questions about how to do something or how something currently works. At the most global level, developers relate entire subgraphs together, reasoning about how disparate sections of the code are related. Many of these questions are about how to implement a desired change (e.g., where should this branch be inserted?) or what the implications of a change will be [SMV08].

An examination of what makes specific code exploration behaviors successful found several differences between unsuccessful and successful developers [RCM04]. Successful participants were more methodical, creating detailed plans before implementation and reinvestigating methods less frequently. Successful participants were also more likely to find information through keyword and cross-reference searches rather than browsing or scrolling. In contrast, lacking the information of the successful participants, unsuccessful participants made changes in one place that should have been scattered. Overall, the successful participants had more experience. However, the five participants in that study had only 1 to 5 years programming experience and limited (if any) industry experience.

As developers explore code, they read code and sometimes traverse into method calls. Several studies have examined how developers pick method calls to traverse. Information foraging, a general theory of how users explore an information space [PC99] that has been applied to developers exploring code [LBB08][LBB10], models this process as a traversal through a graph. At a node in the graph, users examine the *scent* of each edge by comparing its cues to the *prey* and picking the highest ranked. Using this theory to model code exploration, developers traverse paths through the call graph by reading method calls in a method, ranking each by it similarity to the prey, and traversing the best edge. For example, one study applied this model to code navigation during debugging tasks [LBB10]. Observing developers in the field, they found that their verbalizations were more often about scent-following than hypotheses; however, this may merely suggest more work is involved in testing hypotheses through foraging than in formulating hypotheses. Overall, the model successfully ranked the next method to which developers navigated in the top 13 (bug one) or 47 (bug two) results half the time. While the model is predictive – demonstrating that relationship traversal is an important part of debugging – the weakness of its predictions suggest that many factors influence where developers traverse and not all navigations may be traversals.

Several studies have shown that existing editors make navigating challenging. As developers explore code, they rapidly move through many methods, often digressing from the current question to answer related questions. This rapid exploration can leave developers *disoriented* or *lost*, unable to remember the context of their location and task. During observations of 10 professional developers at work on their everyday tasks in the field, 4 of the 10 mentioned that they were or were observed to be disoriented; 8 of the 10 reported that they had previously experienced disorientation [AM06]. Disorientation was largely attributed to a lack of context – as developers navigated through a flurry of files, they found it hard to remember how they were related. Observations of 7 experienced developers working on small lab study tasks found similar results [DKC05]. All developers agreed that finding the entry point and understanding the control flow were the most difficult tasks. And as developers moved through the code, they often had difficulty returning to and "re-finding" areas of code where they had already been. Developers often ended up with many open files, making it difficult to look through the open file tabs and identify the correct one to return to. Finally, a third study observed 10 developers in the lab and found that they spent 35% of their time navigating dependencies [KAM05]. As developers worked, they formed a *working set* of task-relevant methods, and then frequently revisited these methods as they worked.

2.2 Tools for exploring code

Many tools have investigated approaches for helping developers explore code. One popular, indirectly relevant approach, is for developers to write contracts. Developers might later be able to read the contracts, rather than explore a method's callers and callees. One variation on this technique is unit tests, in which developers attempt to understand the behavior of a method by writing tests. Other tools support code exploration more directly. Many of these tools help developers identify relevant methods, either by using task-context to recommend methods, letting the developer write queries, or helping the developer traverse relationships. Finally, many tools support code exploration through call graph views, including trees, graphs, sequences, and maps.

2.2.1 Contracts

Contracts let developers express constraints (e.g., pre- and post-conditions, invariants) on a program's state that must be satisfied at distinguished points in the CFG (e.g., method entry and exit) [H69][M92]. A program analysis traverses a statement graph, tracking state, and determines if the constraints are satisfied. If not, errors are reported to the user (e.g., compile errors). Instead of exploring callers or calles of a method, developers might instead be able to read the contract describing a callee or the preconditions that a caller must meet [LC06]. In this case, developers might never explore code at all and have no need for tools supporting code exploration.

A large number of verification and bug finding tools use the contract paradigm. Many tools statically traverse the CFG to check constraint satisfaction. For example, ESC/Java checks pre- and post-conditions written as JML annotations using a theorem prover [FLL02]. Other tools encode constraints as runtime assertions, generate concrete traces by running the program, and check for assertion violations. Contract verification allows developers to explicitly describe design intent – a contract – in an unambiguous notation for both other developers understanding the code and tools checking that the contract is satisfied. Instead of explicitly traversing control flow, developers implicitly state constraints about what must or must not happen on paths between constraint checks. Contracts can be particularly helpful for specifying interfaces between code produced by different teams or companies (e.g., frameworks) as it supports understanding foreign code without the code itself. For behavior specified in the interface, developers can answer downstream questions simply by reading the contract. Finally, when callers depend only on the characteristics of a method specified in its post-conditions, rather than all possible characteristics, developers may be able to change the method's implementation in ways that still satisfy the post-conditions without the need to understand the callers and ask upstream reachability questions.

But despite its benefits, contract verification is ill-suited for answering most reachability questions. Contracts rely on the original developer specifying in the contract behaviors about which it is permissible to ask a question. However, if a developer wishes to specify everything, contracts quickly become unscalable – at every method entry, a constraint on every effect that might possibly occur downstream would be necessary. Instead, contract-using developers are forced into a position of obliviousness to unspecified behavior. In practice, developers ask reachability questions about a wide range of behaviors. Moreover, even when developers ask a question specified in a contract, determining why a contract verification tool has reported an unsatisfied constraint requires determining the path taken by the tool [KFH08].

In property verification, developers specify a property using a specification language (e.g., a temporal logic) over program state that must be satisfied by all concrete traces (e.g., *a* always occurs before *b*). Property verification tools traverse feasible abstractions of concrete traces. Property verification generalizes contract verification by removing the modularity restriction that constraints may only be checked at distinguished points in the CFG. For example, the static driver verifier is able to check that drivers do not incorrectly use resources by checking that specified temporal orderings between method calls are respected [BNR03]. While property verification systems are specifically designed to check the reachability of error conditions, the specification languages are designed for the original developers to state complex correctness properties, not for investigating simpler reachability relationships in unfamiliar code. While property verification systems can be used to search for statements along feasible paths, they require the statements to be specified unambiguously, find only a single path reaching the specified statements (the error trace), and have primitive text displays listing any found paths. Thus, property specification is a poor interaction for investigating unfamiliar code where developers do not know the full names of what they seek, wish to see all paths matching search criteria, need to quickly iteratively refine search criteria, wish to filter or compare paths, or need to make sense of control flow relationships.

In unit testing, developers write a short program to generate a concrete trace and constraints expressed as assertions over execution state. Unit tests differ from contract testing in that constraint checks need not occur only at distinguished program points. In contrast to property verification, unit tests state constraints over paths downstream from the test rather than globally over all paths, and they constrain the functionality code provides rather than ensure the preservation of global invariants. Unit tests are widely used in practice, often through tools such as JUnit [J11] that automate running tests and viewing results. More recently, symbolic unit tests have been proposed which allow developers to add parameters to tests by performing constraint verification over feasible paths rather than a concrete trace [G07]. For answering reachability questions in unfamiliar code, however, unit tests suffer from all the same limitations as property verification.

2.2.2. Tools to identify elements

Applying the idea of automated recommendations (e.g., Amazon customers who bought *a* also bought *b* [LSY03]) to code investigation, recommender systems implicitly or explicitly determine artifacts in which a developer is interested and recommend other similar artifacts to investigate. For example, Suade uses call graph structure to recommend methods to investigate based on code elements a developer indicates are relevant [R08]. Recommender systems assume that there exist delocalized concern elements in code – methods or types implementing a feature provided by the application – and that the goal of developers' investigative activity is to navigate from discovered elements to the remaining hidden elements. In general, however, reachability questions need not be about relationships between parts of a concern but may be about how loosely related portions of code interact. Moreover, only considering as input the elements a developer has already found yields insufficient information to determine what question has been asked. But while recommender systems are ill suited for answering reachability questions, algorithms for inferring relevance from call graph structure could be applied to ranking the relevance of reachability question search results.

A variety of tools extend traditional IDE search tools' support for finding methods by adding expressiveness. CodeQuest lets developers formulate queries in datalog against a database of relationships in the program [HM06]. Developers can, for example, find subtypes of a type or combine searches together to find subtypes of a type defining a method matching some name. Similarly, Ferret builds a database of elements and relationships spanning static, dynamic, and version control information [DM08]. Developers can use Ferret to answer questions like "What are casts to this type?" or "Who has changed this element, and when?" Other work extends a Ferret-style database with a structured natural language front-end, letting developers query the database using English and have their query translated [WGR10]. All of these tools focus on structural relationships between elements of code and lack the expressiveness to follow paths through code.

2.2.3 Exploring call graphs

A number of tools and techniques have investigated approaches for helping developers explore call graphs, which are reviewed in this section. For example, a number of documentation approaches let developers visualize a specific part of the call graph. Other tools let developers traverse relationships in code or provide a tree view depicting paths through code. Traditional call graph visualizations tried to depict the entire call graph in a graph visualization. More recent approaches have investigated using a map metaphor, embedding source editors inside methods in a call graph. Finally, several tools let developers see paths through code, often using a UML sequence diagram notation.

When the original developer recognizes code as important and complex, a developer may document relevant traces. Structured design includes diagrams for depicting control and data flow relationships [YC79], and UML sequence diagrams depict concrete traces [RJB99]. Documenting concrete traces has the advantage of recording the original developers' intent by their choice of elements to depict or text accompanying the diagram. But many developers do not invest the time to write documentation or do not reliably update the documentation, making it suspect (see Chapter 4). Views of code that are not reverse engineered or checked for conformance against the code always run the risk of being inaccurate. This may be especially true of path documentation, as paths are likely to change with even minor code edits. But most importantly, the huge number of paths through a program makes documenting all of them impractical.

Several tools help developers more easily traverse paths through code. Modern development environments such as Eclipse and Visual Studio provide a call hierarchy that lets developers hierarchically expand the callers or callees of a method. JQuery (the code exploration Eclipse plugin [JV03], not the more recent Javascript library) extends the call hierarchy to also include additional elements and relationships between these elements (e.g., method membership, subtyping, containment, references, or constructors of a type).

Other tools help developers explore code by visualizing the complete call graph as a graph. For example, Shrimp provides an extensible framework for visualizing graph structures during reverse engineering tasks [SM95]. It provides interactive tools, such as fisheye views, for exploring graphs. Software cartography uses a map metaphor to show classes, representing each as a hill [KEN10][KEL10]. Height encodes lines of code. Integration with the development environment highlights the open file on the map and search results and shows call graph paths from the call hierarchy. Both of these tools provide no support for searching along paths and do not encode attributes of calls (e.g., ordering, conditionals, loops). Moreover, the focus is on showing class relationships, with calls overlaid on them, rather than helping developers interactively find and visualize a task-relevant portion of the call graph.

Several systems have explored approaches for reducing disorientation during code exploration. Relo [SMK06], Code Canvas [DVR10], and Code Bubbles [BRZ10] help developers to stay oriented by providing a map of code. Replacing a conventional editor in which developers edit in a full size window, methods are instead shown in many small bubbles, providing context during reading and making it easier to rapidly switch among related methods. Like these tools, REACHER's visualization helps to minimize disorientation by letting developers select task-relevant methods and visualize relationships among these methods. One important difference is that REACHER shows only method names and task relevant statements rather than the entire method's implementation. This makes REACHER'S visualization substantially more compact, allowing developers to simultaneously view many more methods. REACHER'S design may more effectively support situations in which developers investigate relationships between small snippets scattered across many methods. Moreover, both visualization styles could be incorporated in the same system by letting developers zoom in to see a method's implementation and zoom out to see additional context.

Several tools reverse engineer UML sequence diagrams from dynamic traces. A UML sequence diagram contains vertical lifelines for several objects, shows calls as horizontal lines between lifelines, orders calls vertically, and includes annotation notation for describing calls (e.g., calls can be guarded by a conditional expression text) [RJB99]. Execution murals reverse engineer a simplified UML sequence diagram view from dynamic traces [JSB97]. Similarly, Diver is an Eclipse plugin that records execution traces and provides a UML sequence diagram view [MS10].

While most code exploration tools do not let developers search along control flow, a few do. Logging aspects allow developers to construct a search string as a pointcut descriptor, run the program, and browse matching target statements written to a log file [KHH01]. But developers must rerun the program whenever they change their search string and there is no support for exploring the log. In Dora [HPV07], developers select an origin method and enter a search string, and then may inspect a graph depicting call graph paths to methods textually similar to the search string. However, using Dora to answer reachability questions would be challenging. It does not support searching for field reads, field writes, library calls, or methods in specific types or packages, making it impossible to directly express most of the reachability questions we observed in our field research. And Dora provides only a rudimentary call graph view. Dora's focus is instead on exploring the use of information-retrieval techniques in searches and is therefore complimentary to REACHER. Diver provides limited support for searching along dynamic traces [MS10]. Diver lets developers search along an execution trace for method calls and visualizes traces as UML sequence diagrams. But, in Diver, searches are used only to locate methods, not to scope the visualization to the search results. In situations where dynamic analysis is possible and helpful, dynamic traces could complement REACHER's static traces by providing certainty of a path's feasibility and supporting inspection of concrete values.

Several studies have observed developers using existing tools for exploring code to produce recommendations for future tools that would more effectively support developers' needs. One study observed developers using a UML tool while editing code [DAB08]. In addition to identifying several usability problems, a key recommendation was to better support selecting task-relevant items in the reverse engineered view to prevent wasted time understanding task-irrelevant items. They also saw the need for much more automated support for reverse engineering sequence diagrams. Another study failed to find much use of detailed large-scale maps of code hung on walls near developers' offices [CVD07]. Designed to be useful for all possible tasks, these diagrams had both too much and too little information – developers required many details but only those that were task-relevant. The authors conclude that diagrams providing concise and targeted answers to situation-relevant questions were more likely to be useful than general-purpose diagrams. Another study observed several students using a UML sequence diagram tool in the lab [BMS08]. Qualitative analysis suggested element labels, animation between layouts, and diagram-to-source linkages are all important for such tools. Participants specifically requested the ability to rapidly configure the diagram to filter or search for items and to easily hide items that were determined to be uninteresting. Finally, the authors suggest that the frequent navigation between corresponding portions of the diagram and source could be reduced by adding additional information to the diagram such as information about conditionals and loops.

Few code exploration tools have been evaluated in user studies to see if they do indeed help developers to explore code. There are two notable exceptions. An evaluation of Whyline found that it helped developers debug 3 times more successfully in about half the time [KM09]. One study evaluated three recent code exploration tools (JQuery, Ferret, and Suade) by having developers perform two small changes in an unfamiliar codebase (jEdit) [AMR07]. A variety of measures were used to test several hypothesized benefits – reducing mental resources used, helping developers avoid irrelevant elements, and more successfully completing the task. However, there were no significant effects on any of these measures, suggesting these tools may not provide any benefit to developers.

2.2.4. Slicing

Slicers follow control and data dependencies through code [HH01][SFB07][T95][W84]. Slicers find statements connected by either data dependencies or control dependencies.

Reaching definitions *reachingDefs*(s_1, *x*) finds the set of statements *S* where each s_2 in *S* may have last defined a variable *x* used in s_1 (the set of immediate data dependencies). A *control dependency* exists from s_1 to s_2 if s_2 controls if s_1 does or does not execute. *cDepend*(s_1) finds all such control dependencies of s_1. A *(backward) static slice* [W84] is simply the transitive closure of the union of these two relations: $(reachingDefs(s_1, x) \cup cDepend(s_1))^*$. In a highly influential study, Weiser found that developers debugging more accurately remembered a static slice related to the bug than either an unrelated slice or an arbitrary portion of the program [W82]. This suggested that developers follow slices when using the strategy of debugging backwards from an error to a bug.

Traditional slicers allowed developers to find the portion of a program that is control or data dependent on a seed statement. More recent tools let developers traverse dependency paths. For example, CodeSurfer statically computes slices and lets developers traverse through them in code [AT01][ART03].

Numerous variants of slicing have been proposed and implemented as research prototypes [HH01][JR94][T95]. A *forward slice* finds control and data dependencies forwards rather than backwards: $(fdDepend(s_1, x) \cup fcDepend(s))^*$. A *dynamic slice* finds control and data dependencies in a particular execution. For example, the Whyline computes a dynamic slice by having the developer demonstrate a situation in a program, recording a trace, and providing interactive support for traversing the dynamic slice [KM08]. Like reachability questions with a filtering constraint, *conditioned static slices* [CCD98] find dependencies across paths which satisfy a constraint. A *thin slice* [SFB07] finds data dependencies $reachingDefs(s_1, x)^*$ while excluding data dependencies at pointer dereferences. A *chop* [JR94] intersects statements in a forward slice on *x* at s_1 with a backward slice on *y* at s_2: $(fdDepend(s_1, x) \cup fcDepend(s_1))^* \cap (reachingDefs(s_2, y) \cup cDepend(s_2))^*$. The central idea of all slicing techniques applied to code exploration is to use control and data dependencies to find statements answering a developer's question.

An important difference between reachability questions and slicing is that most reachability questions are a search across control flow paths rather than data dependencies (except for reachability questions that use *reachingDefs*(s_1, *x*)). By design, slices are intended to be a subset of the statements across control flow paths: statements that are not dependent are not included (although an imprecise slicing algorithm may still include them). Slices correspond to questions about influence: "Why did this execute?" (control dependency), or "Where did this value come from?" (data dependency). In contrast, control flow captures questions about what happens before ("What are the situations in which?") or after ("What does this do?"). When developers ask a question about control flow, the slice may not include the statements answering their question. And while our reachability question formalism includes searches for data dependencies (*reachingDefs*), we observed only 1 example of such a question out of the 17 important reachability questions we found (see Tables 7.1 and 7.2).

The most important difference between slicing and reachability questions is that a reachability question is a search for a set of statements described by any of a wide variety of search criteria. Consider an example from the Exploration Lab Study (see Section 7.1): a developer wondered why calling a method *m* is necessary. The reachability question *find ends in traces(jEdit, m,* m_{end} , ?) identifies a few statements (5 at a call depth of 5 or less from m_{start}) while a static slice from *m* finds all of the statements in hundreds of methods. Because the first line of *m* conditionally throws an exception depending on the input to *m*, everything afterwards is control dependent on the input to *m*. If this were not the case, the static slice still would not help locate *ends* and might not even include these statements if they do not happen to be control or data dependent. Even the searches supported by chopping are different: in chopping, both the origin and target statement are supplied by the user. Thus, the user must already know the statements in *ends* when they ask a *chop* question.

2.3 Path-sensitive static analysis

REACHER draws heavily from path-sensitive static analyses for bug detection and verification. Broadly, such tools traverse feasible paths through a program, update abstract state by inspecting statements, and output an error whenever an error state is encountered. Different tools strike different performance / precision tradeoffs for the bugs they seek to discover.

A symbolic execution of a program symbolically propagates input variables across paths through a program [K76]. Instead of executing a program on concrete values (e.g., *5*), the program is executed with names of input variables (e.g., *x*) and constraints on these variables (e.g., $x > 5$). At conditional statements, a symbolic execution first attempts to determine which branch is feasible. If this is not possible, multiple paths are forked off, constraints are added to variables on each path indicating which branch was taken, and each path is explored in a separate context. No merging of contexts is performed at control flow merges. In this way, a symbolic execution constructs an execution tree of potential paths a program might follow. However, the precision of a symbolic execution is limited by the ability to determine which paths through a conditional are feasible. Moreover, the number of paths through real programs are intractably large, so tools are often unable to exhaustively explore all paths and are instead forced to sample paths.

The most precise approach to determining feasible paths is CEGAR (counter-example guided abstraction refinement) model checking (c.f., SLAM [BR01]). In contrast to symbolic execution, CEGAR model checkers lazily add precision by only adding constraints to variables when necessary. These constraints are used to guide which paths are taken at conditionals. After finding any feasible or infeasible path to an error statement, a theorem prover or SAT (satisifiability) solver is used to determine if the path is feasible. If the path is infeasible, constraints are added to variables to prevent this path from being traversed, and the model checker again begins searching for a path to an error statement. While CEGAR model checkers have been used in practice, both the use of the theorem prover and iteratively searching for paths can result in runtimes of hours or days for even small programs. And CEGAR model checkers have difficulty dealing with paths that are data and pointer intensive.

Dataflow analyses are less precise than model checkers but take much less time to execute. Most dataflow analyses do not attempt to eliminate infeasible paths. Instead, dataflow analyses simply iteratively traverse paths through a program to populate a context mapping variables in scope at each statement in the program to abstract values – constraints designed by the analysis author. The key feature of a dataflow analysis is that cycles in paths due to loops or recursion are iteratively traversed until a fixed point is reached and none of the abstract values have changed on the final iteration. Interprocedural dataflow analyses are generally considerably more scalable than model checkers and can run on even large programs in minutes or hours. A path-sensitive dataflow analysis is sometimes able to determine which paths through a conditional are feasible by using information in the context. When this is not possible, these analyses traverse both paths using separate contexts. Such a fully path-sensitive analysis is impractical, as the number of contexts grows exponentially in the number of conditionals that cannot be resolved. Instead, practical tools (c.f., ESP [DLS02]) join contexts with identical abstract state at control flow merges.

Call graph construction algorithms eliminate infeasible paths created by dynamic dispatch or first class functions [GC01]. By propagating information about the possible runtime types of objects, these algorithms eliminate infeasible paths by determining the possible runtime types that might reach each receiver object or function pointer. These algorithms are fast and are often used as an input to other dataflow analyses, but only eliminate infeasible paths arising from dynamic dispatch. Moreover, these tools are not path-sensitive and do not propagate constants other than types.

In summary, most existing techniques for eliminating infeasible paths are either slow (model checking, symbolic execution, fully path-sensitive dataflow analysis), do no eliminate infeasible paths (path-insensitive dataflow analysis), or eliminate a much more restricted set of infeasible paths than REACHER (call graph creation algorithms). The closest approach is partially path-sensitive dataflow analysis which both eliminates many of the same infeasible paths and is also fast. However, the benefits of this approach rely on merging contexts with identical abstract state. When only a simple property is being checked, there will be few potential distinct contexts possible. If this approach were to be applied to propagating constants to determine path feasibility, the number of contexts would be exponential in the number of variables that might have a constant value. Thus, this approach is also too slow.

2.4. Summary

Existing studies have shown some of the outlines of how developers explore code, but many gaps remain: how is code exploration related to software development activities; what, exactly, are developers doing while exploring code; and what hard to answer questions are related to code exploration? How can a tool most effectively support code exploration? While studies suggest that developers could benefit greatly from diagrams that are more task-relevant, the studies provide little guidance on the information developers need to find when exploring code.

While many approaches have been tried to help developers explore code, there is scant evidence that any of those tools are useful. Indeed, the largest comparative study of code exploration tools found that none had any discernable effect. And none of the tools let developers search along control flow and navigate a compact, task-focused view of call graphs.

Finally, there is a large body of work in path-sensitive static analysis, but none is able to eliminate infeasible paths quickly enough to be used in code exploration tools.

3.

DESIGNING USEFUL TOOLS FOR DEVELOPERS[1]

Is a development tool useful? This question ultimately asks how the tool affects developers' work. This is not, as others argue, a question of philosophy, mathematics, or esthetics [B10], but of science: if a developer adopts a tool, is his or her work faster or better? Claims about a tool's usefulness are falsifiable statements about the real world and thus scientific. For the purposes of this chapter, the word "tools" includes anything used by a developer to develop software, ranging from development environment plugins to online documentation to type systems to programming languages.

Unfortunately, usefulness is challenging to measure. After designing and implementing a tool's core features, more work may be required before developers can or will adopt the tool. This may involve adding features, fixing usability problems, or even building a user interface. This work is traditionally viewed as an engineering effort with little research value, but generally must be completed before the tool's usefulness can be measured.

The most direct measurement of usefulness is through a field deployment. But measuring usefulness in the field requires controlling a myriad of confounding factors. If the developers who adopted a tool fix bugs 10% faster than they did last week, was this effect caused by the tool or by easier bugs, new debugging strategies, or more code knowledge? Even when confounds have been controlled, skeptics may still ask if the result generalizes to developers with different skills, in different domains, with different processes, or with different existing tools. While not impossible (c.f., [ABG02][JC07]), field evaluations are no small undertaking.

Thus usefulness is often not directly evaluated. Instead, researchers often evaluate a tool's usability – its ease of use – in a lab study. Developers, or (more typically) students with development experience, are brought in and asked to complete tasks while using a new tool and their performance is compared to others using a comparable existing tool. Developers' performance is measured by recording time and success. Performance with those using the new tool and the control is then compared. If differences between conditions are statistically significant, the results provide evidence of the tool's usability.

[1] This chapter based on work previously published in [LM11].

But such a study does not, by itself, demonstrate usefulness. First, does it generalize? Is the result specific to the situation studied: how might it change with different tasks, codebases, or expertise? Second, what does it mean for developers in the field? How frequently do developers do tasks equivalent to those in the study? Do limitations of the tool prevent it from being used in less controlled settings? Third, how did learning a new tool influence the result? If the results showed the tool did not help, was this because the tool is not useful or because developers had not yet learned new strategies or processes [B02].

Due to these challenges, tool designers often evaluate usefulness less empirically but with a motivating example. Motivating examples demonstrate a tool on an example task and can be used to claim that while existing tools make the task time-consuming, tedious, or error-prone, the tool solves these problems (e.g., statically prevents null pointer exceptions, reduces boilerplate code). But while motivating examples can be highly effective for explaining a tool's features and usage, they do not show that developers do these tasks, that developers do them as described, that developers have the assumed problems, that developers would use the tool in the way described, or that developers would be more productive if they do so. Motivating examples might explain the mechanism by which a tool's designers hypothesize it would change developers' work and make them more productive, but do not provide evidence that this occurs.

This chapter describes a process for using data to design useful tools for developers. Data is used before, during, and after design to understand developer's work and how it is affected by a tool. Exploratory studies generate data to identify and describe a problem. Tools are designed to address specific problems, with lightweight evaluation studies testing early design ideas before a large commitment has been made. Evaluation studies help both quantify performance effects and to understand how these effects occurred, allowing greater generalizability. While this design process is heavily influenced by contextual design [BH97], this chapter explains how it can be adapted to designing tools for developers.

This chapter begins by examining the structure of software development work and how tools may support this work. The design process is then traced from beginning to end, describing techniques for understanding problems, designing a solution, and evaluating its effects. Chapters 4 – 10 show how I used this approach in the development of REACHER.

3.1. Supporting development work

Useful tools support software development work. But what is "work," and how do tools support it? Software development work can be hierarchically decomposed into tasks through task analysis (see Figure 3.1). In each task at each level in the decomposition, developers have a goal, either a question to answer or something to accomplish. At the highest level, tasks reflect *activities* – e.g., fixing bugs, implementing features, refactoring. These can be decomposed into *sub-activities* – e.g., debugging, editing code, reusing code, understanding code. At a lower level, developers formulate a specific plan for accomplishing a goal as a strategy – a sequence of steps. And, steps in a strategy may themselves involve using other strategies.

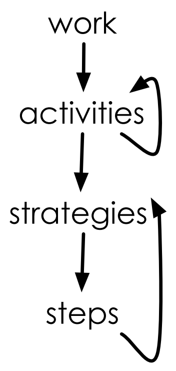


Figure 3.1. Developers' work is hierarchically composed of tasks that may be activities, strategies, or steps (arrows indicate a task is composed of one more other tasks). All tasks have a goal and may also have associated problems.

Tasks are driven by a goal. For many of developers' activities, the goal is to answer a question. Developers experience problems answering a question when strategies to answer it are time consuming, error-prone, or tedious with the available tools [AAL10][B95][BPZ10] [AMR07][FKS08][KKI02][K08][KM08][KM09][KAM05][KDV07][LBB10][PO11][SMV08] [SC07]. Debugging, determining how to reuse an API, or predicting the implications of a change are all examples of problems that require answering questions. Tools support answering questions by helping developers answer them more quickly or accurately, often by automatically producing information. Debuggers, type systems, reverse engineering and understanding tools, defect detectors, and protocol miners all provide information intended to help answer questions. For the designer, the key challenges are to determine exactly what question developers ask, what information is helpful to answer it, and how developers can obtain this information more easily.

Other activities involve accomplishing something. Here, developers seek to change an artifact in some fashion. Like answering questions, strategies can again be problematic when they are time-consuming, error-prone, or tedious.

Tools' support of a developers work can be viewed as a theory describing its usefulness. Such a theory postulates a way in which developers work in some situation and the way that this work is affected by the use of a tool.

Tools support work by making a strategy faster or more successful. For example, the Whyline [KM08] helps developers debug (an activity), which involves answering why and why not questions about the causes of erroneous output (questions). Rather than formulate and test hypotheses about a bug's cause (a frequently unsuccessful strategy), the Whyline lets developers directly select output and follow dynamic slices explaining why it did or did not occur (a new strategy made possible by the Whyline). Studies of the Whyline provided evidence of its usefulness by demonstrating that developers frequently ask why and why not questions during debugging, that many of the hypotheses developers formulate are wrong [K08], that developers can use the Whyline to answer their why and why not questions, and that the Whyline helps developers work quantitatively more effectively [KM09]. Together, these studies provide a theory describing how the Whyline supports work.

3.2. Understanding a problem

Useful tools solve an *important* problem. Problems may be important for many reasons, but often not the ones researchers expect. Exploratory studies can help to identify and understand a problem's true cause and importance.

3.2.1. Important problems

The importance of a problem can be characterized along three dimensions: frequency, duration, and quality impact. Problems vary along all these dimensions and come in many shapes and sizes. Problems need not be frequent and long to be important: an hour every week may have the same direct impact on productivity as 30 seconds 120 times a week. An autocomplete tool which frequently saves a second of time might have the same impact as a specification checker preventing a very hard to debug but infrequently occurring defect. Tools may also indirectly impact productivity; perhaps the autocomplete tool helps keep the developer more focused on the task, reducing time that would be spent switching among tasks. Problems that are neither frequent nor long in duration can be important if they substantially impact quality.

Useful tools must solve an important problem in order to justify their adoption cost. Adopting a tool imposes costs to install it and to learn how to use it effectively, and there is always the risk that it will not actually help [B02]. One reason practicing developers are skeptical of academic tools is that their perceived benefits are too small [H11]. Therefore, determining which problems are sufficiently important to matter, so the tools addressing those problems will be perceived to be beneficial, is an important function of exploratory studies.

Solving a problem requires understanding its true cause, which often requires digging into symptoms. Consider the problem of code duplication. Developers have long been accused of copy and paste reuse – reusing short snippets of code by copying and editing rather than refactoring code into new abstractions. Copy and paste reuse creates clones, which are frequent in open source codebases (c.f., [B95]). Believing the problem to be one of awareness and detection of clones, researchers designed tools to automatically detect short copy and paste clones, hoping developers made more aware of copy and past clones would refactor them [KKI02]. But other researchers believe that awareness is not the problem, and that code duplication is instead caused by expressiveness – existing languages make refactoring clones to abstractions mentally challenging, introduce unnecessary overhead, make code more complex, and may not support all types of clones [TBG04]. Other studies suggest that much of code duplication may not be caused by copy and paste reuse at all, as it is but one of many causes including forking codebases for organizational reasons and maintaining old versions (see Chapter 4). For example, developers sometimes work with code where entire codebases have been duplicated and modified, in order to maintain different versions, configurations, or releases. As a result, commercial clone detectors have found more success focusing not on copy and paste reuse but on maintaining multiple versions. Fixing a bug in these situations requires an extra step: find the equivalent code in all of the versions. Missing a bug in one of the versions can be a serious problem: nearly half of software releases contain a security vulnerability already fixed elsewhere in the codebase [P11]. Pattern Insight's clone detector is primarily used to solve this problem [P11]. Common wisdom can suggest interesting aspects of software development – code duplication – but initial solutions – detecting copy and paste reuse – may not solve the problem without understanding its true cause – fixing bugs in multiple versions.

One of the most successful approaches to identifying and understanding a problem is to identify problems with using a strategy or answering a question (c.f., [AAL10][FKS08][KAM05][KDV07][SMV08][SC07]; Chapter 7). Problems at this level can be directly addressed by tools. For example, one study examined how developers choose a class in an API to accomplish a goal [SC07]. Developers pick a candidate, try it out by instantiating it, and often get compiler errors prompting them to supply a required parameter. Developers then investigate how to correctly construct the class, only to later discover that the class is missing the methods they need. The compiler errors encourage a *premature commitment* [G89], causing investigation into something that may be irrelevant. Understanding this strategy (how developers pick classes in an unfamiliar API) helps to identify the problems that make it hard and time consuming (compiler errors encourage premature commitment). Tools can then be designed to address these problems (preventing premature commitment) and evaluated in terms of their success in doing so.

3.2.2. Choosing an exploratory study

Exploratory studies help to identify important problems and understand their cause. Exploratory studies may gather data both through developers' perceptions of problems and through directly examining the problems themselves. Developer perception can be an important source of ideas, particularly for poorly understood tasks. Developer may identify challenging tasks or hard-to-answer questions. But other problems are less salient, and may be important without developers realizing it. For example, developers may not notice how much time they spend scrolling to revisit code [KAM05].

One of the easiest exploratory studies to conduct is an interview. Interviewing developers can reveal a developers' typical tasks and problems (c.f., [HH11], Chapter 4). But recollection of the past is imperfect: people give vague descriptions and generalize [BH97]. While generalizations may suggest ideas, they may not be based on facts and can be biased by opinion and perception. But, for tasks or situations for which little is known, interviews give a sense of the basics and help to focus further study on interesting aspects.

Contextual inquiries augment interviews with direct observations, using a real, in-the-moment tasks to provide context [BH97]. Contextual inquiries replace generalization with examples; an experimenter watches the developers as they work and asks questions about the task at hand. Developers work on a representative task and think aloud as they work. When the developer's goals, questions, or strategy is unclear, the experimenter asks for clarification. When the developer generalizes, the experimenter asks for a concrete example. When frequently interrupting is inconvenient or obtaining accurate timing of steps and activities is important, direct observations can be used alone, without an embedded interview. The experimenter may still briefly interrupt, but interruptions are focused on brief clarifications rather than extended discussions. Direct observations can be conducted both as field studies (c.f., [KDV07][SMV08]; Chapter 7), watching developers in the field do their everyday work, or in lab studies which permit choice of the task and comparisons of developers doing the same work (c.f., [AAL10][FKS08][KAM05][SC07]).

Direct observations lead to generalization by analyzing the data afterwards. Depending on the aspect or situation of interest, many types of analysis are possible. Simply reporting observations is sometimes sufficient. But often it helps to examine their generality by looking for patterns, often through content analysis [P02]. Taxonomies investigate what things exist in the data – e.g., types of activities, questions, and strategies.

3.2.3. Understanding context and frequency

How well a strategy works often depends on the context [LM10-3]. Consider answering the question "Does this method repaint the screen?" One strategy which developers use is to determine, through code inspection and by traversing paths through the code, if the code calls repaint. Information foraging predicts that developers use their knowledge to pick which edge to traverse based on its similarity to the goal [LBB10]. This is sometimes easy. But when there are many edges to choose from, longer paths to follow, or identifiers that are misleading, this strategy is likely to take longer or fail. Other factors that influence its difficulty include characteristics of the code and the developer. For example, developers less knowledgeable about the code may have a harder time predicting which identifiers are most related to what they are trying to find. Learning about factors influencing a strategy's success is an important part of understanding a problem, and helps ensure that solutions can be targeted to the most important situations.

While observations and interviews help to understand questions, strategies, and problems, data about frequency is limited by the few situations observed. Frequency can be measured through studies designed to sample many developers such as surveys or indirect observations. Surveys gather frequency data by asking many developers questions (c.f., [BPZ10] [KDV07]; Chapters 4, 6, and 7). Surveys can also be used to understand correlations. For example, one of my studies found that a class of questions becomes neither less frequent nor easier to answer as developers become more experienced or spend more time in a codebase (see Section 7.2). Indirect observations gather data about developer's work not by directly seeing it but by capturing summary data, such as with logging, or by studying artifacts created by work such as code, code change logs, emails, bug discussions, and forum posts (c.f., [BKA11][KMC06][MPB09]). For example, one study measured the prevalence of protocols through an automated technique for detecting protocols in code [BKA11], built a taxonomy of protocol types, and examined their typical complexity.

3.3. Designing a solution

Designing a tool begins with an important problem to solve. A problem is the beginning of a solution – it identifies a specific aspect of work to improve. Designing a solution envisions a new way of doing this work and determines the features necessary to make this possible. Designing a tool is a leap from an old to a new way of working; designing a tool that solves a problem is an inherently creative process that no amount of data can guarantee. But data can help to understand what a design must achieve to solve a problem, and can help to understand if the design is likely to succeed.

When using a tool to answer a question, a developer must translate a high-level question (e.g., what caused this bug?) into lower level questions the tool supports (e.g., using a breakpoint to answer: "What is the value of this expression when this code executes?"). For the designer, the key challenge is ensuring that the information that the tool provides really helps to answer the high-level question more quickly or accurately than the alternatives. One way to bridge this gulf is to understand how developers currently work: what strategies do they use to answer high-level questions, and what lower-level questions do these strategies entail? For example, I found that developers sometimes answer questions about the implications of a change (e.g., what it might break – see Chapter 5) by searching along control flow for things that the code does (see Chapter 7). Helping developers search along control flow is likely to help developers answer higher-level implication questions, as developers are already using this strategy. But supporting developers' current strategy is not the only approach – tools could instead provide an entirely new strategy that is impossible with existing tools. But it is then necessary to determine if the low-level questions the tool answers actually help answer higher-level questions, whether these low-level questions are really an important part of the problem, and whether the developers will think to use the strategy the tool supports in the relevant situations.

Consider the following example: many automated debugging tools attempt to predict the faulty statement that caused the bug and provide the developer a ranked list of candidate statements [BNR03][CZ00][GKL04][JHS02]. These tools change how developers answer "What caused this bug?" by letting developers inspect the list of statements and answer the question "Which statement contains the fault?" But how big a part is finding the faulty statement in determining the cause of a bug? Is seeing a statement sufficient for a developer to determine that it is faulty? A user study investigated these questions by comparing automated to conventional debugging tools [PO11]. It suggested the answer is no: most developers spend an average of 10 minutes inspecting each statement to understand how it might have caused the bug. As a result, the automated debugging tools only helped a fraction of developers debug one of two tasks more quickly. Understanding exactly what information a tool should provide to help answer a high-level question is crucial to a tool's success.

3.4. Evaluating a solution

A tool is the embodiment of a tool designer's assumptions about how developers currently work and the way in which that work may be more effectively supported. Unfortunately for the designer, these assumptions may be wrong. This risk can be minimized by getting feedback early in the design process through *prototypes* and *lightweight evaluation studies*. In a *paper prototype study* [B07], users interact with screenshots, narrating which buttons they would click, while the experimenter manually simulates the tool by showing the next screen (see Section 10.1 for a pilot paper prototype study of REACHER). Higher fidelity mockups are also possible. In a Wizard of Oz study [MGM93], an interface is built, but the implementation is remote-controlled by the experimenter. For example, a bug detector might provide error messages that seem to be automatically generated when they are actually triggered by the experimenter. Such a study allows the effects of different error message designs to be evaluated before determining how such errors will be generated. Lightweight evaluation studies enable an iterative design process which is tailored to what works rather than what the designer assumes will work.

The usefulness of a tool depends on its success in solving a problem by supporting work (mechanism) and the importance of the problem it solves. Lab studies are most effective for understanding a tool's mechanism. Do developers use the tool to answer questions? Does the tool help them do it more quickly or successfully than before? What strategy(ies) does it support? On what aspects of the situation does the strategy depend? How might these aspects affect how developers use the tool? Did developers enjoy using the tool? Evaluating a tool's usefulness is difficult without understanding *how* it supports work.

Consider an example: one study investigated the productivity effects of dynamic typing [H09]. Participants took anywhere from 4% to 42% less time using a dynamically typed language compared to an otherwise identical statically typed language. But does this result generalize? Did the tasks involve any situations in which static typing might be expected to provide useful feedback? If so, how did developers use this information and why did it not help? Was it simply that the error messages provided were poorly designed and unhelpful? What were developers doing during the additional time in the statically typed condition? Would developers working with different codebase or with different tools still have these problems? Are there ways to provide useful static typing feedback without incurring the productivity costs? Answering these questions requires a deeper understanding of developers' tasks, activities, questions, and strategies.

Lab studies can also quantitatively measure a tool's effect on task time and success. A result which shows an effect is best viewed as an existence proof: it is possible to achieve significant productivity benefits. Such results are an important demonstration that a tool can have a strong effect (or that it did not) (c.f., [AMR07][H09][KM09][PO11]). Yet skeptics can always argue that, for a slightly different task or situation, the tool's benefits may vanish. Studying a tool in more situations helps to address these concerns. But the strongest argument is also based on mechanism – an explanation of *how* a tool changes the way that developers work. Mechanisms predict the effects of a tool in unobserved circumstances. These predictions may be wrong, but even then, they focus research on what prevented the predicted effects from occurring. Did developers not do the expected tasks, use unexpected strategies, or did they experience unexpected problems with these strategies?

Another benefit of understanding mechanism is in designing lab study evaluations. Lab studies sample particular situations by picking tasks, participants, and materials. Which of the many possible situations are most interesting to study? Situations where developers do not do the task supported by the tool are not interesting, as the results only show the tool is not relevant. However, observing tasks for which the benefits are unclear is interesting. Situations where the tool is expected to have a strong effect provide evidence that the effect actually exists.

3.5. Conclusions

Designing a useful tool requires more than finding a compelling motivating example, evaluating the tool's technical merits, and performing a carefully designed user study. Designing a useful tool requires understanding how a tool supports work and addresses an important problem that developers face. This understanding is built over time through hypotheses and studies, investigations of the problem and potential solutions, and through direct and indirect observation. Understanding involves identifying both the *mechanism* of how a tool supports tasks, questions, and strategies and the *frequency* that these occur. And evaluations must examine not only whether some quantitative productivity effect is possible but how, when, and why this effect is achieved.

This chapter has considered theories at the level of specific situations and their relationship to specific types of tools. But as more theories are formulated, they may have much in common, paving the way for larger theories of developers' work. These might permit predictions of developer behavior and the effects of tools over whole tasks and activities, leading to a science of software development.

4.

CONTEXT OF CODE EXPLORATION[2]

While software engineering researchers have long viewed exploring and understanding code as important and have designed tools to support it (see sections 2.1 and 2.2), less is known about the role and context of understanding code in software development. What activities do developers spend their time on, and how is understanding code related to these activities? What tools and techniques do developers use, and how do these differ for similar activities such as designing or editing code? What practices influence how developers understand code? What role do design documents play in understanding code? When do developers choose to use tools to understand code rather than consult with their teammates? What problems do developers face understanding code, and how are these related to other software development problems?

This chapter describes a field investigation into how developers understand code. The investigation began with a survey, used interviews to explain findings in this survey, and then used a second survey to investigate the generality of these findings. The central theme that emerged from this investigation is developers' great reliance on implicit knowledge about code. Developers invest great amounts of time creating and maintaining a mental model of the code. As developers do this, they share knowledge with their teammates through face-to-face communication and through the code itself. Developers avoid, when possible, explicit, written repositories of code-related knowledge in design documents or email, preferring to explore the code directly and, when that fails, talk with their teammates. Exploring code is made difficult by tool limitations and difficulties in traversing relationships. Using the social network as the second line of inquiry causes interruptions and lost work, but those costs are offset by other benefits. Knowledge retention is made possible by a strong, yet often implicit, sense of code ownership, the practice of a developer or team being responsible for fixing bugs and writing new features in a well-defined section of code. This increases the payoff from the large investment in understanding code.

In this chapter, we first describe a taxonomy of developer activity we used to begin our investigation. Next, we describe the methods we used in conducting two surveys and a number of interviews with professional software developers in the field. We then describe our results about the frequency of developer activities and the tools developers use. Next, we discuss our findings about the context of understanding code and the role of software development practices in influencing how developers maintain mental models of code. One

[2] This chapter based on work previously published in [LVD06].

particularly challenging aspect of code to understand is its rationale, which we investigated in more detail. Finally, we report our findings on developers' openness to changing their practices, make recommendations for addressing several of the problems we identified, and conclude.

4.1. Activity taxonomy

This investigation began with a characterization of how developers interact with code – their activities, tools, and biggest problems. Rather than bring a preconceived area of focus, we wished to be more opportunistic and let our users – the developers – guide us in selecting what they perceived to be the most painful problems. To do so, we asked them to report their use of time, perceptions of tool effectiveness, and most serious problems.

In contrast to previous studies (see the reviews of [PSV94][SLV97] in Section 2.1.2), we designed our taxonomy (Table 4.1) to specifically focus on code related activities and the motivation behind these activities. We wished to know the types of activities for which developers use development environments and the activities that these environments most poorly support. We also wanted to know whether developers use different tools for different activities.

From our own personal experience as software developers, hypotheses about what developers might find difficult, and topics of ongoing research, we also formulated nineteen hypothesized problems developers might have in obtaining or communicating about code-related knowledge (see Table 4.2).

Designing	Analyzing a new problem and mapping out the broad flow of code which will be used to solve the problem. […]
Writing	Creating a new method, source file, or script and getting it to a compilable state
Understanding	Determining information about code including the inputs and outputs to a method, what the call stack looks like, why the code is doing what it is doing, or the rationale behind a design decision. […]
Editing	Editing existing code and returning it to a compilable state.
Unit testing	Ensuring that code is behaving as expected. […]
Communicating	Any computer mediated or face-to-face communication about information relevant to a coding task […]
Overhead	Any other code related activities including building, synchronizing code, or checking in changes.
Other	Other code related activities.
Non code	Any other activities included in your work time

Table 4.1. Descriptions of activities read by respondents. Descriptions ending in […] have been shortened. See Appendix 1 for the complete materials.

4.2. Method

The study consisted of three parts: a survey about activities, tools, and problems (the "activities survey"), a series of semi-structured interviews, and a survey of work practices (the "follow-up survey").

4.2.1. Organization

The population we selected for study was software developers at Microsoft Corporation. Microsoft is a large software company whose products span a wide range of markets: operating systems (Windows); web portals (MSN); consumer devices (Windows Mobile, Xbox); office productivity applications (Office); and developer tools and infrastructure (Visual Studio, Great Plains, SQL Server). Of the roughly 63,000 employees in 2005, roughly 6,000 are software developers who work on shipping code in product groups. Other developers include those who work on test infrastructure and tools and those in Microsoft Consulting and Microsoft Research. These latter groups were excluded from our study.

It is not clear how a study of software development at Microsoft generalizes to software development in other professional environments. Given the diversity of environments – large software companies, small software companies, software developers in companies whose product is not the software itself, open-source development of commercial software – it is difficult to hypothesize how the present results would apply, but as the study crossed many of these variables, the results are likely to apply to other situations.

Within a product group at Microsoft, there are three core roles – software design engineer (SDE), program manager, and software test engineer. SDEs are responsible for software design, fixing bugs, and writing new features. Program managers are responsible for specifying and prioritizing features and for writing high-level feature specification documents which developers use to write code and testers use to write test cases. Software test engineers translate feature specifications into test cases and manually test the software. A somewhat less common role is software design engineers in test (SDE/T) who write test automation infrastructure. Members of each of these roles work in small teams of "individual contributors." Individual contributors are managed by a lead (e.g. lead software design engineer) who reports to a manager (e.g. software design engineer manager). Other less frequent roles include software architect, product designer, and usability engineer. Nearly all individual contributors have private offices (not cubes) and most do not share an office.

Product group work for a particular release of a product is divided into milestones. In the first milestone, program managers make initial decisions about what features will be in the release, what features developers will work on in subsequent milestones, and write initial feature specification documents. SDEs may work on bug fixes and patches from the previous release, try out new technologies, or plan major changes. Several milestones of development follow. Each milestone is divided more or less into a coding phase, where features are added, and a stabilization phase, where developers concentrate on fixing bugs. During the last milestone, most of the work involves fixing bugs. As the release nears, most changes become too time consuming and risky to test, and developers spend more time making the next version's code more maintainable (see Figure 4.1).

4.2.2. Procedure

The survey participants in the activities study first completed a number of demographic items. They next read the activity descriptions found in Table 4.1 (see Appendix 1 for the complete materials). They were then asked to report the fraction of their past week work time spent on each activity, choosing among 10% increments plus choices for 1%, 2%, and 5%. For each activity, they were asked the percent of time on that activity they used each of a set of tools or techniques, using the same scale. For each combination of tool/technique and activity, developers were asked to rate its effectiveness on a seven-point Likert scale. Finally, they rated the seriousness of nineteen hypothesized problems using a seven-point Likert scale. There were 204 questions in all.

While we expect respondents misremembered, misestimated, and misreported the time fractions, we expect they were able to differentiate across large distinctions like 0% and 5% or 10% and 40%. We normalized each group of fractional responses to sum to 100%.

Before deploying the activities survey we used two techniques to ensure that its design fit the activities, tools, techniques, and problems relevant to our target population. First we ran three experienced developers through the survey using a think-out-loud protocol. We adapted the survey wording and structure based on their feedback. Second, we developed a reduced version of the survey that included extensive opportunities for participants to write in additional activities, tools, and problems. We deployed this pilot survey to 99 randomly selected developers and received 28 responses. Any write-in response from two or more respondents was included in the final activities survey. No activities or problems met this criterion, but a few tools did.

We selected the four problems rated as the most serious amongst the nineteen problems developers rated in the activities survey (see Table 4.2 for ratings of the nineteen questions). We then designed a series of interview questions to elicit qualitative information about the character and impact of these problems. Several general, open-ended questions were added about how participants characterize their work and activities and on team communication patterns. Two interviewers attended each interview. Ten of the eleven interviewees consented to having the conversation audio-recorded. All three experimenters – Rob DeLine, Gina Venolia, and myself – used our notes and recordings to generate nearly 1,000 note cards of observations. The cards were then used for a card sort [WA87] where they were placed on the walls of a ~30 foot hallway to form groups, elicit themes and trends, and consolidate observations across interviewers and interviewees.

From the card sort we identified several preliminary hypotheses. We developed a follow-up survey to assess the hypotheses amenable to surveying. In this survey, participants first answered demographic questions. Next they answered questions about the size of their feature team, which was defined as, "the core group of developers that you work with." They then answered a series of questions about communication patterns, code ownership, design documents, understanding unfamiliar code, code duplication, unit testing, and adoption of agile practices. There were 187 questions in all.

4.2.3. Participants

We drew our participants for the three studies from the population that deals directly with code: SDEs, SDE/Ts, and architects at both the individual contributor and lead level. After the activities survey we decided to focus on developers working on *shipping* code, and so removed the responses from architects and SDE/Ts from our analysis and the subsequent observations. We excluded contractors because of logistical problems and excluded interns because we wished to generalize to professional software developers.

Participants were invited to participate in the surveys by email and sent a reminder email several days before the surveys were closed if they had not yet responded. Respondents were compensated by entry in a drawing for $50 gift certificates. In the activities survey, we randomly sampled 1,000 participants from the participant pool, excluding those invited to take the pilot survey. We received 157 responses, 104 from SDEs, including 18 from lead SDEs. We were somewhat disappointed with the response rate and attribute it to the survey being deployed in early July when many were on vacation, some technical problems with the survey deployment, and sheer size of the survey. In the follow-up survey, we randomly sampled 1,000 from the same pool excluding SDETs and recipients of the activities and pilot surveys. We received 187 responses, 176 from SDEs.

The activities survey contained several demographic questions. Since participants from all surveys were randomly sampled from the same population of SDEs, these demographics apply to all study participants. The average respondent is in their 30's with an undergrad degree, 12.1 (± 6.5) years programming, 5.8 (± 4.2) years at Microsoft, and 2.9 (± 2.4) years on their current team; 89% of respondents are male. 37% reported that most of their code base was written in C#, compared to 56% in C or C++, reflecting both older, established code bases and newer code bases written in C#.

We interviewed eleven respondents, five SDEs from the pilot survey and 6 lead SDEs from the activities survey.

4.3. Activities survey results

Far from spending all of their time understanding or editing existing code, developers reported spending most of their time elsewhere. As the study was exploratory rather than being hypothesis driven, results are presented with descriptive statistics. Times are reported using the mean (± standard deviation).

4.3.1. Time breakdown

Developers reported spending a little less than half of their time (49% ± 39%) fixing bugs, 36% (± 37%) writing new features, and the rest (15% ± 21%) making code more maintainable. This confirmed our expectation that most developers spend much of their time fixing bugs. But the vast variability in these numbers also demonstrates that typical development activity varies greatly across software's lifecycle (Figure 4.1) and between developers (Figure 4.2). We attribute this variability to reporting error, differences across lifecycle stages, and interactions between activities (Figure 4.3). Activities are not independent, but are interrelated (Figure 4.3). For example, developers who are designing are also likely to be writing code. But developers understanding code are substantially less likely to be writing code. Most developers engage in multiple activities in a given week (Figure 4.4).

There was a positive linear relationship of tool usage to effectiveness (Figure 4.5). There are many possible explanations for this relationship, which our data are unable to distinguish. Developers might choose to use the tools they find the most effective. Developers might view the tools they are using as the most effective. Or an individual developer might not use the tools they find most effective, and the relationship might occur for another reason.

4.3.2. Communicating

Developers both preferred and spent more time using face-to-face communication than electronic communication (Figure 4.5a), replicating a 1994 finding [PSV94] of a strong preference for face-to-face over email. Yet, email has since increased in prominence and sophistication and instant messaging has made possible short response time and interactive communication. Developers gave a number of reasons for preferring face-to-face communication. Developers reported that email questions often took hours or days to receive a response, that developers frequently misinterpreted emails' meanings, writing an email without immediate feedback required explanations in more or less detail than required, and that email was just tedious to write. We believe many of these problems generalize to other electronic communication such as documentation, bug databases, and IM. Developers still use email when the issue is of low priority, involves multiple people, or involves non-teammates, averaging 16.1 (± 14.5) emails sent to teammates in the prior week and 5.9 (± 11.5) to non-teammates. The low use of email might limit benefits from systems helping developers locate old emails, and the barriers discouraging email use might make it difficult to encourage more retention of knowledge in emails. Unplanned, face-to-face meetings happen frequently with teammates, averaging 8.4 (± 11.7) per week, and much less frequently with non-teammates, averaging 2.6 (± 4.0). Communication within the team is much more common than communication across teams, indicating that the culture of informal communication works well and that the team boundaries are typically in the right places.

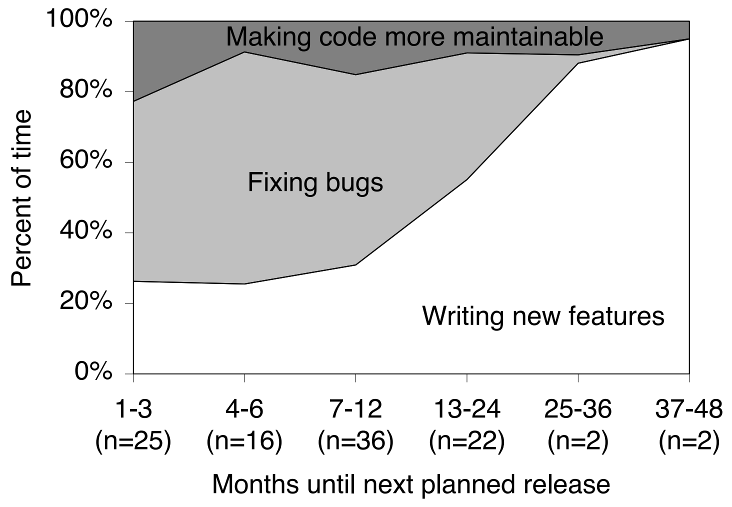


Figure 4.1. The time spent fixing bugs, making code more maintainable, and writing new features varies with the time until the product is planned to be released. Time flows from right to left, ranging from 37-48 months before the next planned release to 1-3 months before the next planned release.

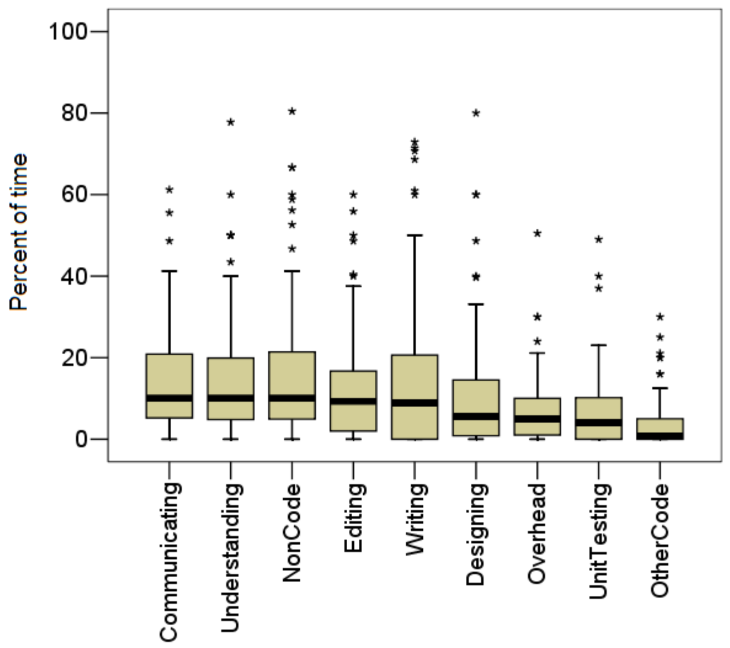


Figure 4.2. There is a great degree of variability in the time spent on the activities described in Table 4.1.

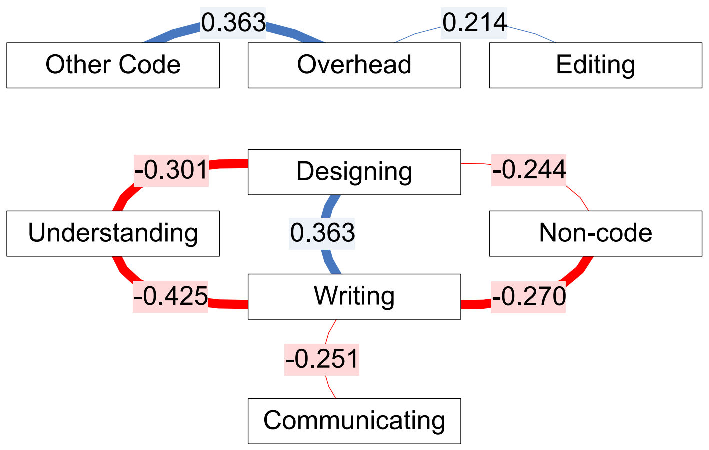


Figure 4.3. Statistically significant correlations between time spent on each activity. Negative numbers (red) indicate inverse relationships. (Spearman's rho, thin lines for *p*<0.05, thick lines for *p*<0.01, *n*=104.)

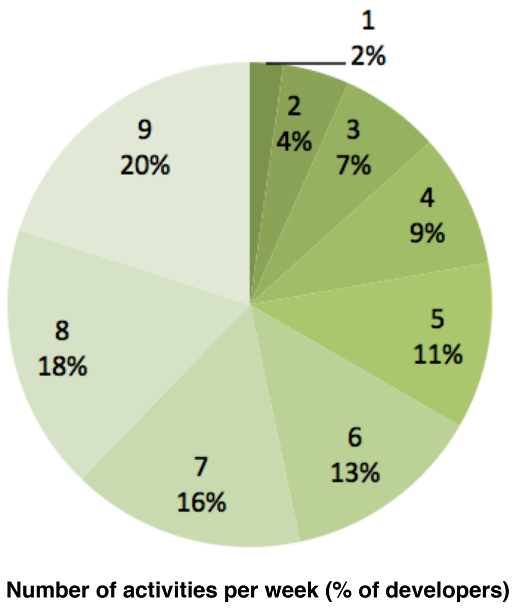


Figure 4.4. Most developers engage in a number of activities in a given week.

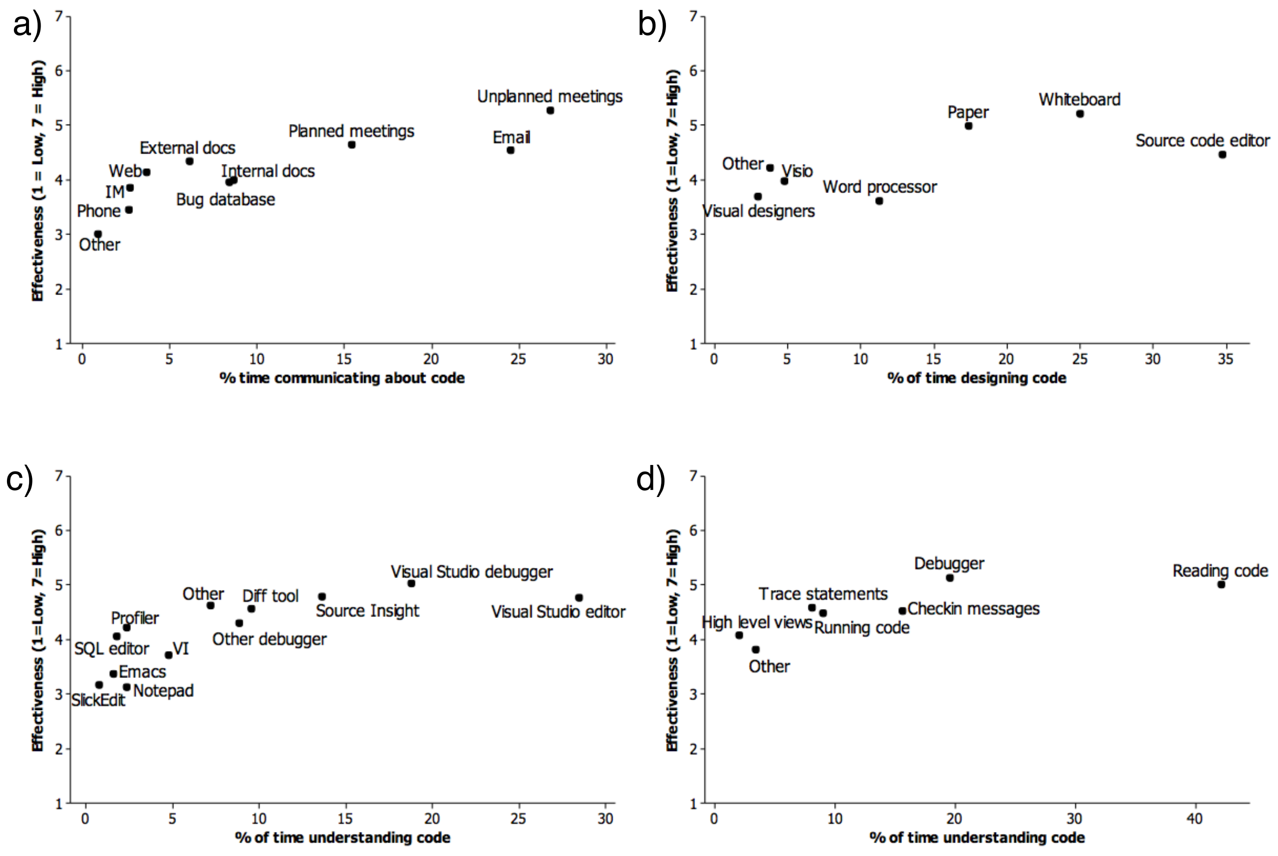


Figure 4.5 a) Developers communicating about code spent the most time in unplanned meetings, also rating it their favorite technique. b) Developers designing code reported spending the most time with the source code editor, but rated diagrammatic tools including whiteboard and paper as more effective. Interactive diagram tools such as visual designers and Visio were rated less effective and used less frequently. c) Of the *tools* for understanding code, developers spent the most time in the Visual Studio editors and debuggers, but also used a variety of other tools. d) Of the *techniques* for understanding code, developers, by far, spent the most time simply reading, but also used the debugger and checkin messages. Developers reported using high-level views of code very rarely.

This is a serious problem for me	**% agree**
Code Understanding	
Understanding the rationale behind a piece of code	66%
Understanding code that someone else wrote	56%
Understanding the history of a piece of code	51%
Understanding code that I wrote a while ago	17%
Task Switching	
Having to switch tasks often because of requests from my teammates or manager	62%
Having to switch tasks because my current task gets blocked	50%
Modularity	
Being aware of changes to code elsewhere that impact my code	61%
Understanding the impact of changes I make on code elsewhere	55%
Links between Artifacts	
Finding all the places code has been duplicated	59%
Understanding who “owns” a piece of code	50%
Finding the bugs related to a piece of code	41%
Finding code related to a bug	28%
Finding out who is currently modifying a piece of code	16%
Team	
Convincing managers that I should spend time rearchitecting, refactoring, or rewriting code	43%
Convincing developers on other teams within Microsoft to make changes to code I depend on	42%
Getting enough time with senior developers more knowledgeable about parts of code I'm working on	34%
Expertise Finding	
Finding the right person to talk to about a piece of code	39%
Finding the right person to talk to about a bug	38%
Finding the right person to review a change before check-in	19%

Table 4.2. Developer ratings of proposed problems. In the survey, problems were presented without headings and in a different order.

Most developers reported using IM only infrequently for code related tasks. It was more frequently used to contact teammates for social functions (e.g. going to lunch) or to talk to family. Use of the telephone for code-related communication was similarly rare.

4.3.3. Designing

Despite the availability of high-level views of code and visual editors such as tools for UML, developers remain focused on the code itself. Developers reported designing the most in a source code editor while paper and whiteboards were perceived most effective (Figure 4.5b). We hypothesize that the need to find details about the existing design by using a source code editor discourages increased use of paper or whiteboards, even though both were viewed as more effective tools.

4.3.4. Perceived problems

Table 4.2 lists the problems we proposed in the survey and the percent of respondents who agreed that the problem is a "serious problem for me." The top four are: understanding the rationale behind existing code, having to switch tasks because of manager or teammate requests, being aware of changes elsewhere, and finding code duplicates. We focused our semi-structured interviews on these problems to discern what makes them difficult. Several themes emerged:

- Developers go to great lengths to create and maintain rich mental models of code and do not rely on external representations.

- Understanding the rationale behind code is the biggest problem for developers. When trying to understand a piece of code, developers turn first to the code itself and, when that fails, to their social network.

- Developers and development managers use a variety of tools and work practices and are actively looking for better solutions.

We present these themes with support from our follow-up survey.

4.4. Maintaining mental models

Developers create and maintain intricate mental modes of the code. Through our interviews, we know that developers, without referencing written material, can talk in detail about their product's architecture, how the architecture is implemented, who owns what parts, the history of the code, to-dos, wish-lists, and meta-information about the code. For the most part this knowledge is never written down, except in transient forms such as sketches on a whiteboard. One interviewee summed it up well - "Lots of design information is kept in people's heads."

4.4.1. Personal code ownership

Mental models are expensive to create and maintain. Developers have a strong notion of *personal code ownership*, which constrains the amount of code they have to understand in detail. In our follow-up survey, 77% of respondents agreed[3] with the statement, “There is a clear distinction between code that I own and the code owned by my teammates.” (On the other hand some teams have a policy to avoid personal code ownership because it makes individuals too indispensable and promotes, in the words of one of our interviewees, “too much passion around the code.”) Code ownership is a long-term proposition, reducing the number of times that a developer has to learn a new code base. In the activities survey, the average time on the current code base was 2.6 years, with 32% reporting 6 years or more. Personal code ownership is usually tacit, i.e. part of the mental model. Written records of ownership, when present, are often out-of-date and distrusted.

We received conflicting information about design documents for issues within a team. Design documents are usually written by a developer immediately prior to implementing a larger change that affects other developers, in order to solicit other developers’ input on important decisions. In the interviews, design documents were described almost as read-once media, serving to structure the developer’s thinking and as an artifact to design-review, but seldom read later and almost never kept up-to-date. On the other hand, our follow-up survey respondents reported a different picture of design documents for issues within the team: feature teams wrote an average of 7.6 (± 10.2) documents in the prior year, and kept 51% of them up-to-date. We were surprised with these numbers and cannot reconcile them with the results of the interviews.

4.4.2. Team code ownership and the “moat”

Even stronger than personal code ownership is a notion of *team code ownership*. An overwhelming 92% agreed with the statement “There is a clear distinction between the code my feature team owns and the code owned by other teams.” Feature teams are small. 93% stated that their feature team consisted of 2-4 people (including the respondent). There seems to be a sweet spot at three-person feature teams, reported by 49%. Feature teams are almost always colocated, facilitating informal knowledge sharing.

One of the ways developers maintain their mental model of their team’s code is by subscribing to check-in messages by email, though several interviewees expressed dissatisfaction with the lack of detail provided by teammates.

Small feature teams’ strong code ownership forms a kind of *moat*, isolating them from outside perturbations. The moat is defined, in part, by design documents, which specify the interface across the moat. Design documents for cross-team issues were less common than those relevant to issues within the team. Although the average number of design documents

[3] Throughout this chapter, the word *agree* means that the participant selected either “Somewhat agree”, “Agree”, or “Strongly agree” from a seven-point Likert scale.

written in the last year for cross-team issues was 4.5 (± 7.8), significantly less than the 7.6 (± 10.2) for within-team issues (two-tailed *t*-test, p<0.01, t=4.78), cross-team design documents are significantly more likely to be kept up-to-date (61% versus 51%, two-tailed *t*-test, p<0.01, t=−3.58). The greater care taken with cross-team design documents reflects their important role in defining the moat.

Unit tests, used by 79% of our respondents, are an important part of the development process for many reasons. One surprising function is to defend the moat from outside perturbations – 54% of respondents agreed that an important benefit of unit testing is that "they isolate dependencies between teams."

Almost all teams have a *team historian* who is the go-to person for questions about the code. Often this person is the developer lead and has been with the code base the longest.

4.4.3. New team members

Creating a mental model from scratch requires a lot of energy for the new team member and the team as a whole. Often the newcomer is assigned a mentor, often the team historian, designated as the first point of contact for questions about the code. The mentor helps to jumpstart the newcomer's mental model and social network. Newcomers are much more likely to read the team's design documents than seasoned team members. Some teams maintain online documents specifically for newcomers. Unguided exploration of the code is rare; more commonly the newcomer is assigned bugs specifically to introduce them to the code while minimizing risk. While all changes are code reviewed before checkin, newcomers receive extra attention and feedback on early changes they make. Several interviewees viewed fixing bugs as requiring less design knowledge than implementing new features. Bug fixing allows newcomers to do useful work while still learning the code base.

4.4.4. Code duplication

Two previous studies [RC96][KBL04] and the focus of clone detection tools (e.g. CCFinder [KKI02]) led us to expect that when developers were asked about code duplication, they would discuss copying and pasting example API usage code, subclasses, or other hard-to-understand example code or even regale us with stories of hard to refactor clones. When pressed, a few admitted to copying and pasting code in dubious ways. Yet most responded with stories that had nothing to do with finding example code or copy and paste.

	Repeated work	Example	Scattering	Fork	Branch	Language
Creation	Separate developers implement same functionality	Copy and paste of example code	Design decision distributed over multiple methods	Copy of other team's code base	Branch maintained separately	Reimplementation by same developer in different language
Aware when created	No	Yes	Yes	Yes	Yes	Yes
Refactoring challenge	Awareness at creation; different design decisions	Investment creating abstraction	Changing architecture	Convincing other team to make changes	Combining released branches	Changing architecture or implementation language
Size of clone	Members, classes	Members, classes	Members, classes	Many classes, code base	Code base	Members, classes
Repeated change	24%	44%	29%	13%	25%	29%
Refactoring	19%	39%	14%	5%	6%	15%
Agree problem	42%	41%	37%	29%	28%	29%

Table 4.3. Each form of code duplication identified in the interviews is listed in each of the columns. The bottom three rows summarize responses from the follow-up survey in which developers reported if they had made repeated or related changes in the past week due to each form of code duplication (Repeated change), had refactored each form of code duplication (Refactoring), and believed that each form of code duplication made their code difficult to manage (Agree problem).

From our interviews, we identified six distinct forms of code duplication (Table 4.3). Each clone type can be characterized by its creation mechanism, whether developers are aware they are creating clones, the refactoring challenges to remove the clones, and the size of the clones. In our follow-up survey, we asked developers to report if they had made repeated or related changes in code during the past week as a result of each form of code duplication. Developers also reported if they have refactored code duplication in the past week and if the code duplication caused their code to be difficult to manage (Agree problem in Table 4.3).

In *repeated work clones*, multiple developers separately and unknowingly reimplement the same functionality. One developer reported that he had been implementing a small piece of functionality that another developer was also working on for a different problem until a program manager suggested that he talk to a second developer. After creation, interviewees viewed these clones as being difficult to refactor as each developer may have made subtly different decisions that are difficult to change.

The most studied clone type, *example clones*, occurs when some usage context code which illustrates how to create or make use of some code is copied and pasted and modified. We expect that this usually involves a small amount of code. Kim *et al.* [KSN05] argue that copies frequently diverge and that it is difficult to predict whether the clones would be better off factored into a new abstraction.

Scattered clones, or logical clones, involve crosscutting changes in the aspect-oriented programming sense [KHH01]. Here, changing a particular decision requires making changes to many widely dispersed areas of code. One developer reported that correctly changing one method required changing another method that was hidden several calls deeper into the component. Another reported that they would sometimes make a change, hope for the best, and rely on testers to find any other necessary related changes.

Fork clones occur when a team takes a large portion of code from another team. One developer reported doing this when they wished to use code that the original team was not ready to ship. They subsequently heavily modified the code to remove functionality they did not need. Forks occur when a consuming team wishes to use functionality provided by a producing team in ways that the producing team is unable to support. Interviewees, when asked, all agreed that it was best to avoid forked code whenever possible. Yet, when faced with the alternative of reimplementing the functionality from scratch, forking is frequently a better alternative. Particularly difficult are bug fixes. The consuming team must monitor bug fixes made by the producing team and reimplement the fixes themselves, taking on much of the maintenance burden of the producing team.

Branch clones occur when developers must reimplement their change in several branches of the same code base. These are not clones in the strict sense of duplicate code but rather copies of the entire code base in various stages of release. One developer reported fixing a bug in both code used in production and the current version under development. Branch clones were rated as being the most frequently occurring and the most frequent in efforts to remove them, presumably by combining branches.

Language clones involve the same code implemented in multiple languages. One developer reported having the same methods in both C++ and C#.

In contrast to the clone detection literature's narrow view of cloning as syntactically similar code, developers viewed cloning as making the same change several times. This includes many cases which involve code not syntactically similar and in a single code base but scattered, cloned across code bases, repeated in multiple languages, or in multiple branches. From the developer's perspective, many of these problems still look similar in that individual bugs have to be fixed in several places, new feature work involves changes in many different places, or changes crosscut the strong team code ownership boundary. Future empirical work might be best served by focusing on this broader definition of repeating the same work.

4.5. Rationale and communication

Of all the hypothesized problems developers rated, understanding the rationale behind code was perceived to be the most serious. 66% of the respondents agreed that "understanding the rationale behind a piece of code" was a serious problem (see Table 4.2). There are many facets to the rationale problem: 82% agree that it takes a lot of effort to understand “why the code is implemented the way it is,” 73% “whether the code was written as a temporary workaround,” 69% “how it works,” and 62% “what it’s trying to accomplish.”

The many developers who reported problems with rationale led us to investigate how developers understand and explore code. We found that developers had many complaints about using their tools to explore code, eschewed design documents for interrupting teammates, had code ownership boundaries to minimize how much they must understand, and rarely documented their understanding for others. This led to the second most serious problem – developers felt they were too frequently interrupted by their teammates. We also investigated how developers maintain awareness of changes affecting their code.

4.5.1. Investigating code rationale

When investigating a piece of code, developer first turn to the code itself: on average, respondents spent 42% of their understanding time examining the source code, 20% using the debugger, 16% examining check-in comments or version diffs, 9% running the program, 8% using debug or trace statements, and 5% using other means (Figure 4.5d). In other words, the code itself is the best source of information about the code. However using the code can be challenging. Developers commonly become disoriented in unfamiliar source code, and discerning the relationship between observed program behavior and the source code is often difficult.

When the code itself does not give the answers the developer needs, one might expect the developer to turn next to the vast amount of information that is written about it – the bug reports, the specs, the design documents, the emails, etc. This however is emphatically not the case. Several factors combined to dissuade most developers from using design documents for understanding code. First, finding design documents was frequently difficult. Design documents were generally stored on internal websites without a usable search facility, forcing developers to manually navigate hierarchic collections looking for the appropriate design document. Thus, even if developers thought there was a possibility of a design document containing the information they cared about, it was not worth looking for. If search were available, it was not clear that developers would know the correct search terms. Second, design documents were not reliably updated. Thus, developers consulting a document would not be sure if the code still conformed to the document and would still be forced to inspect the code.

The second recourse for investigating the rationale behind code is the social network. If the developer thinks a teammate might be able to provide the needed information (or the name of the person who might), he will walk down the hall to talk with the teammate.

Once the developer has the desired information, he returns to his office, applies the newfound information, and gets on with his work. This information is precious: it is demonstrably useful, demonstrably hard to ascertain from the code, and was obtained at a high cost. Yet it is exceedingly rare for the developer to write this morsel down anywhere. The next person who needs the same information must go through the same laborious discovery process. There are plenty of reasons that a developer would choose to not record the information. The overhead of checking the code out, editing it, and checking it back in (possibly triggering check-in review processes, merge conflicts, test suite runs, etc.) is enough to dissuade the developer from recording the information as a comment in the code. Some interviewees expressed the concern that the newfound information was not authoritative enough to add permanently to the code or that checking in the comment under their own names would inappropriately tag them as experts. Hence the information tends to remain in the developers' heads, where it is subject to institutional memory loss.

4.5.2. Interruptions

Each of these unplanned, face-to-face meetings represents an interruption of at least one person. Recovering from these interruptions is a substantial problem, ranking second with 62% of developers agreeing that this is the case (Table 4.2). Recovering from an interruption can be difficult. Developers must remember goals, decisions, hypotheses, and interpretations from the task they were working on and risk inserting bugs if they misremember.

Developers have adopted various strategies to mitigate the effects of interruptions on themselves, such as using a closed office door or other social cues to deflect interruptions, working on complicated tasks at times of the day when interruptions are infrequent, staving off an interruption for a moment while finishing a thought, or scheduling "office hours." Sometimes the interrupter mitigates the impact of interruption by using email instead of face-to-face for low-priority issues or emailing a warning 10 minutes before the interruption to give the interrupted person a chance to save his working context by writing down notes.

While many (though not all) interviewees indicated that they received too many interruptions, all acknowledged that interruptions were a valuable part of the work culture. Interestingly, two interviewees indicated that interruptions had become more of a problem since their teams had adopted agile processes.

4.5.3. Bug investigation example

Developers reported spending nearly half of their time fixing bugs. A bug investigation helps illustrate how their tools, activities, and problems interact to make fixing bugs possible but also suboptimal. When asked to describe an instance of a difficulty understanding the rationale behind a piece of code, one developer responded with a bug investigation narrative. This narrative is only a single story and not necessarily general. And it is based on a recollection of events that may not be completely accurate. But it illustrates several themes supported by the interview and survey data.

After being assigned a new bug through a bug tracking tool, the developer first reproduced the bug by navigating to a webpage and ensuring that *error 500 – internal error* was received as reported in the bug. Next, the developer attached the Visual Studio debugger to the web server, set it to break on exceptions, reproduced the error again, and was presented with a null reference exception in Visual Studio. From an inspection of the call stack window, the developer considered the functions that might be responsible for producing the erroneous value. The developer switched to emacs to read the methods and used ctags.exe to browse callers of methods. The developer then switched back to the Visual Studio debugger to change values at run time and see what the effects were. The developer made a change, recompiled, and found that the same exception was still being produced. Finally, the developer browsed further up the call-stack, tracing the erroneous value to one object, then to another object, and finally to a third object protected with mutexes.

By this time, the developer had wandered into code that he did not understand and did not "own" – or have primary responsibility for making changes. But a second developer was working on a high profile feature that touched this code, so he immediately knew that this second developer would understand this code. He went to the second developer's office, interrupted the second developer, and engaged him in a discussion about the rationale behind the code. He walked back to his office, made a change based on this information, determined that the change would not work, leaving him with a new problem with this unowned code. He walked back to the second developer's office who then told him that the functionality causing the problem was actually related to code that a third developer was working on. They both went to visit the third developer's office, only to find the third developer is at lunch. The first developer, now blocked, switched to another task. After lunch, both developers returned to the third developer's office, had a design discussion about how the functionality should behave, and finally passed the first developer's bug to the third developer to make the fix.

This story illustrates several themes in our surveys and interviews:

- Developers rapidly switch between multiple tools.
- When looking for detailed information about code, developers first explore the code by reading it and using a debugger.
- When unable to find answers exploring code, developers consult knowledgeable teammates rather than specs, design documents, email, or other artifacts.
- Face-to-face communication is strongly preferred over email or IM.
- Developers switch tasks when blocked or interrupted by teammates seeking code knowledge.
- Software development is a highly social process.
- While code ownership within a team is well understood, changes crosscut ownership boundaries.
- Developers spend vast amounts of time gathering precious, demonstrably useful information, but rarely record it for future developers.

4.6. Openness to change

Developers and development teams are constantly trying new tools and work practices to try to optimize their work. Developers use a variety of tools to do their job. When writing code, 49% use two or more tools, and 19% use three or more.

In our interviews we found several development teams that were experimenting with "agile practices," a collection of behaviors intended to make the software development process more efficient[4]. Some teams were gingerly dipping a toe into the agile water, while others were jumping in with both feet. In our follow-up survey, we found little overall use of agile practices (see Table 4.4). On the other hand, 48% of respondents agreed that their team was using two or more of the eight practices, 32% three or more, and 20% four or more. A few respondents (3%) reported that their teams use seven or all eight of the practices. Most developers want to continue adopting agile practices (53% agreed that they thought their team "should adopt agile software development methodologies more aggressively") while a few were skeptical (14% agreed that their team should adopt *less* aggressively).

Does your team use	**% agree**
Collective code ownership within the team	49%
"Sprints," i.e. a development cycle that last four (or so) weeks	42%
An intentional policy to involve customers (internal or external) deeply into design and planning	33%
"Scrum meetings," i.e. a brief daily status meeting including all stakeholders	25%
"Burndown" estimate or chart, i.e. a measure of the time remaining in the sprint	24%
An intentional policy of preferring face-to-face over electronic communications	16%
Pair programming, i.e. developers working together, shoulder-to-shoulder on a problem	16%
A "bullpen" or other open-floorplan space for the team	10%

Table 4.4. Agile practices adopted by respondents.

Developers adopted specific agile practices when they felt their benefits were compelling. Developers shunned design documents in favor of face-to-face communication, designed minimally rather than up front, and employed unit testing. Developer leads reported prefer-

[4] http://agilemanifesto.org/

ring daily standup team meetings over weekly team meetings, since daily meetings encouraged teammates to help each other and assisted the lead in responding to problems blocking developers' progress. Several teams had gone further by adopting an entire agile process, Scrum [SB01], and reported using radical collocation, collective ownership, and sprints.

4.7. Conclusions

This chapter demonstrated the central importance of understanding and exploring code to software development. Developers rated understanding rationale behind code the most serious problem they faced, creating work for them to reconstruct rationale by exploring and investigating code. Developers often instead ask their teammates, usually after performing a due diligence amount of code investigation themselves, but this causes additional problems – developers complained about being less productive because they were constantly interrupted. Developers tried to use design documents to capture important information about code, but developers we interviewed generally did not greatly rely on these, instead preferring to use the code rather than risk working with out of date documents. These findings suggest that a tool that helped developers to more effectively understand and explore code might better align with developers' development process than documentation tools, and that, if it were able to help developers rely on the code more rather than interrupt their teammates, might even reduce the problems developers that experience with interruptions. However, the results in these studies revealed little about information needs during activities understanding and exploring code that tools could more effectively satisfy. What strategies do developers use to understand and explore code, what questions do they ask, and what further problems do developers experience? The next three chapters explore these issues.

5.

UNDERSTANDING DESIGN DECISIONS[5]

One way to view a program and its design is as a network of interconnected design decisions (e.g., [P72][SGW01]). Software engineering teaches that developers work with design decisions describing a choice between possible alternatives and dependency relationships between these choices. Developers are told to apply information hiding to prevent likely anticipated changes from rippling through a system [P72], to refactor code clones to allow a single decision to be changed in one place [KKI02], to write modular specifications allowing reasoning in isolation of the rest of the system, and to respect architectural styles [SG96] in their decisions to prevent architectural drift and erosion [PW92]. But, despite efforts to make design decisions easier to reason about, little is known about how developers explore and understand design decisions during coding tasks.

One important factor likely influencing how developers understand design is a developers' knowledge of software design. A long tradition of studies in cognitive science has established that experts perform better not because they are smarter but because they have knowledge which novices lack (e.g., [CS73]). Studies of programmers have also found these differences, but have mostly studied knowledge in the form of highly local code idioms such as for loops (e.g., [D90]). Software engineering suggests that developers have a wide variety of knowledge about good design in the form of abstractions such as design patterns [GHJ95] and architectural styles [SG96]. But little is known about this knowledge or how it helps developers work more effectively. A better understanding might lead to better guidelines for training software engineers and inform the design of tools that help developers without this knowledge.

This chapter describes a study conducted to understand how developers perform challenging code modification tasks involving design decisions and to understand the effects of experience on this process. In two lab tasks, participants were provided with criticisms of the current design of a program and instructed to improve the design. Since the prior research had not yet identified the key variables with any degree of confidence, the study was conducted in an exploratory, open-ended way. We observed in detail how different developers approached the tasks, which allowed us to observe patterns and identify key variables that might be studied by future experiments. This study addressed three research questions:

- How do developers reason about design decisions during coding tasks?

[5] This chapter based on work previously published in [LGH07].

- How does experience affect changes made to code?
- How does experience affect how developers work?

We found that:

- Developers described the design using facts which took several forms and served a number of roles.
- Experts' changes addressed the cause of the problems while novices' changes addressed the symptoms.
- Experts made better decisions about which methods were relevant, they talked about the code using abstractions rather than statement-by-statement descriptions, they explained facts novices were unable to explain, and they implemented a change more quickly than novices.

5.1. Method

We conducted an exploratory lab study in which participants worked on two tasks for 1.5 hours each. The tasks were challenging and involved changes to a real open source application. We recorded participants' activity using think-aloud, video, and Eclipse instrumentation to get a full picture of what participants were doing.

5.1.1. Study design

We recruited developers with diverse levels of experience, and brought them into the lab to observe their work in detail. A lab study had several advantages over a field study. We could compare participants' behavior on exactly the same tasks, use tasks designed to require that they understand the code's design, and control for prior experience with the application. We controlled for ordering effects between tasks by assigning half of the participants to receive each task first and ensured that there were experienced and novice participants in both conditions. Doing an exploratory, observational study, rather than a controlled experiment, let us build a model of developer activity and differences suggested by it that we did not know beforehand. Our quantitative comparisons between experts and novices are not a controlled experiment because we picked dependent variables post-hoc from qualitative analysis of participant activity. We chose 13 participants, rather than a larger number (which might have resulted in statistically significant differences), to make manually transcribing and analyzing the voluminous transcripts feasible.

We initially planned to investigate the effect of providing architectural information on how developers work with code. We provided half of our participants with a component and connector diagram [CBB02] that we reverse engineered. While these participants read the diagram at the beginning of the task, most used the diagram only to generate and test hypotheses about how classes were connected or as scratch paper to draw call graphs or write down method names. We were unable to observe any differences about how the developers were working that could be attributed to having the diagram. We thus do not consider these diagrams further.

Participant		yrs industry experience	KLOC largest program edited	yrs Java experience	Design patterns	Architectural styles	Refactoring tools	Code navigation proficiency	Enjoy designing
NOVICES	a	0 (research)	10	4	3	7	6	2	6
	b	0 (research)	7.5	3	3	1	1	2	1
	c	0.5	1	few	3	4	3	4	1
	d	1.5	75	5	2	3	1	3	1
	e	2	2	1	2	3	1	1	1
	f	2.5	1	2	3	6	7	1	1
	g	2.5	10	8	2	4	4	1	2
	h	2.5	136	4	2	2	2	2	2
	i	3	2	4	4	6	1	1	1
	j	3	10	6	2	6	2	4	1
		2.25	**8.75**	**4**	**3**	**4**	**2**	**2**	**1**
EXPERTS	K	3	100	7	1	1	1	1	1
	L	10	100	10	1	1	1	2	1
	M	10.5	500	3	1	2	6	2	2
		10	**100**	**7**	**1**	**1**	**1**	**2**	**1**

Table 5.1. Participants' self-reported experience with medians for novices and experts. We assume internships lasted 1/4 of a year. For the experience columns on the right, 1 is the most experience and 7 the least.

5.1.2. Participants

Thirteen participants were recruited from undergraduate students, masters students, doctoral students, and staff at Carnegie Mellon University who reported that they (1) had at least two programming internships or fulltime development experience and (2) were comfortable programming in Java. Industry experience and self-reported expertise data were collected with a short demographic survey completed when potential participants responded to our recruiting materials. Participants also self-rated their experience with design patterns, architectural styles, and refactoring tools, their perceived proficiency navigating code, and the degree to which they enjoyed designing (Table 5.1). Participants were asked to give the size of the largest program they had worked on. The low responses of several participants to this question suggest that they may have had inaccurate knowledge or misunderstood the question. Two participants who responded to our recruiting materials had no industry experience. Both were graduate students who reported significant research programming experience, so we accepted them for our study. Ten participants reported they had used Eclipse before, one reported she had not, and two were not asked about their experience using Eclipse. This suggests our results do not reflect challenges learning Eclipse.

Participants included one undergraduate student, four masters students, seven doctoral students, and one staff member. Participants had industry experience on a wide spectrum of applications including databases, banking software, and operating systems. Twelve males and one female participated. Participants were paid for their time. Table 5.1 shows the self-reported experience sorted by years of industry experience. We refer to participants as "experts" and "novices" for brevity. However, although the "novices" had limited industry experience, they still had substantial programming experience and should not be confused with novice programmers. Two participants (L, M) were labeled experts because they had far more experience than the novices. We labeled a third participant (K) an expert because he had just as much experience as the novices but considerably higher self-ratings and because his behavior was closer to the other experts. We refer to novices by lowercase letters and experts by uppercase letters.

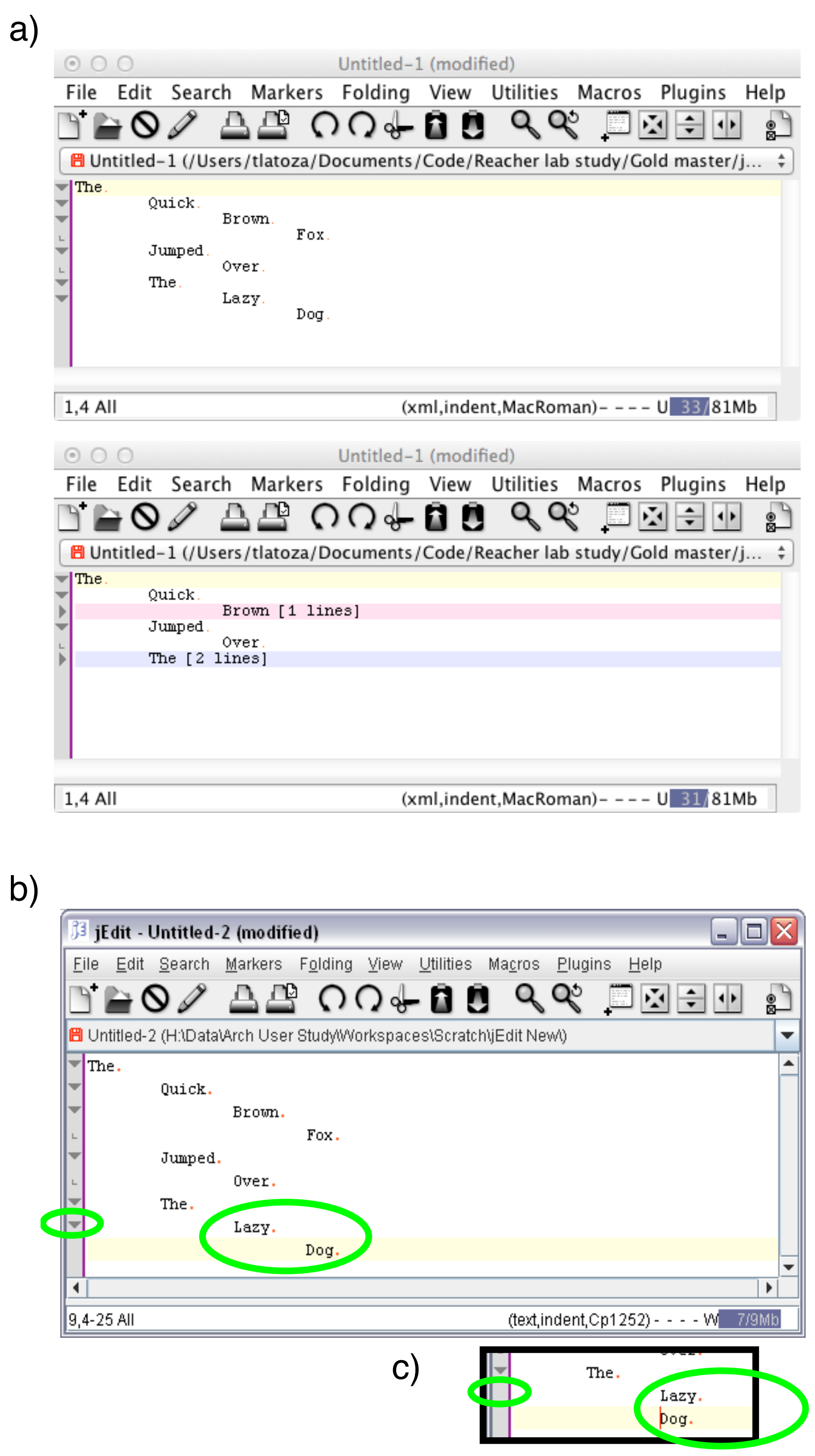
One expert (L) had participated in an earlier study using the same application we used. Any advantages this participant had are potentially attributable to greater knowledge about the application rather than experience. However, we believe the effect of this possible contamination is minimal since our task required an understanding of an entirely different part of the application than the previous study, and we did not observe that the participant's knowledge from the previous study helped in any substantial way.

5.1.3. Tasks

Participants worked with jEdit, an open source text editor, which has also been used in previous lab [RCM04] and version control [ZZW05] studies. Participants were provided an Eclipse workspace with the entire jEdit 4.3pre5 source, which is 54,720 non-comment, non-blank lines of Java.

To ensure that the tasks were the right length and difficulty and that they challenged developers in their ability to understand design, we piloted them with three pilot participants. After poor experiences with functional change tasks, we picked nonfunctional tasks focused on improving the design rather than implementing features or fixing bugs. We hoped this would challenge participants' ability to understand design more than fully specified changes to the application's behavior. Both tasks were designed to be architectural in nature by involving interactions between classes that we had identified as top level components on our component and connector diagrams. Many of the methods that participants studied were architecturally significant in that they participated in the connectors joining these components.

In both tasks, we provided design criticisms and corresponding code locations. Participants were instructed to "investigate why this is the case and implement a better design" and "make the design as ideal as possible by the criteria of performance, understandability, and reusability". To ensure they knew that they were expected to implement changes, they were instructed to "carefully budget your time to make your improved design as ideal as possible while carefully scoping your changes to what you can implement within your allotted time" while changing "as much or as little code as you'd like".


a)
Untitled-1 (modified)
File Edit Search Markers Folding View Utilities Macros Plugins Help
Untitled-1 (/Users/tlatoza/Documents/Code/Reacher lab study/Gold master/j...
b)
jEdit - Untitled-2 (modified)
File Edit Search Markers Folding View Utilities Macros Plugins Help
Untitled-2 (H:\Data\Arch User Study\Workspaces\Scratch\jEdit New\)
c)

```
The
        Quick
                Brown
                        Fox
        Jumped
                Over
        The
                Lazy
                        Dog
```

```
1,4 All                    (xml,indent,MacRoman)- - - - U 33/81Mb
```

```
The
        Quick
                Brown [1 lines]
        Jumped
                Over
        The [2 lines]
```

```
1,4 All                    (xml,indent,MacRoman)- - - - U 31/81Mb
```

```
The.
        Quick.
                Brown.
                        Fox.
        Jumped.
                Over.
        The.
                Lazy.
                        Dog.
```

```
9,4-25 All                 (text,indent,Cp1252) - - - - W 7/9Mb
```

```
        The.
                Lazy.
                Dog.
```

Figure 5.1. a) Clicking arrows in the margin toggles folds between expanded and hidden. b) When text is edited, its fold level may change (circled lines), c) causing the arrow in the margin to disappear.

On the **FOLDS** task, participants investigated how fold level state was updated following edits to a file. jEdit allows hierarchical regions of text of the viewed file (e.g., a method body) to toggle between being "folded" up and hidden or viewed, by clicking on an arrow (see Figure 5.1.). The fold level refers to a line's level in the hierarchy of folds. Following an edit to a line, the line's fold level becomes invalid. When it is next requested by a call to `getFoldLevel`, it is recomputed and stored in a cache in `LineManager`, part of the buffer's implementation. If the fold level changes, a `fireFoldLevelChanged` event is sent. Subsequent calls to `get-FoldLevel` retrieve the line's cached fold level from `LineManager` rather than recomputing it.

Participants were provided the following code excerpt:

```
/* force the fold levels to be updated. when painting the last line of a buffer,
Buffer.isFoldStart() doesn't call getFoldLevel(), hence the foldLevelChanged()
event might not be sent for the previous line. */

buffer.getFoldLevel(delayedUpdateEnd);
```

This is a call in the `doDelayedUpdate` method from a class owned by `JEditTextArea` (responsible for editing) to the buffer (jEdit's term for a file). Participants were told that this call was "architecturally questionable" in changing "the buffer's state from a different component" and "clearly bad design" "using a getter method solely to change the state of the buffer and ignoring the information the getter method is supposed to be used to obtain".

Underlying the symptom of the problem (updating fold levels by calling a getter), the cause was the need for a fold update to be triggered from this method (see Figure 5.2. for a call graph of the relevant methods). Participants were left to discover this and why it was bad. Folds are a responsibility of the buffer but the implementation has leaked into another component (`JEditTextArea`) because of this call's presence. Fold levels are lazily computed only when queried by `getFoldLevel`. The call to `getFoldLevel` is required due to this decision (it could be removed if fold levels were not lazily computed) and thus breaks information hiding. In most cases, `isFoldStart` calls `getFoldLevel` and the call from `doDelayedUpdate` is unnecessary. But `isFoldStart` does not call `getFoldLevel` when painting the last line of the buffer because it computes the fold level by comparing the current line's fold level with the next (undefined for the last line in a buffer) and instead always returns false. In this situation, the call to `getFoldLevel` from `doDelayedUpdate` is necessary. Thus, that the presence of the `doDelayedUpdate` call to `getFoldLevel` depends on this very private implementation decision (it would not be necessary if `isFoldStart` were implemented differently), and this also breaks information hiding.

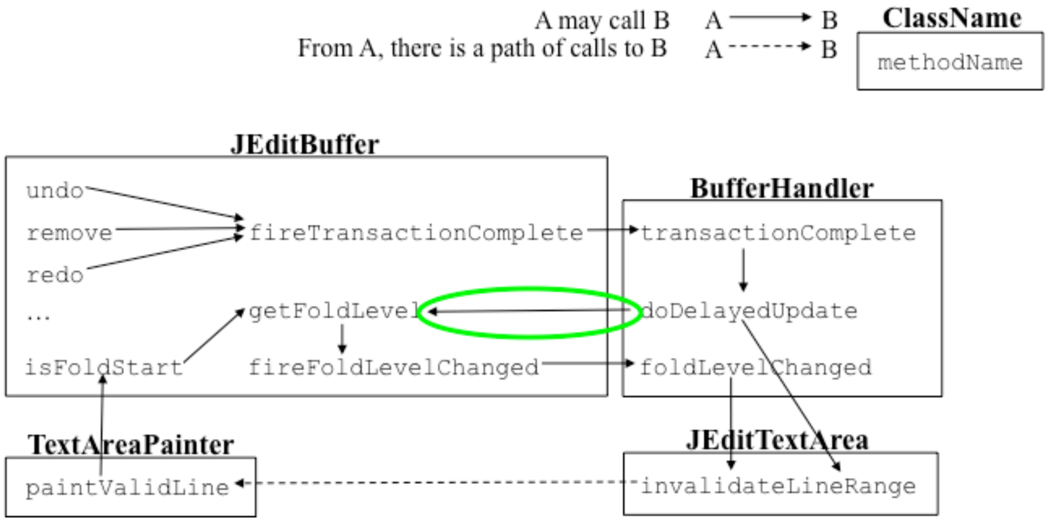


Figure 5.2. Participants, none of whom were given this diagram, spent much of the task reconstructing these call graph relationships and understanding why they were necessary.

The **CARETS** task related to the status bar at the bottom of the jEdit window which displays the line and column of the caret (insertion point) and the scroll position of the window within the buffer. This is implemented, in part, using the `updateCaretStatus` method. Participants were asked to set a breakpoint on `updateCaretStatus`, make the buffer visible in jEdit, and observe that `updateCaretStatus` is called many times. Participants were instructed that this was bad from a performance perspective and "likely reflects deeper problems in the semantics of what the events that trigger these updates mean." The performance critique was contrived in that no extremely resource intensive operations were performed even though methods were needlessly executed. But an expert reported:

> But I've seen this situation before with something that was more directly expensive. – M interview

The **CARETS** task required understanding the design of the buffer switch process. Any action changing either the caret position or the scroll position must call `updateCaretStatus` to update the status bar. Buffer switches change both of these. They begin with a `setBuffer` call. Control then passes through nineteen methods on paths ultimately resulting in 6 or 7 `updateCaretStatus` calls (see Figure 5.4.). Many of these methods are also called for reasons other than buffer switches (including changes in text selection, window scrolling, or caret moves). Removing any calls to `updateCaretStatus` risks breaking these other features.

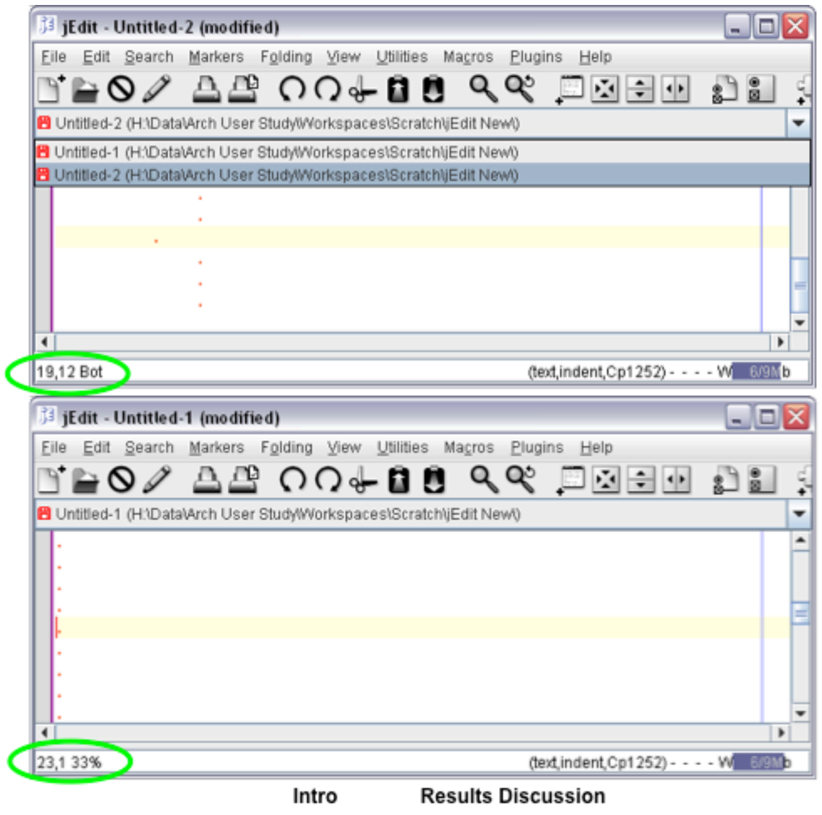

jEdit - Untitled-2 (modified)
File Edit Search Markers Folding View Utilities Macros Plugins Help
Untitled-2 (H:\Data\Arch User Study\Workspaces\Scratch\jEdit New\)
Untitled-1 (H:\Data\Arch User Study\Workspaces\Scratch\jEdit New\)
Untitled-2 (H:\Data\Arch User Study\Workspaces\Scratch\jEdit New\)
19,12 Bot
(text,indent,Cp1252) - - - - W 6/9Mb
jEdit - Untitled-1 (modified)
File Edit Search Markers Folding View Utilities Macros Plugins Help
Untitled-1 (H:\Data\Arch User Study\Workspaces\Scratch\jEdit New\)
23,1 33%
(text,indent,Cp1252) - - - - W 6/9Mb
Intro
Results Discussion

Figure 5.3. The caret position is displayed in the far left edge of `jEdit`'s status bar.

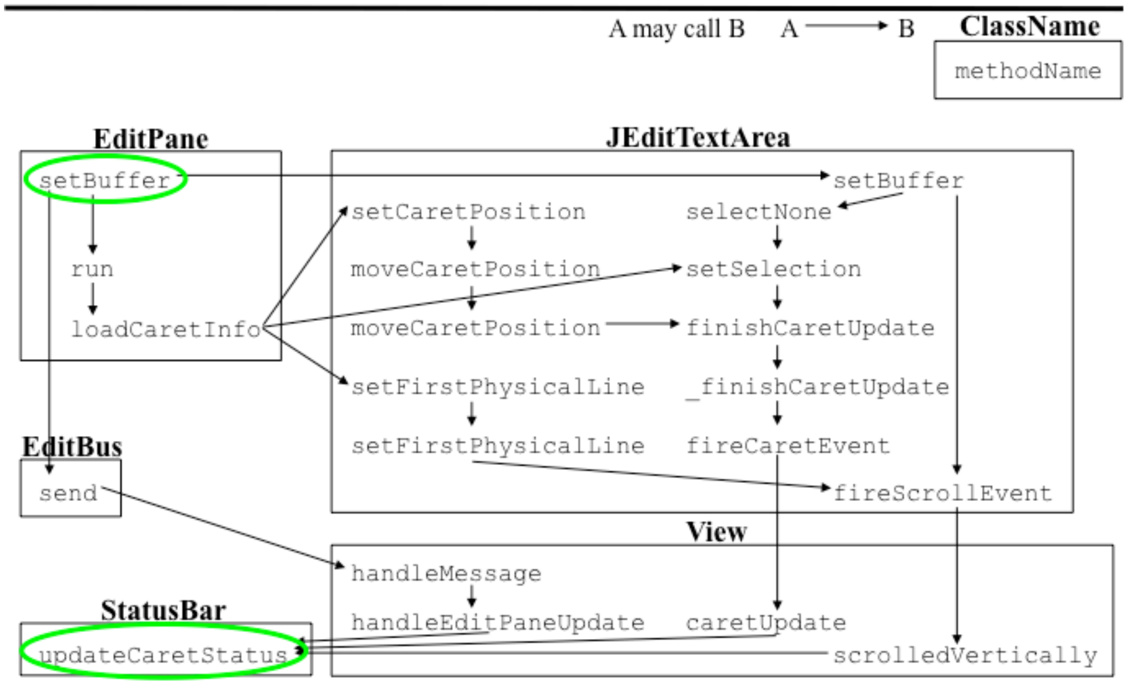


Figure 5.4. 7 paths connect `EditPane.setBuffer()` to the method `Status-Bar.updateCaretStatus()`.

We illustrate our results with think-aloud episodes which we label by participant, time within the task, and task (**C** for **CARETS**, **F** for **FOLDS**)(e.g., M 1:20(**C**) is expert participant "M" at time 1 hour, 20 minutes, doing the **C** = **CARETS** task).

5.1.4. Tools and instrumentation

Participants were provided with the Eclipse 3.2.0 IDE and were allowed to use any Eclipse feature, take notes with Windows Notepad or on a piece of paper, and open files created by jEdit in Notepad or jEdit. To prevent searching for jEdit documentation, bugs, or other information that only some might think was relevant, participants were forbidden from using other applications, including web browsers. One participant asked and was allowed to see the JavaDoc for a collection class in a web browser. The experimenter answered questions about invoking specific Eclipse commands (e.g., how to stop the debugger or that they should use `System.err.println()` rather than `System.out.println()`) or what the task asked them to do, but not any other questions such as questions about the code (e.g., "is my understanding correct?") or strategies about how to use Eclipse to locate information (e.g., "how do I locate a method that triggered an event?").

Participants were recorded using a diverse set of recording devices so none of their actions would be lost. We used Camtasia to record the screen, a video camera of the participant's desk area to track referencing paper handouts and see which area of the screen was being viewed, and a second video camera to track information written on paper. Participants were asked to think aloud and prompted approximately every five minutes if they forgot to do so. Unfortunately, we prompted participants with "what are you trying to do?", leading some to talk more about their goals than the facts they had discovered. In retrospect, a better prompt might have been "what are you thinking about?".

5.1.5. Procedure

Participants first worked through a brief tutorial on Eclipse code navigation features (such as using the call hierarchy, navigating to method declarations, and reference searches) to ensure they effectively used Eclipse. To simulate some of the architectural knowledge that an experienced developer might possess, participants read a one-page description of the responsibilities of eight important task relevant classes. Finally, they worked on a jEdit tutorial where they used the functionality they would be editing so that later testing would be easier. This portion of the study lasted approximately 30 minutes.

Next, participants received a sheet of paper describing the first task. Participants had as much time to read the task description as they liked. Participants then navigated to the code described in each of the tasks. On the **CARETS** task, they also tried out the behavior they were to change by setting a breakpoint and verifying that it was hit many times as the task description claimed. Participants were instructed that they had 1.5 hours to work on each task but were actually allowed up to five extra minutes. Afterwards, participants were asked a series of exploratory interview questions about how they worked, what they found challenging, and ratings of how well they believed they did. Participants then received a clean Eclipse workspace and the description of the second task and began working on the second task. See Appendix 4 for the complete materials.

The tasks were successfully made challenging. While it was expected that some participants would be unable to make meaningful changes, we expected all participants to at least try to understand the code. However, one novice gave up on the **FOLDS** task, and two novices gave up on the **CARETS** task. They felt the code was too complicated for them to comprehend:

> It's too tough for me. I can't figure it out. There's bits and pieces that I understand but I don't understand precisely what the design issue is. – f 1:05(**F**)

An expert thought the **CARETS** task was realistic:

> That is just tough. Yikes, glad I'm not getting paid for this.– M 1:20(**C**)
>
> Yeah, this is realistic. I mean this is realistic on a bad day, at least in my assessment. – M interview(**C**)

Many participants were still working when time expired. Two **CARETS** participants (e, c) elected to describe in notes the list of changes they felt they did not have time to implement.

5.1.6. Analysis

Our analysis started with the low level data we recorded and built successively more abstract representations. We transcribed think-aloud recordings and screen capture video into 26 action logs consisting of a total of 11,821 lines. Every time a participant changed the method or field (referred to as a "member") visible in Eclipse, we added an entry naming the member and Eclipse command used to bring it into view. These included hitting breakpoints, stepping in the debugger, navigating using the call hierarchy or search results, going to declarations, navigating gutter references, and scrolling. We also coded edits, refactor commands, and running the program. We also noted goals participants appeared to be working towards.

Next, we used qualitative protocol analysis. We built a list of activities we saw developers engage in and coded what developers did using this model. Our analysis remained qualitative as we did not produce definitions sufficiently reliable to count and quantitatively compare activities. We discovered that many activities revolved around *facts* about the code. Participants chose methods to read, *seeking* facts they deemed relevant to the task. While reading methods, they sometimes *learned* facts which they believed with varying degrees of confidence. Participants felt some facts violated their design norms and wished to change them. Participants *explained* facts to understand how facts were related and the consequences of changing a fact. This sometimes generated hypotheses which led participants to seek evidence to confirm or reject facts. As participants learned more facts, they began to *propose* design changes that addressed their criticisms and task goals. Finally, participants *implemented* their proposals by editing code. When participants discovered facts leading them to believe their changes would not succeed, they removed the changes and proposed different changes.

Experts nearly constantly talked while most novices said nothing for minutes on end. This suggests that novices were more overwhelmed or spent more time immersed in details. When comparing experts and novices, we chose situations where some experts and some novices said something or situations where we could rely only on observations of what they did.

5.2. Results

We first discuss the changes participants implemented. We then present the model we built to describe how our participants worked. We model developers as seeking facts, learning facts, critiquing facts, explaining facts, proposing changes, and implementing proposals. We consider the structure of each of these activities in turn and differences between experts and novices. Figure 5.5. depicts each of the activities and transitions between the activities.

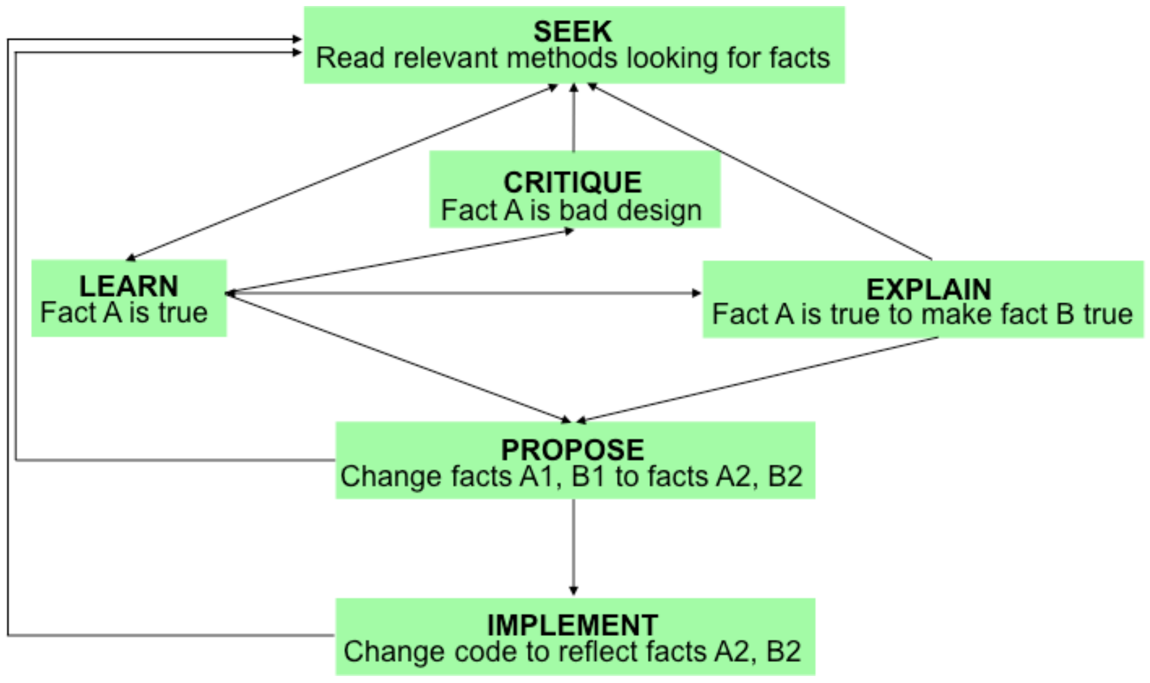

SEEK
Read relevant methods looking for facts
CRITIQUE
Fact A is bad design
LEARN
Fact A is true
EXPLAIN
Fact A is true to make fact B true
PROPOSE
Change facts A1, B1 to facts A2, B2
IMPLEMENT
Change code to reflect facts A2, B2

Figure 5.5. Development activities we observed and transitions between these activities.

5.2.1. Code changes

Changes made by the experts addressed the cause of the underlying design problems. Changes made by the novices (if any) were inferior in that they only addressed the symptoms. We described the underlying design changes ignoring defects they may have introduced or whether they finished. We then clustered similar changes. Table 5.2 lists the final changes (if there were more than one) that the participants implemented or began implementing.

Participant(years industry exp)(time)	**FOLDS task final code changes**
a(0)(1:30)	All getFoldLevel callers check if fold update necessary and conditionally update
b(0)(1:11)	Update folds indirectly by firing the foldLevelChanged event
c(0.5)(1:18)	Renamed getFoldLevel to updateGetFoldLevel
e(2)(0:46)	Do not force fold update
f(2.5)(1:06)	Added debug statement, gave up
d(1.5)(0:44) g(2.5)(1:35) h(2.5)(1:34) i(3)(1:31) j(3)(0:53) K(3)(1:34)	Force fold update by calling method extracted from getFoldLevel
L(10)(1:35)	**Folds updated immediately after buffer changes by call from within JEditBuffer**
M(10.5)(1:14)	**Moved fold update to isFoldStart within JEditBuffer**

Participant(years industry exp)(time)	**CARETS task final code changes**
b(0)(1:34) c(0.5)(1:13) e(2)(1:18)	No changes
a(0)(1:30) d(1.5)(1:15) g(2.5)(1:33) h(2.5)(1:23)	Removed calls believed to be unnecessary.
f(2.5)(1:34)	Added class to log events that happened and detect if caret update should fire
i(3)(0:59) j(3)(1:03)	No changes, gave up
K(3)(1:35) L(10)(1:32) M(10.5)(1:35)	**Added field to stop caret updates during buffer switches**

Table 5.2. Code changes implemented by participants, grouped by change and then sorted by years of industry experience, with total time on task. Changes in bold address the underlying design problem.

On the **FOLDS** task, only the experts worked on changes that would work. One novice made no changes and gave up (f). Another (e) could not determine why the `getFoldLevel` call was necessary and removed it. The remaining novices changed the way in which `doDelayedUpdate` updated folds to address the symptom that a getter was being used purely to set. One (c) renamed the method to `updateGetFoldLevel` to indicate that it was not merely a getter. Another (b) literally interpreted the provided comment to mean that `doDelayedUpdate` needs to send the `fireFoldLevelChanged` event and created a method to do this. Six novices and one expert extracted an update method from `getFoldLevel` and had `doDelayedUpdate` update folds by calling this method. Changes made by two of the experts addressed the cause of the design problem by removing the need for `doDelayedUpdate` to force fold update. One (L) moved the fold update to two methods in `JEditBuffer` which are called after the buffer changes. Another (M) moved the fold update to `isFoldStart` within `JEditBuffer`. Both experts addressed the hidden design problems by removing the `get-FoldLevel` call in `BufferHandler` that added questionable dependencies.

On the **CARETS** task, only the experts worked on changes that would work. Two novices (i, j) made no changes and gave up, and three made no changes but worked for the entire time (b, c, e). One (f) added a class to log `udateCaretStatus` calls with the (mistaken) intention to have it decide if `updateCaretStatus` should proceed from other recent calls. Four novices removed calls they believed were redundant. Expert changes differed from novice changes in starting and stopping caret updates using a field. This approach alone addressed the cause of the design problem in that it could reduce the number of calls to one.

5.2.2. Seeking facts

Participants began their tasks navigating from the methods we provided to methods and fields they believed likely to reveal relevant facts about the code. Participants visited between 5 and 59 members on the **FOLDS** task and 25 and 41 members on the **CARETS** task (Figure 5.6). There was no effect of experience on how many members participants visited, and participants visited similar numbers on both tasks. Thus, experts' superior changes were not due to reading more members but from selecting better members to read and learning more from reading them.

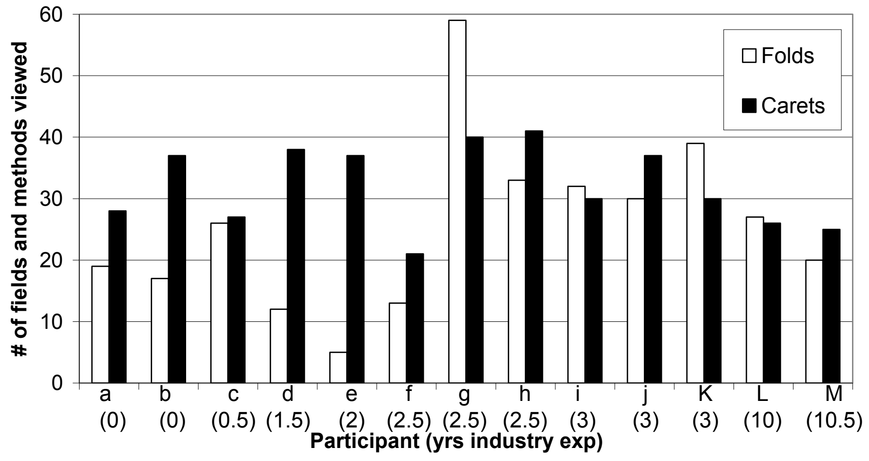


Figure 5.6. Total distinct fields and methods viewed by each participant sorted by years of industry experience.

Participants made *path choice decisions* when choosing between locations in which to seek, deciding if seeking was likely to discover a useful fact, or choosing between seeking and implementing the current change. A novice abandoned seeking in a location:

> So after it runs runnable thread, I get three extra calls to the update caret method. I don't know what thread it is --. I can go in and find out more, but I don't think it is the unnecessary type that I'm looking for. – d 1:09(**C**)

An expert considered whether a change should be implemented or whether better alternatives should first be sought:

> I can reduce the event firing from here, huh, is that even the right path to go down? Let's see, we've got `setCaretPosition`, no, oh wait what about, ohh `setCaretPosition` is the one that is called by many people. Ok, I'm going to give it a long hard look at the `finishCaretEvent`, no `finishCaretUpdate`. – K 0:41(**C**)

We investigated members visited only by novices to understand why novices wasted time visiting members that experts did not need to visit. On the **CARETS** task, there were 12 members visited by no experts which were visited by at least three of the ten novices. One was a class definition, which novices visited more because they used the open class Eclipse command more. 9 were transitive callers of `updateCaretStatus` which novices navigated to more because of inferior navigation strategies. The remaining two members were the most interesting. One was a field – `showCaretStatus` – which guarded the body of `updateCaretStatus`. The other was `propertiesChanged` where `showCaretStatus` was initialized at startup. One novice (f) stumbled into `propertiesChanged` and quickly left it. Four other participants read one or both of these methods because they were interested in the meaning of `showCaretStatus`. One (g) spent a minute looking for `showCaretStatus` references. Another (c) spent two minutes looking at how `propertiesChanged` worked. Two (a, h) spent 7 minutes understanding in detail how `propertiesChanged` worked:

> So I guess the whole debug that is remaining is that when I switch buffers this `showCaretStatus` variable needs to be reset as soon as I update the caret position. – h 0:58(**C**)

Novices seemed to perceive `showCaretStatus` as indicating the presence of a changeable fact that might help them reduce `updateCaretStatus` calls. Reading `propertiesChanged`, they eventually discovered `showCaretStatus` merely controls whether caret and scroll position is displayed on the status bar and would not be helpful for their task. That experts never wasted time reading these members suggests that knowledge helped them guess from the field's identifier and use that `showCaretStatus` did not turn on and off updates during an event but rather in general. This suggests that knowledge helps experts predict what code does before reading it, thereby preventing wasted time reading irrelevant methods.

5.2.3. Learning facts

When reading methods, participants found interesting facts that confirmed or refuted expectations:

> These all look like mutators on the buffer. So that makes sense. So at the end of the mutating operation on buffer, it's going to end in `doDelayedUpdate`. – L 0:06(**F**)

Facts played a variety of roles. Facts were *changeable* when developers believed alternatives to them might help accomplish their task goals. Others acted as *constraints* which suggested that some changes would break them and should not be chosen. Others made changes *expensive* by suggesting lots of investigation would be required:

> Wow, many, many, many methods call `getFoldLevel` and that is not good because it's going to be hard to figure out what all of those are. – K 0:02(**F**)

Facts also differed in the degree of certainty with which participants believed them. Some were *hypotheses* thought likely to be true. Some hypotheses were generated from knowledge about how the application would probably have been built to satisfy its requirements:

> So mouse released represents the bottom of the tree for certain. `setSelectedIndex` is part of `JComboBox.fireAction` event. It's possible that we're getting multiple action handlers involved here, but let's assume that that is not the case. – M 0:16(**C**)

Other facts were directly *observed* in code. But many relied on both observation and knowledge-driven speculation. Figure 5.7 lists some facts found by an expert.

1. HACK: `getFoldLevel` has effects
2. Buffer mutating operations result in a `doDelayedUpdate` call
3. HYP: `doDelayedUpdate` does changes that happen later
4. Many methods call `getFoldLevel`
5. Folds invalidated by buffer changes are updated on screen. EXPLAINS 2, 1, 8
7. `getFoldLevel` updates a fold data structure EXPLAINS 1
8. `getFoldLevel` fires events
10. CRIT: `getFoldLevel` determines if folds must be set
11. CRIT: `doDelayedUpdate` triggers fold update
12. `isFoldStart` calls `getFoldLevel` on startup
13. `getFoldLevel` mutually recursive with `FoldHandler.getFoldLevel`
14. Folds are initialized at startup EXPLAINS 12
15. `BufferHandler` is only buffer listener
16. Either `fireContentInserted` or `fireContentRemoved` is called after every buffer mutating operation

Figure 5.7. Some facts found by expert participant L in the first 41 minutes of the FOLDS task in the order they were discovered. Facts are labeled with hack, hypothesis, and critique roles and the explanation of the relationships.

Experts more frequently and rapidly used facts at higher levels of abstraction which focused on the important and relevant parts of code rather than irrelevant implementation details. For example, experts and novices described `getFoldLevel` very differently. One minute into the task, an expert described `getFoldLevel`:

> Well this is just updating a cache. So, what we're upset about is that you want to issue an event and you are doing it by forcing an update of the cache for the fold level of a particular line. – M 0:01(**F**)

After 38 minutes in the task and 10 minutes reading `getFoldLevel`, a novice still had not figured out how it changed state:

> What it did was it compute I mean computes the new line number and fires an event. But I didn't see it change any state. – b 0:38 (**F**)

51 minutes into the task, after over 12 minutes staring at `getFoldLevel`, and having read numerous callers and callees, a different novice was still stuck at the statement level, never describing it as caching:

> So what it does, it starts off from this line, it has this `firstInvalidFoldLevel`, it goes through all these lines, it checks whether this fold information is correct or not, which is this `newFoldLevel`, this is supposed to be the correct fold level. If that is not the case in the data structure, it needs to change the state of the buffer. It creates this, it does this change, it sets the fold level of that line to the new fold level.
> – h 0:51(**F**)

These differences suggest that schemas, such as caching, allow experts to see design abstractions and chunk individual statements using these schemas. Applying the caching schema helped the expert infer the intent of the code. Lacking the expert's schema, novices were not able to uncover this intent and painfully worked through the code statement by statement**.**

5.2.4. Critiquing facts

Consistent with instructions to improve the design, participants used their good design norms to criticize facts. Growing skeptical of design choices they perceived the original authors had made, they designated those as *hacks:*

> And this guy who is probably hacking away… This started out with this thing as just a getter and said, oh look when you're getting the fold level there can be a case where your data is now invalid so I might as well go fix it up right here. And he might have wandered himself into the bad design situation that we've got right now. – L 0:16(F)

In their criticisms, participants exhibited design knowledge by perceiving a design choice, alternatives, and justifying the inferiority of the current choice. A single expert perceived this design choice:

> And the second thing that I don't like is that it is firing these updates. It seems like when you're making the edit, that in order to keep the responsibilities of these guys very simple, when you're making the edit, the people that care about that would be notified. – L 0:26(**F**)

Several novice criticisms resulted from missing design knowledge:

> It just seems really confusing for me to have this exact same method with the exact same parameters, they both have the `handleMessage`. I should investigate that. Hold on. – c 0:40(**C**)

Only after investigation did the novice realize these methods implemented the same interface.

5.2.5. Explaining facts

Participants explained the rationale of facts they learned:

> So because this is lazily evaluated, which you probably want to do for performance reasons anyway, you're always going to have the risk that a get is going to fire an event in any case. – M 0:09(**F**)

Explanations established traceability from low-level facts about the implementation bottom-up towards motivating requirements, leading developers to hypothesize requirements that motivated the low-level facts they discovered. Dependencies on requirements became important when a participant wished to change a lower-level fact. Because of the dependencies, changing a fact risked changing other facts, potentially breaking requirements. Figure 5.8 depicts the explanations one participant discovered.

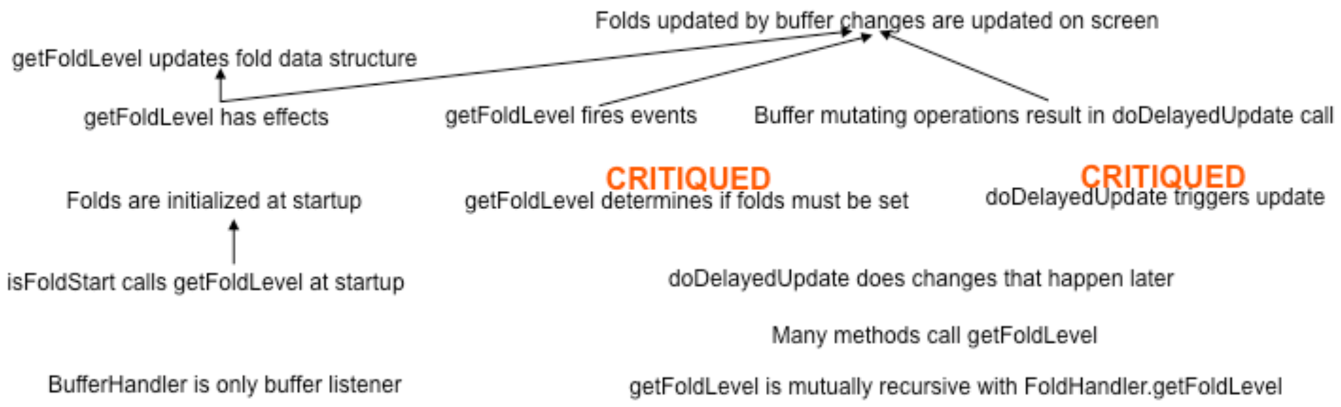

Folds updated by buffer changes are updated on screen
getFoldLevel updates fold data structure
getFoldLevel has effects
getFoldLevel fires events
Buffer mutating operations result in doDelayedUpdate call
CRITIQUED
getFoldLevel determines if folds must be set
CRITIQUED
doDelayedUpdate triggers update
Folds are initialized at startup
isFoldStart calls getFoldLevel at startup
doDelayedUpdate does changes that happen later
Many methods call getFoldLevel
BufferHandler is only buffer listener
getFoldLevel is mutually recursive with FoldHandler.getFoldLevel

Figure 5.8. Explanations and critiques the expert participant L generated for the facts in Figure 5.7.

Participants also applied explanations top down to hypothesize how the code was likely built to satisfy higher level constraints:

> He must be either firing events to tell people to update. Or somehow there must be some other code to then update the display. But it looks like the event firing is happening inside there. – L 0:17(**F**)

When participants believed *false* facts, explanations produced more false facts which were then critiqued or used as constraints. These formed breakdown chains [KM05] where the participant's model of the code had gone badly awry. A developer explained a false call graph fact as due to something triggering a buffer edit:

> 'Cause I'm thinking that when I perform the action of switching from one buffer to another buffer, somewhere it calls a method that indicates that the buffer has been edited. But I didn't edit the buffer. I'm just switching between buffers. So that has to be removed. – d 0:30(**C**)

This hypothesized call from a buffer edit did not exist because the call he used it to explain did not exist.

Participants reasoned using *code* facts which described the implementation and *requirements* facts which described application behavior in terms of the domain. False requirements led to missed constraints. A novice forgot that the instructions stated that the status bar displays caret and scroll position:

> This is, I think this is completely unnecessary because why would a scrolling event cause a caret update. Like if I'm just scrolling, by ---, it doesn't change the caret offset. So I think I should just get rid of this one actually. – d 0:33(**C**)

Participants with a changeable fact that they could not explain faced a choice – optimistically assume it was *overlooked* by the original developer or pessimistically assume it was *intended* to satisfy a hidden constraint. Overlooked facts are true because they happen to be true – changing them does not affect other facts. Intended facts can be safely changed only when the developer is able to generate an alternative fact that still satisfies all the constraints. Optimistic assumptions caused bugs. Pessimistic assumptions led developers to abandon considered changes, *freezing* the fact and preventing consideration of changes:

> So here they're basically deselecting everything and then they're going to reselect everything. So initially I'm going to ignore that because maybe that's intentional by the designer because maybe they would want to if there's an error switching or --- reading from file. – a 1:25(**C**)

Participants investigated hypothesized constraints before concluding none existed and the fact was overlooked.

Some participants used beliefs about the abilities of the original developers to help distinguish intended and overlooked facts. An expert attempted to understand why an original developer had chosen a less desirable decision over an obvious decision:

> Why wouldn't they call it? Now, can I test this? So why if you know the answer to the problem, do you put the code in the wrong place and then leave a comment? That's not like these people. – M 0:35(**F**)

The expert believed the decision could not possibly be overlooked but must be intended, suggesting the search for a hidden constraint must continue. Subsequent discovery of a second example where the original developer overlooked an obviously better decision revised his beliefs:

> What a horrible little thing to do. Ok, that changes my view on the coding style. – M 0:37(**F**)

Participants gambled when deciding if a proposed change would work based on information they did not yet have. Explanations helped predict the probability a change would succeed. One expert implementing a change found it unexpectedly difficult. He became concerned that a fact that he believed to be overlooked was intended and that his work implementing the change would be wasted when they discovered a frozen constraint which had prevented the original developer from making the same change. He presciently predicted, for the wrong reasons, that his 23 minutes implementing the change would be wasted:

> [laughing] This is never going to work, the thing is there's just all this mess going on with this caret listening… If it was just as easy as getting `EditBus` messages and updating the caret it would be straightforward. And the other question I've got, is that there's already `CaretListener`. And why doesn't it just… do caret listening itself? – L 0:41(**C**)

When developers had a hypothesized explanation of the underlying cause, they rejected changes that did not address this cause, even lacking evidence supporting their hypothesis:

> Somehow if I can track it down from the origin of when the event occurs and from there I can pass in a Boolean false to every function call except for one. So it's trickle down... But that seems like a hack because this is called 4 times and it shouldn't be. – j 0:54(**C**)

After proposing several similar changes, he gave up lacking a strategy to check his hypothesis.

In understanding why two experts made different changes than the other participants on the **FOLDS** task, we observed that both better understood why the call was necessary and sought a better way for this constraint to be satisfied subject to their critiques. One expert explained the call using a model of how the application behaved:

> What's going on is that when you're inserting text you could actually be doing something that makes the folds status wrong. So, if in our example here, in the quick brown fox. If fox is under brown and I'm right at fox and I hit backspace. Then I would need to

> update my fold display to reflect the new reality, which is that it's in a different place. – L 0:15(**F**)

No other participant produced this explanation. The expert subsequently mapped specific code locations to serving specific goals and constantly talked about how each of the locations served a purpose in satisfying this requirement. This unique explanation of why the call was necessary allowed him to propose a unique solution – moving fold update from its current, poorly chosen location to a point earlier in the process.

In response to the task description's vague instructions that the `getFoldLevel` call was "architecturally questionable", another expert asked a unique question about `BufferHandler`:

> So what I need to do is figure out how it's using its buffer. Is this the only mutation that they're doing? – M 0:18(**F**)

This generated a unique critique – the `getFoldLevel` call caused `BufferHandler` to retain an "architecturally weird" buffer reference. He then moved fold updating to `isFoldStart` to address this critique.

Three novices who were tantalizingly close to these changes abandoned them following pessimistic assumptions. One novice (g) implemented moving fold update but gave up when he believed a bug indicated he was breaking a frozen hidden constraint. Another (j) tried to explain why the call was in `BufferHandler` by understanding how its parameter was computed until he abandoned this path. Another (h) failed to explain the purpose of `BufferHandler` and felt this hidden constraint made a change too risky. An expert (K) abandoned considering this change when he stated the false hypothesis, without checking it, that `BufferHandler` was intended to change the fold level, rejecting the task's architectural criticism.

5.2.6. Proposing changes

Participants proposed design changes, composed of individual fact changes, to accomplish their task goals and address problems they had perceived. Participants usually first talked about a summary of what they had learned and then proposed a change. Changes often began as vague goals, generating hypotheses, and were then refined by learned facts. One expert (L) proposed six changes in 20 minutes before discovering a change that he believed he had time to implement. Many changes reflected the application of design patterns [GHJ95] they had seen before:

> When I do this, I have two different styles; I have two different methods. So there might be something that directly manipulates a variable and then there's like a publicly visible, sorry if somebody calls like `setX`, I update, send notifications that x has changed or whatever, but if I'm doing something internally I munge, munge, munge and then manually tell people at the end. – L 1:19(**C**)

Novice proposals often did not solve the problem and worked out implementation details rather than considering general patterns:

> How about maintaining for every View, for every buffer, maintaining the caret position in a hashtable. ... The key would be the buffer object and the value would be, say I have the x,y positions of the caret. That's all. I'll have one hashtable, a static hashtable for the application. – e 1:14(**C**).

Of the 29 proposed changes on the carets task, experts (K, L, M) were the only participants to propose using a field to start and stop caret updates. Other proposals included removing redundant calls, passing a Boolean of whether to call `updateCaretStatus` to all of its callers, and recording caret information.

5.2.7. Implementing proposals

We observed one situation where many participants made the same change – 8 participants extracted a fold update method from `getFoldLevel`. Table 5.3 shows that more experienced participants did this more quickly. An expert extracted it merely to better understand it. Other participants intended it as their final change. Participants taking longer appeared aimless or confused, spent tens of seconds staring at code, revisited perceived decisions, visited callees, and moved statements between methods. Participants taking less time recognized the block of code they wanted to extract and used the Eclipse command "Extract Method". This is consistent with a *chunking* interpretation – experts encoded what the code did using more abstract facts. Novices saw the code statement by statement and the interrelationships between statements and got bogged down considering changes at this level.

a(0)	d(1.5)	g(2.5)	h(2.5)	j(3)	i(3)	**K(3)**	**L(10)**
10	13	4	11	9	4	3	4

Table 5.3. Minutes to extract update method from `getFoldLevel` for participants (years industry experience) who tried to do this, sorted by experience with experts in bold.

5.3. External validity

By studying developers in a lab, rather than in the field, participants worked differently in ways which likely made the tasks more challenging. Participants were new to the application and code and could not rely on anything more than the rudimentary information we provided about the design, architecture, and features of jEdit to reason about the application. Participants were asked to make changes designed to require substantial understanding of the design. Developers might typically have much more experience before taking on such changes. Otherwise, such tasks are often used to learn the code with much more relaxed time requirements than our hour and a half tasks. Developers also answer tough questions by seeking out other developers who may know the code better and provide important insights (see Chapter 4). Developers working on code with unit tests might learn why functionality is necessary by commenting it out and finding failing unit tests. By asking participants to make the design as ideal as possible, we may have caused the participants to spend more time or be more careful with the design implications of their changes than they would have otherwise been. But, as our aim was to model the program comprehension process and expertise effects, rather than measure the magnitude of these effects, we do not believe these concerns call into question our findings.

5.4. Discussion

We discovered that program comprehension is driven by beliefs about *facts*. Dependencies between decisions took the form of explanations that developers used to form chains of facts and elicit constraints they would need to respect in their proposals and changes.

A key driver of the program comprehension process was uncertainty. Developers chose how much confidence to express in their hypotheses and made path choice decisions about whether to seek evidence to support them. Developers were uncertain whether a hidden constraint would force them to abandon their changes or unknowingly break a requirement. Developers used sophisticated strategies such as judging the skill level of the original developer to judge the likelihood of a hidden constraint's presence.

An interesting finding is that many of the facts developers thought about took the form of simple predicates about the code. The simplicity of these facts suggests simple analyses could help discover and visualize them. When understanding a method, an expert thought about facts about it (e.g., `getFoldLevel` has effects), explanation relationships with other facts (e.g. `doDelayedUpdate` calls `getFoldLevel` to update folds), critiques (e.g., `getFoldLevel` should not have effects), and design changes resolving these critiques subject to constraints (see Section 7.1.2. for a detailed analysis of these questions). A tool externalizing these facts might help make it easier for developers to remember them and return to the code associated with them. A previous study viewed developers' "working set" of task-relevant facts as regions of code and proposed an editor to externalize these [KMCA06]. We found developers abstractly discussing sets of statements with facts, perhaps because our design tasks focused on constraints while the previous study [KMCA06] focused on changeable facts. When developers focus on facts, not statements, externalized views could be more compact by showing only relevant facts. Design rationale systems have long sought to capture explanations of facts. But these systems were designed to support up-front design in design meetings, and as a result work only with requirements and high-level design [MC96]. A tool that captured explanation linkages might make it easier to find these later (see Section 11.2).

When developers considered alternatives, facts played the role of design decisions. This suggests a measurable definition of information hiding – a fact is hidden during a task when a developer does not think about it. This differs from defining information hiding in terms of methods read by a developer. A developer may hypothesize constraints without ever reading or even locating the code embodying these constraints, but these facts may still profoundly influence design choices. Conversely, a developer may read a method at a high level of abstraction and not notice or consider detailed facts that are explained by the facts of interest (see Section 11.3 for a further discussion of this issue's connection to specifications).

5.5. Conclusions

This study further demonstrates the importance of understanding and exploring code to software development. Developers understanding and exploring code require substantial design knowledge to succeed. Developers lacking this knowledge are unlikely to succeed, and instead became lost trying to make sense of task irrelevant code. In some situations when developers were uncertain of their understanding or erroneously believed false facts, they implemented changes that they later abandoned. Overall, this study demonstrates that challenges understanding and exploring code lead to inadequate fixes, abandoned changes, and time consuming work.

But all of these problems are ultimately caused by developers' inability to effectively answer their questions about code. Code exploration tools that more effectively satisfy developers' information needs might mitigate a wide variety of problems developers face. But building such tools requires a detailed understanding of the information needs to be satisfied. The next chapter addresses this topic by investigating coding activity information needs.

6. HARD-TO-ANSWER QUESTIONS ABOUT CODE[6]

When developers realize they need information, they ask a question. At the highest level, questions may correspond to entire tasks such as debugging (How did this runtime state occur?) or testing (Is this code correct?). When these questions cannot be directly answered, developers decompose high-level questions into lower-level questions, forming a question decomposition graph. Often, lower-level questions may help developers answer many different higher-level questions. Developers might ask "Who calls this method?" when debugging, determining the implications of a change, or determining how code should be tested. Thus while there are debugging questions, implementation questions, and testing questions, many of the questions developers ask can be relevant for all of these tasks.

A better understanding about developers' questions and the challenges developers face when answering these questions would have many benefits. When questions match information provided by tools or language features, this provides evidence that the tools or language features address an important problem. For questions that do not match existing tools or language features, hard-to-answer questions reveal opportunities for new tools and programming languages to solve an important problem that developers face. And when evaluating tools or language features, questions provide a benchmark against which they can be evaluated: does the new tool or language feature help developers more effectively answer these questions?

To identify hard-to-answer questions about code, we conducted a survey of professional software developers. Developers were prompted to report questions about code that they perceived to be hard to answer. We then combined similar questions into a single distinct question and arranged all of the resulting distinct questions into categories. Finally, we looked for existing tools that might help answer these questions.

[6] This chapter based on work previously published in [LM10-2].

6.1. Method

A survey of software developers at Microsoft was conducted as part of the Reachability Survey (see Section 7.2). Approximately 2000 developers were invited to participate by randomly sampling all developers at Microsoft. 460 developers responded. The complete survey included several demographic items, ratings of the importance and frequency of 12 reachability-related questions, and a free response item about other hard-to-answer questions. This chapter focuses on the data from the free response item; results from the rest of the study are presented in Section 7.2. 179 developers completed the free response item; 149 were individual contributors, 22 were lead developers, and 8 were architects. Respondents ranged in development experience from 3 months to 39 years (median 10 years) and had spent from 0 to over 8 years working on their current codebase (median 1 year). 68% agreed that they were “very familiar with my current codebase”. Respondents reported their team was currently in a variety of life cycles phases: 19% planning, 33% implementation, 42% bug fixing, and 6% other. Respondents reported spending anywhere from 0 to 100% of their time editing, understanding, or debugging code (median 50%).

Following completing the first section that focused on reachability questions, developers answered a free response question: “What other hard to answer questions about code have you recently asked?” Responses included both questions and stories illustrating factors that developers perceived to have made these questions challenging. We analyzed responses by breaking them into individual questions, yielding 374 questions. We then clustered the reported questions into categories using the underlying intent of the question – what did developers want to know by answering the question? For example, “Why did this happen?” was usually a question about runtime behavior, not rationale. Finally, within each category, similar reported questions were grouped into distinct unique questions.

We also evaluated the match between the questions that the developers reported were hard to answer and the information that tools can help developers to obtain. For each question, we considered both commercial tools and research tools that might be relevant to the question, listing commercial tools, if present, and research tools if commercial tools were not available. A match only indicates that the information provided appears to align with a question and does not indicate that the tool successfully helps the developer to obtain the information more effectively. Thus, our goal in listing the tool is to identify to what extent a tool *exists* that *can* address this information need – not to what extent that tool is usable by or available to any programmers.

6.2. Results

Developers reported 92 distinct hard-to-answer questions. We grouped these into 18 different categories spanning three topics: changes to code, properties of code elements, and relationships between elements. We discuss each of these topics and categories below. When reporting question developers ask, we indicate the number of developers that reported each question (or version of a question) with the number listed in parenthesis. Questions reported by previous studies of developer questions are indicated with a citation. Questions for which we identified a corresponding tool are indicated with the name of the tool or category of tools or "unsupported" if we were unable to identify any such tools.

6.2.1. Questions about changes

As developers work to implement features and fix bugs, they work with changes, past, present, and future. Developers reported a variety of hard-to-answer questions about changes.

Debugging (26)
How did this runtime state occur? (12)[KDV07]*:* omniscient debuggers, dynamic slicing
What runtime state changed when this executed? (2): REACHER
Where was this variable last changed? (1): dynamic slicing
How is this object different from that object? (1): unsupported
Why didn't this happen? (3): Whyline
How do I debug this bug in this environment? (3): statistical debugging
In what circumstances does this bug occur? (3)[KDV07]: statistical debugging
Which team's component caused this bug? (1): unsupported

Developers faced with unexpected runtime behavior ask hard-to-answer questions about why or how some runtime state did or did not occur. Runtime state included changes to data, memory corruption, race conditions, hangs, crashes, failed API calls, test failures, and null pointers. Debugging questions are caused by past changes that do not work as expected. Developers wondered about differences between executions – e.g., why functionality did not work on some browsers – or the circumstances necessary for the bug to occur. Crash dumps and environments that could not be recreated locally were particularly challenging to debug. Debugging did not always involve preparations to devise a fix – sometimes developers wanted simply to trace the flaw far enough to understand which team should be assigned the bug and understand its ultimate cause.

Many of the debugging questions developers reported are addressed by existing research tools. Omniscient debuggers record and play back traces, letting developers reverse execution and go back in time. This may make answering "How did this runtime state occur?" easier, as developers can then follow dependencies back in time without having to repeatedly rerun the program. Dynamic slicers, such as the Whyline [KM08], let developers traverse dependency chains backwards, letting developers directly answer questions about where a variable was changed and how runtime state occurred. The Whyline directly supports determining why something did *not* happen by identifying branches that would have caused something to happen but were not taken, and identifying reasons why these branches were not taken. REACHER helps developers answer questions about what state has been mutated when something executes by statically searching for writes to fields.

Research tools also address problems developers reported about understanding and recreating the circumstances in which a bug occurs. Statistical debuggers [LAZ03] first collect data from users' computers by sampling executions. Using this data, they then find correlations between bugs and execution trace properties. Using a statistical debugger, developers could understand the circumstances in which a bug occurs by its correlation to properties of the execution trace and, if this provided sufficient information to debug, would not be required to determine how to recreate the environment.

Another debugging challenge developers experience is triaging bugs. After a bug is reported, triagers must determine who will be assigned to fix the bug. Triage involves determining which team's components are responsible. Bugs involving a crash with a stack trace or incorrect behavior in a particular part of the interface may have a symptom located in an easily determined component, but the ultimate cause may or may not be in that component. And determining which component is ultimately responsible may involve as much work as the debugging itself. Thus, the developer triaging cannot assign it to the correct team until they have themselves done much of the debugging work. While automated tools have explored machine learning techniques that use the bug report text and ownership of components to assign bugs to team members [AHM06], these tools face the same challenges as in manual triaging. These tools are unlikely to be any better determining which component ultimately caused the bug.

Implementing (20)
How do I implement this (8), given this constraint (2)? (10): Blueprint
Which function or object should I pick? (2): Jadeite
How overloaded are the parameters to this function? (1): autocomplete
What's the best design for implementing this? (7): unsupported

Developers with partially formed ideas for a change ask hard-to-answer questions about how to implement it. Changes include connecting together components, integrating code, reusing a library, determining how to correctly set a field of a shared data structure, implementing tests, editing protocols, and changing exception policies. In some cases, developers seek implementations subject to constraints such as API backwards compatibility. When reusing functionality, developers ask about differences between similar methods or objects to decide which to pick. Finally, developers weigh design quality tradeoffs – where should functionality be located between callers or callees or in classes or layers.

Developers trying to determine how to implement functionality have a variety of resources to consult ranging from tutorials, to pattern catalogs, to API documentation. Most modern development environments provide an autocomplete mechanism which lists alternative implementations of a method with different combinations of parameters. Research tools have also been designed to help support implementation. Given keywords describing relevant functionality, Blueprint searches the web for relevant example code snippets [BDW10]. However, Blueprint requires the snippets to be on the web, which prevents its applicability to situations involving reusing functionality within a large codebase, which were the primary situations developers reported in our study. Jadeite helps developers determine which class to pick by using web counts to rank classes by popularity [SFY09]. However, Jadeite does not help select methods.

Implications (24)

What are the implications of this change (5) for API clients (5), security (3), concurrency (3), performance (2), platforms (1), tests (1), or obfuscation (1)? (21) [KDV07][SMV08]: unsupported

When proposing a change, developers ask questions about its effects to determine constraints that should be respected, other changes that might be necessary, or if the change is worth making (see Chapter 5). Implication questions are a step further into implementation. Developers ask implementation questions (e.g., "How do I implement this (given this constraint)?") when they have a goal but no potential implementations. Developers ask implication questions when they have a potential implementation but are unsure if it will work. When changing functionality exposed to other components or other teams, developers consider if a change could cause bugs elsewhere and if the old behavior must be maintained alongside the new. Developers consider what issues a change might cause such as security concerns, timing issues like deadlocks, or performance effects to execution time or network or disk usage. Developers also report questions about dependencies on code or design decisions, which ask about implications and code affected for any possible change.

Tools provide little, if any, direct support for answering implication questions. Perhaps the closest tools are in the area of impact analysis [L11]. Given an element to be changed, impact analysis tools attempt to predict other elements that must also be changed using a variety of techniques such as slicing, information retrieval, or code history. But impact analysis, even if its predictions are successful, only predicts elements. In contrast, implication questions dealt with properties and facts about code. Seeing a potentially related element seems unlikely to answer these questions.

Policies (12)

What is the policy for doing this? (10) [SMV08]: unsupported

Is this the correct policy for doing this? (2) [KDV07]*:* unsupported

When designing a change, developers ask questions about relevant precedents or policies such as when resources could be freed, design pattern use, security, configuration settings, error logging, exceptions, versioning, installation infrastructure, and expected public APIs.

Developers use a number of practices that may help answer questions about policies. Design documents and coding standards often capture high-level ideas about how something might work. Or developers might simply ask a teammate (see Chapter 4). But documents may or may not capture the policy of interest, can be hard to locate, and are often not updated (see Chapter 4), and asking teammates causes interruptions, a problem by itself. So developers are often left to reverse-engineer policies from the code or guess. Research and commercial tools have not investigated approaches for improving this situation.

Rationale (42)
Why was it done this way? (14) [FM10][KDV07]: unsupported
Why wasn't it done this other way? (15): unsupported
Was this intentional, accidental, or a hack? (9) [KDV07]: unsupported
How did this ever work? (4): unsupported

Rationale questions ask about the rationale for surprising design decisions found in the code or in previous changes, often to discover hidden criteria motivating a decision (see Chapter 5). Developers gave many examples of design decisions with unclear rationale including naming, code structure, inheritance relationships, where resources are freed, code duplication, lack of instrumentation, lack of refactoring, reimplementing instead of reusing, algorithm choice, optimizations, where behavior is implemented, parameter validation, visibility, and exception policies. In some cases, developers had an alternative design choice in mind and wondered what hidden design criteria caused it to not have been chosen. Sometimes, developers also considered if the decision might have been a hack made in haste or was simply overlooked, rather than being a carefully considered judgment reflecting a deeper understanding of the problem ([KDV07]; Chapter 5). In other cases, developers seemed surprised and nearly convinced that the original decision was erroneous (How did this ever work?).

Developers have a number of strategies they might use to answer rationale questions. In some situations, developers ignore the original rationale, implement a change, and test if it works. But, for many of the non-functional properties developers reported (naming, code duplication, lack of instrumentation), testing is not an option. As with policies, developers can ask teammates; but this causes interruptions (see Chapter 4), blocks the question asker when the teammate is unavailable, and does not work when the responsible party has left the company. And, of course, comments and design documents can help, but require future questions to be anticipated, the comments to be correctly updated, and the author to invest the time to write them.

Efforts to build research tools that help developers more effectively answer rationale questions have been limited. Several systems explicitly capture and store rationale, but focus on high-level decisions earlier in the life cycle rather than code-level design decisions [MC96]. In some cases, developers answer rationale questions by browsing history to look for relevant check-in comments or changes that might have motivated unexpected behavior. Research tools do exist for browsing code history (see next paragraph).

History (22)

When, how, by whom, and why was this code changed or inserted? (13)[FM10]: Deep Intellisense
What else changed when this code was changed or inserted? (2): Deep Intellisense
How has it changed over time? (4) [FM10]*:* Deep Intellisense
Has this code always been this way? (2): Deep Intellisense
What recent changes have been made? (1) [FM10][KDV07]: Deep Intellisense

As developers browse past changes to a code snippet, they ask questions about a change's author, date, identity, and motivation. Developers most frequently reported difficulties answering code history questions about code snippets, confirming a previous finding that developers are most interested in the history at the level of code snippets [HB08] rather than larger units such as files or modules. Sometimes they are interested in seeing how a change to a snippet of interest is part of a larger change, perhaps to understand its motivation and scope. In some situations, developers simply wish to know if code's current implementation is the only one that had ever existed, perhaps to see if alternatives had been previously considered or if past differences necessitated a different implementation.

Most modern version control systems provide facilities for browsing code history. But many only support browsing files, forcing developers to manually browse all changes to a file to find those changes affecting the snippet of interest. Research tools have addressed this limitation. For example, Deep Intellisense lets developers select a specific element and see history of that element [HB08]. It is unclear what other challenges or barriers developers experience when browsing the history.

Refactoring (25)

Is there functionality or code that could be refactored? (4): unsupported
Is the existing design a good design? (2): unsupported
Is it possible to refactor this? (9): unsupported
How can I refactor this (2) without breaking existing uses (7)? (9): refactoring tools
Should I refactor this? (1): unsupported
Are the benefits of this refactoring worth the time investment? (3): unsupported

Developers refactor to improve the design of existing code. As developers refactor, they have a variety of information needs. First, developers must identify functionality or code with design problems that might benefit from refactoring. Next, developers consider the quality of the existing design, ask if it is possible to refactor, and if so, how? As they generate and consider alternative designs, developers consider how existing uses of methods constrain changes to these methods. Finally, developers weigh the costs and effort of implementing the refactoring against its benefits. Developers reported a number of refactorings they found to be challenging: changing a method's scope, moving functionality between layers, changing the implementation of configuration values, making operations more data driven, or generalizing code that would be more reusable.

A variety of tools help support parts of the refactoring process. Developers refactor in order to address design problems, and research tools have been designed to identify design problems that researchers hypothesized would be important to developers. Clone detectors such as CCFinder [KKI02] detect syntactically similar code snippets, which according to software engineering principles, are code clones that should be refactored into an abstraction. Smell detectors (e.g., [MB08]) detect design idioms believed to signify bad design (e.g., feature envy, typecasts, magic numbers, large classes). However, none of these tools address the types of design problems that developers reported: obsolete code, duplicated functionality, and redundant data between equally accessible data structures.

Modern development environments automate some refactorings by letting developers invoke and describe the expected change, checking for necessary preconditions, and changing the code. However, support is limited to low-level refactorings (e.g., moving code between methods or renaming methods). In contrast, developers reported challenges making larger refactorings (e.g., making operations more data driven, or changing semantics of config values). While lower-level refactorings might be involved in larger refactorings, existing tools do not directly support the larger refactorings.

No existing tools help developers to predict the time to refactor. Tools to reduce the cost of refactoring might indirectly address this issue by making it less of an issue. But the most costly refactorings are likely to be the largest, which are currently not addressed by existing tools.

Testing (20)
Is this code correct? (6) [KDV07]: testing and verification tools
How can I test this code or functionality? (9): testing and verification tools
Is this tested? (3): coverage tools
Is the test itself responsible for this test failure? (1): unsupported
Is the documentation wrong, or is the code wrong? (1): specification checkers

Developers work to assure that their code is correct and ask testing questions. Developers evaluate if their code is correct – if it did what the comments imply or the callers expect, if it works in situations with multiple users or servers, or if it has security vulnerabilities. Developers ask how to test code which depends on an external API or with error paths and how to check for memory leaks, race conditions, or hangs. Developers consider if existing unit tests already exercise functionality or codepaths. After discovering a problem, developers want to know if the test, documentation, or code is at fault.

Developers' reported testing questions are well addressed by existing research. To help understand if code is correct and make testing it easier, a variety of research and industrial tools exist to test and verify code using a variety of program analysis techniques. Tools even exist to automatically test or verify the specific hard-to-test situations reported – memory leaks (e.g., [E03]), race conditions (e.g., [AFF06]), and hangs (e.g., [CGP07]). A variety of commercial tools exist for determining the code coverage of tests. And, when documentation on a class or method has been specified using a specification language, verification and testing tools can check if the code conforms to the specification. But one question that has not been addressed by existing research is attributing blame: when a test or specification fails, is the test wrong or the code? This ultimately requires a developer to make a judgment about what the correct behavior should be. But in doing so, the developers' work in understanding the code, requirements, or conventions might be supportable by tools.

Building and branching (12)
Should I branch or code against the main branch? (1): git
How can I move this code to this branch? (1): merging
Have changes in another branch been integrated into this branch? (1): notifications
What do I need to include to build this? (3): include generation
What includes are unnecessary? (2): include generation
How do I build this without doing a full build? (1): incremental compilers
Why did the build break? (2): [FM10] logging
Which preprocessor definitions were active when this was built? (1): preprocessor flags

Developers use version control systems to track and coordinate checkins and use branches to let them work on changes separately. Developers question when creating a branch is necessary. And after finishing work, developers ask how their branch can be merged into the main branch. Developers also wish to stay aware of other developers' work and ask if developers' changes in other branches have been merged back. Developers use build systems to manage inclusions, preprocessing, and other dependencies. As developers copy and migrate code, they ask questions about what includes or dependencies to add; when inspecting code, they ask which includes are no longer required. Developers want to determine the minimal number of packages necessary to rebuild, rather than trigger a time-consuming full build. When faced with an intermittently breaking build, developers debug and understand its cause. After building, developers wonder what the preprocessor has done – which definitions are active.

A variety of tools help developers to answer questions about building and branching. One solution to the problem of deciding *when* to use a separate branch is to *always* use a separate branch. The increasingly popular version control system git makes this choice, and gives each developer their own branch. There is no main branch.

Version control systems support moving code between branches through merges. However, as the developers reporting this question almost certainly had access to such features, we can conclude that it can require substantial work for the developers to determine which parts of code must be moved to integrate a change or, when merge conflicts are present, how to resolve them. Many version control systems also provide notifications through email of changes.

Modern IDEs such as Eclipse automatically generate include statements, requiring the developer only to correctly configure library dependencies and resolve ambiguous references.

As code changes, IDEs can update the include statements to remove statements that have become unnecessary. Similarly, modern languages often make possible efficient incremental compilers that rarely require full builds.

Developers working with builds also debug. When faced with a broken build, most build systems provide log messages explaining the failure. However, beyond these logs, developers are left to debug and resolve the issue on their own. Similarly, preprocessors often provide flags for logging their generated content. But, as the developers who reported problems likely had such features, there may remain significant challenges using these tools to debug build problems.

Teammates (16)
Who is the owner or expert for this code? (3) [BPZ10][FM10]: expertise recommenders
How do I convince my teammates to do this the "right way"? (12): unsupported
Did my teammates do this? (1): check-in logs

When trying to understand unfamiliar or complicated code, developers often try to find an owner or expert responsible for past changes to the code. But, in other situations, interactions with teammates are more contentious. Developers reported frustration at teammates for not doing things the "right way" by following conventions or coding styles, such as using a C style in C#, using an outdated style, or writing hacks.

Expertise recommenders help developers to identify code owners and experts, often by mining version control information (e.g., [BPZ10][MH02]). Helping developers to find consensus and agreement with conventions and standards is an open, unexplored area of research.

Check-in logs record the activities of software developers as they change code and can be indexed and searched. Logs might be able to answer questions about what teammates have done. However, for the developer who viewed this as poorly supported, it might not contain the information about what teammates were doing that was of interest. Better understanding how developers coordinate and share information within teams is an important area of research.

6.2.2. Questions about elements

As they do work and ask higher-level questions, developers decompose many of these questions into lower-level questions about elements and sections of code. Developers reported a variety of hard-to-answer questions about elements and sections.

Location (13)
Where is this functionality implemented? (5) [SMV08]: feature location
Is this functionality already implemented? (5) [KDV07]: unsupported
Where is this type defined? (3): IDE navigation support

When developers fix or change functionality, they must first locate it. Developers reported this as being a hard-to-answer question. When planning to implement functionality, developers ask if the functionality is already implemented. And, when seeing references to types, developers sometimes asked where it is defined. This can be challenging when it requires picking the correct definition referenced by include files, understanding type renames done at compilation, and navigating between multiple files defining the same class.

Research tools have long considered how to help developers find where behavior they see at runtime is implemented in code (e.g., [WS95][KM09]). But developers also wish to locate functionality in a codebase to reuse based on what it does. This is unsupported by existing tools.

Modern IDEs provide support for navigating to class definitions and even support selecting implementations of an interface. Developers who reported challenges navigating to definitions may have not had access to such tools or might be using a language where the preprocessor or other issues prevent the use of such tools.

Performance (21)
What is the performance of this code (5) on a large, real dataset (3)? (8): profilers
Which part of this code takes the most time? (4): profilers
Can this method have high stack consumption from recursion? (1): profilers
How big is this in memory? (2): profilers
How big is this code? (1): metrics
How many of these objects get created? (1): profilers
Is this method or code path called frequently, or is it dead? (4): profilers

Developers reported hard-to-answer questions about the performance characteristics of code. Developers sought to localize poor performance to specific code to understand where improvements should be made. This was particularly challenging when the hotspots in the optimized version shipped to users differed from the hotspots in the debug build used for profiling. Developers also sought to understand the memory usage of stack allocations done in recursive methods and the number and size of objects created.

Profilers provide performance data about code including execution time, hotspots, memory usage, and object creation counts. It is unclear whether the developers who reported challenges with these questions did not have access to these tools, experienced issues that prevented their use, or that they simply were not as effective as developers would like. A wide variety of commercial and open source tools provide metrics about code, including its size.

6.2.3. Questions about element relationships

Many of the developers' reported questions dealt with relationships between code elements: contracts describing interactions between methods, how threads, methods, and locks interacted in concurrent situations, and how control and data flow connects pieces of code. These questions require developers to determine relationships between code and how these interactions are necessitated by design decisions and code's intended behavior.

Contracts (54)
What is the intent of this code? (12) [KDV07]: contracts
What does this do (6) in this case (10)? (16) [SMV08]*:* REACHER
How does it implement this behavior? (4) [SMV08]: unsupported
What assumptions about preconditions does this code make? (5): contracts
What assumptions about pre(3)/post(2)conditions can be made? (5): contracts
What exceptions or errors can this method generate? (2): checked exceptions
What throws this exception? (1): checked exceptions
What is catching this exception? (1): unsupported
What are the constraints on or normal values of this variable? (2): invariants
What is the correct order for calling these methods or initializing these objects? (2): typestate
How is the allocation lifetime of this object maintained? (3): typestate
What is responsible for updating this field? (1): unsupported

Developers reported a variety of questions about code's intended behavior and realization of this behavior. Developers ask questions about the intent of code, SQL queries, structures, objects, files, and components. Developers ask what code actually does, often about its behavior in a specific situation such as an exception or error, a slow or timed-out operation, in the presence of multiple threads, how it behaves at boot up, or how it executes on a server farm. From expectations of code's behavior, developers try to understand how code realizes this behavior in its implementation: how optimized code implements an algorithm, how an exception could be thrown, how binding works, or how a class implements application functionality.

Developers also reported questions about assumptions and expectations about a method's input and output. Developers consider both assumptions currently made about parameters and possible additional assumptions that could (safely) be made. Assumptions included constraints on parameters such as ordering or size and the possible states a program might be in. Developers also reported questions about the errors or exceptions that might be generated and the conditions under which they are generated. Finally, developers wanted to understand the meaning and semantics of parameter values: e.g., if parameters were 0 or 1-based indices and what constitute typical values.

Developers also reported questions about the correct protocol or order in which methods should be called. One situation where this was an issue was for object lifecycles: how objects were initially created and how they would be destroyed after they were no longer needed. Finally, a developer reported difficulty understanding what code was responsible for ensuring a field was correctly updated.

Software engineering researchers have long designed systems for specifying and checking contracts, dating back to early efforts such as Hoare logic [H69] and culminating in modern tool efforts such as JML [LC06] and Code Contracts [FBL10]. For expressing and dynamically checking contracts, assertions are widely used in practice and even built into modern languages such as Java. Specifications document the assumptions about preconditions currently made. Adding an additional specification, and using a tool to check if it holds, can answer questions about what additional preconditions are possible. Invariants specify relationships between variables and could be used to specify, for example, that an index value is always greater than 0 or 1. Typestate [SY96] captures specifications about the order in which methods should be called. Checked exceptions specify the exceptions generated by a method [G75]. REACHER helps developers understand what code does, letting developers search for specific effects (mutating a field) or calls into libraries. But REACHER does not support filtering searches to specific situations (see Section 11.1.1 for a discussion of how REACHER could be extended to support these questions).

What have not been addressed by existing tools are questions regarding how a specification is realized in code. Research tools have not explored supporting questions about how code implements behavior or what code is responsible for updating and maintaining the correct values of a field.

Concurrency (9)
What threads reach this code (4) or data structure (2)? (6): thread coloring
Is this class or method thread-safe? (2): specifications
Which methods in this class does this lock protect? (1): references search

When reasoning about the concurrency of code, developers report asking about what threads are able to reach an area of code or to access a data structure. Developers also ask if classes or methods have already been designed to ensure thread-safety and which of the methods in a class are protected by a lock.

Many tools have been designed to help developers reason more effectively about concurrency and address all of the questions developers reported. Thread coloring documents the threads expected to reach a method with a specification and checks that the specification is accurate [S08]. Modern languages such as Java have keywords and annotations that document the thread-safety of classes and methods. And a references search will show the methods in which a lock is referenced, which indicates the methods it is likely protecting.

Control flow (13)
In what situations or user scenarios is this called? (3) [KDV07][SMV08]: REACHER
What parameter values does each situation pass to this method? (1): unsupported
What parameter values could lead to this case? (1): dynamic symbolic execution
What are the possible actual methods called by dynamic dispatch here? (6): code browsing tools
How do calls flow across process boundaries? (1): instrumentation tools
How many recursive calls happen during this operation? (1): profilers

Developers report asking questions about control flow such as the situations or user scenarios in which methods are called, the parameter values used in different situations, and what parameter values were necessary to reach specific code within a method. Determining calls from a method is made more difficult by dynamic dispatch – calls to interfaces, signaling events, or changing bound properties. Other challenging issues developers reported were understanding control flow between multiple processes and understanding recursive calls.

Modern development environments provide support for browsing control flow interprocedurally. IDEs such as Eclipse let users navigate to the definition of a method; for calls to interface methods or overridden methods, a list of implementing methods is shown, directly showing the developer the possible dispatch targets. Profilers can show what recursive calls are made. And logging and instrumentation tools can be used to track control flow across processes (e.g., Dtrace [CSL04]).

Recent research tools are designed to support some of the other control flow questions developers reported. REACHER lets developers search upstream from a method, making it possible to search for methods that are framework callbacks which respond to user input. To find parameter values which lead to paths through the code, dynamic symbolic execution tools can be used, which attempt to force execution through all paths and collect representative parameter values for each path (e.g., Pex [TH08]). But tools have not been designed to collect and group values passed into methods.

Data flow (14)
What is the original source of this data? (2) [KDV07]: thin slicing
Where can this global variable be changed? (1): references search
What code directly or indirectly uses (6) or modifies (2) this data or resource? (8): unsupported
Where is this data structure used (1) for this purpose (1)? (2) [SMV08]: references search
What parts of this data structure are modified by this code? (1) [SMV08]: REACHER

Developers reported questions about data flow through code – where data originates, where it goes, and how it is aggregated, translated, or transformed. Developers ask about both where a variable is modified and where data referenced by a variable is modified. Finally, developers reported questions about relationships between specific data and code – modifications to a data structure and use of resources by a specific part of code.

Several tools help support developers' reported data flow questions. Modern development environments let developers search for references to find places where a global variable or data structure is referenced. However, these tools do not track the data as it is copied from variables and cannot be directly used to find code that uses data referenced by a variable. Thin slicing follows data flow relationships backwards and can identify the original source of data; but the tool does not support following data forwards to see the uses of data. REACHER can search for writes by code to a data structure by searching for a class (data structure) and scoping the search to field writes, indicating the parts (fields) of the class which are modified. However, REACHER cannot scope its searches to specific instances of a data structure.

Type relationships (15)

What are the composition, ownership, or usage relationships of this type? (5) [SMV08]: UML tools, ownership systems
What is this type's type hierarchy? (4) [SMV08]: code browsing tools
What implements this interface? (4) [SMV08]: code browsing tools
Where is this method overridden? (2): code browsing tools

Developers reported a variety of questions about type relationships: the composition, ownership, and usage relationships of types, super and subclasses of a type (its hierarchy), classes implementing an interface, and places where a method is overridden. Answering questions was made more challenging by classes spanning modules or projects that were not open in the development environment.

Modern development environments provide code browsing tools that show type hierarchies and overrides. UML reverse engineering tools can generate class diagrams from code that depict composition and usage relationships between types [KSS02]. Research systems can specify and check ownership relationships between types (e.g., [DM05]). However, such tools do not work in all situations. Some developers reported that these questions were most challenging for projects or modules not open in the development environment, which might not be indexed.

Architecture (13)

How does this code interact with libraries? (4): REACHER
What is the architecture of the code base? (3): reverse engineering tools, architecture description languages
How is this functionality organized into layers? (1): reverse engineering tools, architecture description languages
What depends on this code or design decision? (4)[FM10]: impact analysis, design structure matrices
What does this code depend on? (1): design structure matrices
Is our API understandable and flexible? (3): unsupported

Developers reported a variety of questions about the architecture and relationships between modules. Developers ask questions about how code interacts with libraries, how it is organized into layers, and incoming and outgoing dependencies. Developers also found it challenging to design code for reuse and ensure that an API design was both understandable by its users and flexible in supporting its potential uses.

A variety of commercial and research tools help support answering questions about architecture, for example by reverse engineering class diagrams or sequence diagrams [KSS02]. Other tools, such as architecture description languages, let developers specify and document an architecture for the code. To understand how code is interacting with libraries, REACHER lets developers search for calls to library methods and visualize these interactions in a call graph. Impact analysis tools seek to find code's dependencies [L11]. Design structure matrices express dependencies between decisions [SGW01], but tools to reverse engineer them from code (e.g., [SJS05]) are limited to dependencies between packages and call relationships between methods, which may not include the dependencies of interest. While user studies can help to evaluate the understandability and flexibility of APIs, there is no tool support for conducting such studies, although it is unclear what support might be provided by tools.

6.2.4. Summary of tool support

To assess the current state of industrial and research tool support for helping developers answer questions about code that they find to be hard to answer, we counted the questions for which there exists a research or commercial intended to help. For the 92 distinct questions developers reported, 47% are addressed by commercial tools and an additional 27% are addressed by research tools. This leaves 26% that have not been addressed by any tools. Looking instead at percentages of the 374 reported questions (where some of the 92 distinct questions were reported multiple times), 34% are addressed by commercial tools and 25% by research tools. 41% of developers' reported questions are unaddressed by any existing tools.

6.3. Discussion

Our survey revealed the huge scope of challenges that developers face when working with code. Developers, asked to report questions they found to be hard to answer when working with code, responded with questions spanning most areas of work with code that software engineering has studied. These results demonstrate that hard-to-answer questions about code are not localized to a small number of tasks or activities, but span a wide variety of developers' work.

But what does it mean for a developer to have reported a question as being hard to answer? Such questions are likely to reflect some of the most salient, challenging, time-consuming, and frustrating programming episodes that developers can recall. From the developers' point of view, their current tools did not make answering these questions sufficiently easy.

There are a number of potential reasons why developers might feel that the tools they use today do not support these questions. Developers might not be aware of features in their tools that better support answering these questions. Developers might choose to use tools or development environments that do not have features other tools have that support these questions. Many developers still choose to use code editors such as vi and emacs (see Chapter 4), and as a result, may have issues answering questions such as "What are the possible dynamic dispatch targets?" that modern development environments directly support. Other developers may be forced by legacy code or corporate standards into using outdated tools.

While developers using Java and Eclipse benefit from automatically generated include statements, C++ developers enjoy no such benefits, and seem more likely to have been reporting questions such as "What includes are unnecessary?" For developers lacking modern tools, our results suggest specific productivity benefits they might potentially receive by adopting better tools that address these problems. We found that 34% of developers' reported questions were addressed by at least one existing commercial tool.

For other questions, commercial tools do not yet exist. But many tools have been proposed by software engineering researchers. We found that such tools collectively address an additional 25% of developers' reported questions. For example, developers trying to determine how to implement something might benefit from the support Blueprint [BDW10] provides. Or developers trying to determine the owner of code might benefit from expertise recommenders (e.g., [MH02]) designed to answer this question. Our results suggest that these tools address important challenges developers face and, if successful, could make developers more productive.

Of course, tools that have been designed to try to support a question developers ask still might not help in practice. Results from previous studies on this are mixed. A number of recent studies have demonstrated that some developer tools *can* successfully help developers answer questions more quickly and successfully. An evaluation of Blueprint found that it helped developers write significantly better code and find example code significantly faster [BDW10]. An evaluation of REACHER found that it was able to help developers understand what code did five times more successfully in less time (see Section 10.3). But an evaluation of three state of the art code exploration tools found no measurable effects of any of the tools [DMR07], and an evaluation of automating debugging techniques found that, even when their performance was artificially boosted, they were only sometimes helpful [PO11]. It is unclear how many research tools would successfully help developers answer questions in the field. One important use for the results reported here is to provide benchmarks for such tools: do they help developers answer the questions that are reported to be hardest to answer? For example, do contracts provide the right information for helping developers determine the intent of code? Such benchmarks are an important part of designing studies for evaluating the effectiveness of research tools and evolving their design to more effectively solve developers' real problems.

Even tools that successfully help developers answer their hard-to-answer questions still may not solve the problems developers reported. In many cases, the questions developers reported reflected situations in which a tool that was generally applicable did not help in the particular situation they experienced. Refactoring tools have long helped support renaming methods, encapsulating fields, and moving methods. But developers reported other refactorings – changing a method's scope, changing the semantics of config values, and making operations more data driven. For these operations, our results identify future directions for research to increase the scope and applicability of existing tools. In many cases, these future directions deal with understanding the semantics and meaning of code. For example, our results suggest it would be useful if future clone detectors could help to detect redundant data rather than simply syntactically similar code. And developers reported problems implementing changes to exception policies that are unaddressed by existing tools.

Many questions were highly focused around specific hypotheses, situations, or proposals relevant to the developer's current task and mental model of the code. For example, rationale questions asked why a specific decision was made or even why a specific alternative was not chosen. Questions about what code does were often scoped to a situation – e.g., what happens when an exception occurs or an operation times out. Thus, our results suggest that a key design goal for tools or languages is to better use the situations described in developers' questions to focus and filter the information that is provided by the tools.

Finally, some of developers' reported questions reflect wide-open, unexplored areas of software engineering research. 41% of developers' reported questions were questions unaddressed by either commercial or research tools. For example, developers asked "Why was it done this way?", "Why wasn't it done this other way?", "What's the policy for doing this?", and "Is this functionality already implemented?" These questions point to important opportunities for future tools to solve problems developers perceive to be hard. Addressing such questions is likely to have significant productivity benefits.

6.4. Limitations

The results from this study were gathered from a single organization. Some of the problems developers experienced may have been influenced by Microsoft's processes, practices, and conventions. While there is a huge variability among the teams and products found in this organization, other organizations might use radically different process or tools that lead to different questions being perceived to be hard-to-answer. More work is necessary to replicate this study in other organizations and contexts.

By using a survey, rather than direct observations, the results rely on developers' own reporting of questions, which is biased by perception and memory. Developers are likely to perceive the hardest, longest, most frustrating problems, not necessarily the problems that often cause a small inconvenience or for which developers never even realize a problem exists. And the prompt itself introduced bias. Developers were first primed with rating the difficulty and frequency of reachability questions, perhaps leading them to be more likely to recall similar questions. However, developers also did not report the reachability questions they rated in their free responses, thereby reducing the frequency of reachability questions in their reported hard-to-answer questions. Thus, the frequency of reachability questions in this study is likely to be biased. More generally, asking developers to report *questions* they ask might have limited the scope of the challenging information needs which the developers considered.

6.5. Conclusions

This study revealed the great breadth of challenges developers face in coding activities. Developers reported 92 hard-to-answer questions, spanning many areas of software engineering research. Some of these questions are addressed by existing commercial or research tools; more work is required to determine if these tools help to answer these questions. Other questions point to new opportunities for tools. Several of the questions developers asked were related to reachability questions, suggesting that a tool that helps developer answer these questions would address some of developers' perceived problems. In the next chapter, reachability questions are investigated in detail.

7. DEVELOPERS ASK REACHABILITY QUESTIONS[7]

In the previous chapters, studies of code exploration focused on its context, relationship to design knowledge, and connection to hard-to-answer questions. As developers build their mental model of the code, developers ask questions and read the code, talk to teammates, and use design knowledge to answer these questions. But how can a tool best support code exploration?

Many of developers' questions involve control flow. In the Exploration Lab Study (see Chapter 5), developers spent much of their time traversing call sites to callees or looking at callers of methods. Beginning at a method they thought was most relevant, they read its code. To understand what exactly it was doing, they read code in methods it called. To understand when it was called and for what purpose it was used, they read its callers. But, rather than simply trying to read all the code, developers seemed to often have a specific question in mind.

While this suggests the potential applicability of a tool, many questions remain to understand how such a tool might support developers' work. What exactly are developers asking when dealing with control flow? Are these questions really an important part of coding tasks? How long do developers spend answering them? What challenges do developers face? And what problems do these challenges cause for software development work?

To answer these questions, I reanalyzed data from the Exploration Lab Study (Section 7.1.2) and conducted two additional studies: a Reachability Survey (Section 7.2) and Reachability Observations (Sections 7.3 and 7.4). These studies are reported in this chapter. These studies discovered that reachability questions are an important part of exploring code. Consider an example from the Exploration Lab Study: after proposing a change, a developer sought to determine if it would work before implementing it. To do so, he wanted to determine "all of the events that cause this guy to get updated". While he was aware that a call graph exploration tool could traverse chains of method calls, this did not directly help. Upstream from the update method was a bus onto which dozens of methods posted events, but only a few of these events triggered the update. Existing call graph tools are unable to identify only those upstream methods sending the events that would trigger the update of interest. Unable to answer the question in any practical way, he instead optimistically hoped his guess would work, spent time determining how to reuse functionality to implement the change, edited the code, and tested his changes before learning the change would never work and all his

[7] This chapter based on work previously published in [LM10-1].

effort had been wasted.

This chapter presents data about reachability questions gathered from over 470 developers and over 70 hours of direct observations of coding tasks. In the Exploration Lab Study, developers often inserted defects because they either could not successfully answer reachability questions or made false assumptions about reachability relationships. The Reachability Survey found that, on average, 4.1 of these were thought to be at least somewhat hard to answer. And these questions were not limited to inexperienced developers or those new to a codebase: neither professional development experience nor experience with their codebase made these questions less frequent or easier to answer. Reachability questions can be time consuming to answer. And, in the Reachability Observations, developers often spent tens of minutes answering a single reachability question.

7.1. Reanalysis of exploration lab study data

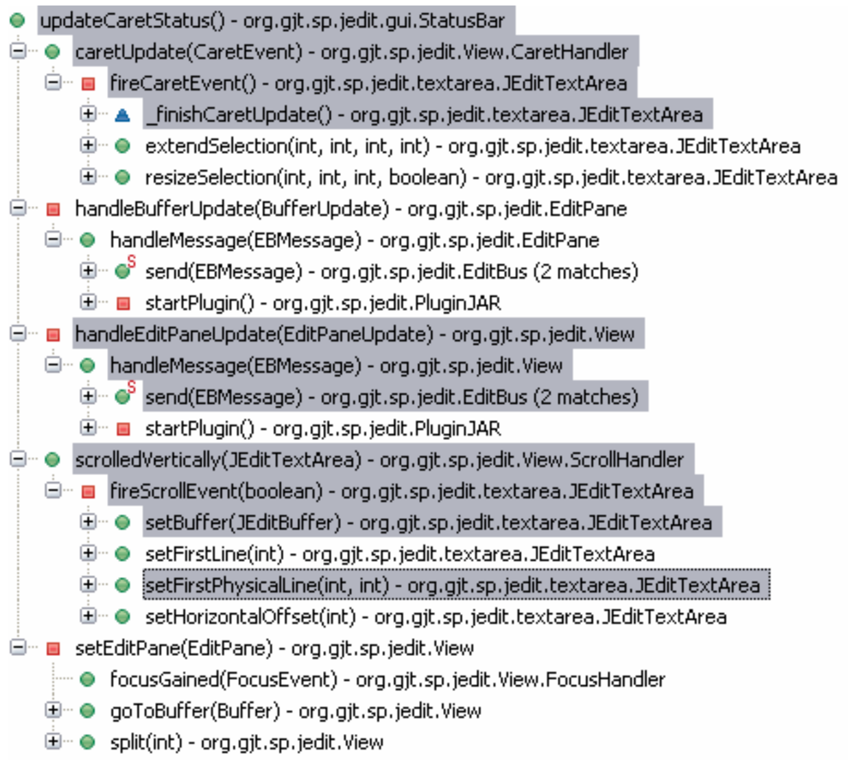
In the Exploration Lab Study (Chapter 5), we observed 13 developers at work on two 1.5 hour long changes to an unfamiliar codebase. It was found that experienced developers used their more extensive knowledge to diagnose the problem and formulate a fix addressing the underlying cause of the design problem rather than simply its symptoms. When we designed, ran, and first analyzed this study, we did *not* have the concept of reachability questions in mind. But, as we conducted the study, they emerged as an observed behavior. Hence, we decided to analyze more carefully how often and in what forms reachability questions occurred.

To do this, we first looked at the bugs developers inserted and their incorrect understandings of code. Despite spending almost the entire task asking questions and investigating code, developers frequently incorrectly understood facts about the code. Acting on these false facts, developers implemented buggy changes. In some cases, developers realized these changes were mistaken and abandoned them, reverting the code. When developers inserted defects (whether or not they were still present in the final code produced), we analyzed questions developers asked and actions they took to look for specific information they incorrectly understood. We then examined which of these defects were related to reachability questions developers asked or might have asked.

When developers did ask reachability questions, they used a number of strategies to answer these questions. We also examined developers' strategies to identify those that were particularly time consuming and error prone, and to understand the challenges developers faced that caused these difficulties.

7.1.1. Method

The method used to conduct this study is described in Chapter 5. We conducted two additional analyses of the data. First, we identified edits to the code and clustered these into changes. We labeled each change as to whether it was later abandoned and if it contained a bug. For changes containing a bug, we then looked to see if the developer had either asked a question or had otherwise made an assumption. We then attempted to determine if the question or assumption could be addressed by a reachability question. In a second analysis, we looked for examples of time-consuming questions that developers spent ten or more minutes answering.



```
updateCaretStatus() - org.gjt.sp.jedit.gui.StatusBar
  caretUpdate(CaretEvent) - org.gjt.sp.jedit.View.CaretHandler
    fireCaretEvent() - org.gjt.sp.jedit.textarea.JEditTextArea
      _finishCaretUpdate() - org.gjt.sp.jedit.textarea.JEditTextArea
      extendSelection(int, int, int, int) - org.gjt.sp.jedit.textarea.JEditTextArea
      resizeSelection(int, int, int, boolean) - org.gjt.sp.jedit.textarea.JEditTextArea
  handleBufferUpdate(BufferUpdate) - org.gjt.sp.jedit.EditPane
    handleMessage(EBMessage) - org.gjt.sp.jedit.EditPane
      send(EBMessage) - org.gjt.sp.jedit.EditBus (2 matches)
      startPlugin() - org.gjt.sp.jedit.PluginJAR
  handleEditPaneUpdate(EditPaneUpdate) - org.gjt.sp.jedit.View
    handleMessage(EBMessage) - org.gjt.sp.jedit.View
      send(EBMessage) - org.gjt.sp.jedit.EditBus (2 matches)
      startPlugin() - org.gjt.sp.jedit.PluginJAR
  scrolledVertically(JEditTextArea) - org.gjt.sp.jedit.View.ScrollHandler
    fireScrollEvent(boolean) - org.gjt.sp.jedit.textarea.JEditTextArea
      setBuffer(JEditBuffer) - org.gjt.sp.jedit.textarea.JEditTextArea
      setFirstLine(int) - org.gjt.sp.jedit.textarea.JEditTextArea
      setFirstPhysicalLine(int, int) - org.gjt.sp.jedit.textarea.JEditTextArea
      setHorizontalOffset(int) - org.gjt.sp.jedit.textarea.JEditTextArea
  setEditPane(EditPane) - org.gjt.sp.jedit.View
    focusGained(FocusEvent) - org.gjt.sp.jedit.View.FocusHandler
    goToBuffer(Buffer) - org.gjt.sp.jedit.View
    split(int) - org.gjt.sp.jedit.View
```

Figure 7.1. Developers using Eclipse's call graph exploration tool to traverse callers found it difficult to identify both feasible paths and those paths that lead to their target. In the view above, the methods on paths to the target have been manually highlighted – Eclipse cannot do this automatically. The target is several methods further away, through paths with additional methods with high branching factors.

7.1.2. Results

Developers implemented an average of 1.2 changes per task. Developers abandoned changes when they learned their changes could never work, found a bug they could not fix, or decided they did not have sufficient time to finish the change. Developers abandoned an average of 0.3 changes per task, two thirds of which contained bugs. Developers abandoned changes that did not contain a bug either because they no longer thought the change was a good design or did not think they had time to finish it. Overall, developers spent over two-thirds of their time (68%) investigating code – either testing or doing dynamic investigation using the debugger (22%) or reading, statically following call relationships, or using other source browsing tools (46%). They spent the remainder of their time editing (14%), consulting or creating other artifacts (task description, notes in Notepad, diagrams)(6%), or reasoning without interacting with any artifacts (11%)

False assumption or question related to a bug	**Correct answer**	**Related reachability question**	**Dist**	**Notes**
Method *m* is fast enough that it does not matter that it is called more frequently.	This method sends an event which triggers a hidden call to an extremely expensive library function.	*find ends in traces(jEdit, m_{start}, m_{end} , ?)*	4	Finds calls to downstream library functions in *m*
Why is calling *m* necessary?	*m* determines if the screen needs to be repainted and triggers it if necessary.	*find ends in traces(jEdit, m, m_{end} , ?)*	5	Finds calls to library functions, including one that triggers screen repainting
From what callers can the guards protecting statement *d* in method *m* be true?	More than one caller can reach *d.*	*find callers(m) in traces(jEdit, ?, d, ?)*	1	Finds callers reaching *d*
Method *m* need not invoke method *n* as it is only called in a situation in which *n* is already called. (2 bugs)	Method *m* is called in several additional situations.	*find n in traces(jEdit, ?, m, ?)*	1, 2	Finds callers reaching *m*
The scroll handler *a* does not need to notify *b*, because *b* is unrelated to scrolling.	Method *b* updates the screen to reflect updated scroll data signaled by *a.*	*find grep("scroll") in traces(jEdit, b_{start}, b_{end}, ?)*	1	Finds functionality invoked by *b* that is relevant to scrolling
Removing this call in *m* does not influence behavior downstream.	*m* no longer clears a flag, disabling functionality downstream	*compare(traces($jEdit_{old}$, m_{start} , ?, ?), traces($jEdit_{new}$, m_{start} , ?, ?)*	4	Finds differences in behavior resulting from the change, including downstream functionality that is no longer invoked.
What situations currently trigger this screen update in *m?*	A variety of user input events eventually cause *m* to be invoked	*find ends in traces(jEdit, ?, m, ?)*	3	Finds upstream methods with no callers, including user input event handlers called only by the framework.

Table 7.1. Questions developers failed to answer or false assumptions developers made in the Exploration Lab Study that are (1) associated with an implemented change containing a defect and are (2) associated with a reachability question (see Section 1.2 and Table 1.1 for reachability question definitions). For each reachability question, dist is the shortest call graph distance between the origin statement developers investigated and any statement found by the reachability question.

7.1.2.1. Causes of defective changes

Half of all changes developers implemented contained a bug. In half of these defective changes (8 changes), we were able to relate the bug to a reachability question either in a false assumption that developers made (75%) or a question they explicitly asked (25%).

Table 7.1 lists the false assumptions or questions that were related to reachability questions and the corresponding reachability question. Developers often made incorrect assumptions about upstream or downstream behaviors as they reasoned about the implications of removing calls currently present in the code. These assumptions took different forms depending on the change they considered. Upstream reachability assumptions often occurred when developers asked or assumed that behavior was redundant and unnecessary because it would always be called somewhere else. In these cases, the call graph distance from the origin statement they were investigating to target behavior was often small (mean = 1.75). These questions were challenging to reason about because it was difficult to determine which calls were feasible. Downstream reachability assumptions often occurred when developers made false assumptions about how a method mutated data or invoked library calls. Here, the relevant effect was further away (mean = 3.5 calls), and developers had no reason to believe that traversing the path to the target would challenge their assumption.

7.1.2.2. Tedious and time consuming strategies

In addition to the bugs that arose from assumptions developers made when they *should have* asked reachability questions, there were many cases where the developers *did* ask reachability questions and formulated a strategy to answer them. Developers spent much of the task investigating code by traversing calls in an attempt to understand what methods did and the situations in which they were invoked. Most participants rapidly switched between a call graph view (static) and the debugger call stack (dynamic). Static investigation allowed developers to navigate to any caller or callee at will. But as developers traversed longer paths of calls, developers were likely to hit infeasible paths. Several guessed incorrectly about which paths were feasible. Dynamic investigation was more time-consuming to begin – developers set breakpoints, invoked application behavior, and stepped through breakpoint hits until the correct one was reached. At task start, most investigation was relatively unfocused – developers attempted to make sense of what the methods did and the situations in which they were called. As the tasks progressed and developers began to propose changes, the questions grew increasingly focused and developers sought to navigate to specific points in code.

Developers differed greatly in the effectiveness and sophistication of the strategies they employed. Particularly challenging for many participants was upstream navigation. Two participants did not realize they could visually scan the call stack to find an upstream method and instead spent much time (16 mins, 10 mins) locating the method by using string searches and browsing files. Three participants spent ten or more minutes (17, 13, and 10 mins) using a particularly tedious strategy to navigate upstream from a method *m* across only feasible paths: adding a breakpoint to each of m's callers, running the application, executing functionality, noting which callers executed, and recursing on these callers. Many participants used Eclipse's call graph exploration tool to traverse calls, but then they traversed infeasible paths and they experienced problems determining which calls led to their search targets (Figure 7.1). The three most experienced participants instead invoked the functionality and copied the entire call stack into a text editor. But even these experienced participants experienced problems reasoning about reachability relationships. These participants created three of the reachability question related defects.

7.1.3. Discussion

Despite spending much of the task investigating code, developers were often unsuccessful in correctly understanding what it did. Developers made many false assumptions about relationships between behaviors that in some cases led to defects. Developers' tools were ill-suited for answering reachability questions, often forcing them to use tedious and time-consuming strategies to answer specific well-defined questions. And had developers been able to more easily check their erroneous assumptions that led to defects, their changes might have been more accurate.

While these results suggest that reasoning about reachability relationships is important for developers who are trying to understand unfamiliar, poorly designed code, these results might not be generalizable. While we expect developers do work with such code in the field, it is unclear how typical such a task is. While the carefully controlled setting of a lab study allowed us to evaluate the success and accuracy to a degree impossible in the field, lab studies are never able to perfectly replicate conditions in the field. Understanding real code in more typical tasks might involve fewer and less challenging reachability questions. Developers working in the same codebase over a period of time might be able to use their knowledge to directly answer reachability questions as studies suggest developers learn facts including callers and callees of methods with increasing experience [FMH07]. Developers had limited time in which to work, which likely led them to rush changes with less investigation than they might otherwise have done. And several developers did not seem to have had much experience understanding large, complex codebases. Are reachability questions frequent and challenging for developers at work in the field?

7.2. Reachability survey

In order to understand the frequency and difficulty of reachability questions in the field, we conducted a survey of developers in which they rated 12 questions for difficulty and frequency.

7.2.1. Method

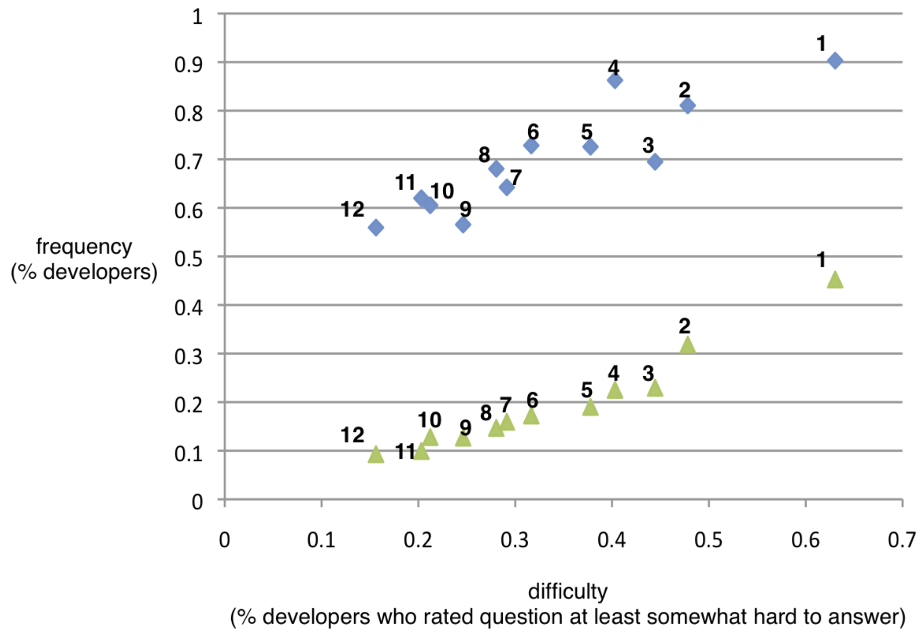
We randomly sampled 2000 participants from among all employees at Microsoft's Redmond campus listed as a developer in the address book. Each was sent an email inviting them to participate in our survey. We received 460 responses from developers and excluded 8 additional responses from non-developer positions (we consider the 460 respondents here, in contrast to the 179 respondents who completed the free response item and are described in Section 6.1). Respondents included 14 architects, 43 lead developers, and 403 developers. Most worked in a single or shared office while a small number (33) worked in an open, shared space. Respondents ranged in professional software development experience from the very inexperienced (0 years) to the very experienced (39 years), with a median of 9 years experience. Respondents frequently changed codebases, ranging in time spent in their current codebase from 0 to 8.33 years, but with a median of only 1 year. Nevertheless, 69% agreed that they were "very familiar" with their current codebase. Developers' teams were involved in a wide range of activities – 43% bug fixing, 34% implementation, 16% planning, and 7% other. Developers reported that they typically spent 50% of their work time editing, understanding, or debugging code, with a range from 0 to 100%.

In the main portion of the survey, developers were asked to rate the frequency and difficulty of 12 questions. These questions were selected from the questions reported in a previous study of developers' questions about code [SMV08] and questions identified in the Exploratory Lab Study. Some of these were closely related to reachability questions ("In what situations is this method called?") while others were more indirectly related ("What are the implications of this change?"). However, we observed many of the indirectly related questions being refined into reachability question in the Exploratory Lab Study. So, we hypothesized that developers often answer these questions by asking reachability questions.

We piloted the survey with 4 graduate students and 1 developer to ensure that the meaning of the questions was clear, and we used their feedback to rephrase questions that were unclear. For each question, respondents were asked to rate how often in the past 3 days of programming they had asked the question and to rate its difficulty on a 7 point scale from very hard to very easy. 56 participants did not answer all questions. When a participant did not answer the questions necessary for a particular comparison, that participant was dropped from that comparison. To analyze the data, we looked both at simple descriptive statistics and correlations between ratings and demographic variables. We report these results using r (the Pearson product-moment correlation coefficient) and p (a statistical significance measure – smaller is more significant).

7.2.2. Results

On average, developers reported asking more than 9 of these questions every day. These questions were often hard to answer. Of the 12 questions that the developers rated, developers rated an average of 4.1 questions at least somewhat hard to answer and 1.9 as hard or very hard to answer. Few developers thought all these questions were easy to answer: 82% of respondents rated at least 1 question at least somewhat hard to answer, and 29% rated at least 1 question as very hard to answer. Surprisingly, developers do not ask these questions significantly less frequently and they are not significantly easier to answer as they become more experienced ($r = -.07$, $p = .14$; $r = -.01$, $p = .81$) or after spending more time in a codebase ($r = -.04$, $p = .41$; $r = -.07$, $p = .15$). Nor does the quality of the codebase significantly affect the frequency of these questions ($r = -.08$, $p = .10$). While it is harder to answer these questions on lower quality code (r = .36, p < .0001), it is not possible to say if this is unique to these questions or simply that all questions become harder to answer in poorly maintained code.



1. What are the implications of this change?
(e.g., what might break)
2. How does application behavior vary in these different situations that might occur?
3. Could this method call potentially be slow in some situation I need to consider?
4. To move this functionality (e.g., lines of code, methods, files) to here, what else needs to be moved?
5. Is this method call now redundant or unnecessary in this situation?
6. Across this path of calls or set of classes, where should functionality for this case be inserted?
7. When investigating some application feature or functionality, how is it implemented?
8. In what situations is this method called?
9. What is the correct way to use or access this data structure?
10. How is control getting (from that method) to this method?
11. What parts of this data structure are accessed in this code?
12. How are instances of these classes or data structures created and assembled?

◆ at least once per 3 days ▲ at least twice a day

Figure 7.2. Frequency vs. difficulty for 12 reachability-related questions sorted by decreasing difficulty.

Figure 7.2 plots the questions' frequency against difficulty. Interestingly, difficulty was positively related to frequency (r = .35, p < .0001). Both the most frequent and hardest to answer question was: “What are the implications of this change?” Generally, the most frequent and difficult questions were the most high level. For example, half of respondents reported asking “What are the implications of this change?” at least twice a day, and 63% of respondents rated it at least somewhat difficult to answer. Of course, some questions are much more frequent and difficult than others. Over 60% of developers thought answering “What are the implications of this change?” was usually at least somewhat hard to answer, while this was true of only 16% of respondents for “How are instances of these classes or data structures created and assembled?”

7.2.3. Discussion

Our results revealed that developers frequently ask questions that they might refine into reachability questions, that these questions are often difficult to answer, and that experience does not remove the need to ask these questions. These results suggest that answering these questions is an important part of how all developers understand code, whether they are new to a codebase or know it well and whether the codebase is poorly designed or well designed. These findings are still limited in that all survey respondents were taken from a single company. But respondents differed greatly by the products on which they worked, by experience with the codebase, by overall professional experience, and by software project phase. These results demonstrate that techniques that help developers more effectively answer these questions are important. However, the results do not establish that developers answer these questions by asking reachability questions. Do developers frequently ask reachability questions, and are they time consuming to answer? What do examples of reachability questions in the field look like?

7.3. Reachability observations

In order to better understand the situations in which developers ask reachability questions and the strategies they use to answer them, we observed 17 developers at work on their everyday coding tasks.

7.3.1. Method

We recruited 20 developers at Microsoft from the respondents to the Reachability Survey to participate in observation sessions. I conducted the observations; in each, I visited a single developer in their office. Developers used a variety of programming languages (C++, C#, JavaScript), editors, and debuggers. After briefly introducing myself and reviewing the purpose of my study, participants were asked to work on a coding task in their codebase for the remainder of the approximately 90 minute sessions. Three participants finished their first task and chose a second task. When selecting tasks, participants were encouraged to choose a task involving unfamiliar code, minimally defined as code they had not written themselves. While only 35% of the tasks that developers chose were tasks they planned to do at the time of our session, 95% (all but one) of the tasks they chose were on their lists of tasks to do. The remaining task was a bug previously assigned to another team-member. The work we observed was not biased towards the beginning or the end of tasks: 45% of the tasks were tasks the developer had previously begun, and developers completed 45% of their tasks. All but one developer stopped working after they had completed testing their fix and before having their teammates code-review the change.

I asked participants to think aloud as they worked. When deeply engrossed in the tasks, participants occasionally forgot to talk, and I prompted them to resume by asking what they were trying to do or having them confirm or reject a statement about what they appeared to be doing. To record the sessions, I recorded audio and took notes. Two of the recordings were lost due to equipment failure, leaving 18 participants. From the recordings and observer notes, I produced time stamped, annotated transcripts of the sessions spanning 386 pages.

To analyze the data, I first reviewed the transcripts and qualitatively summarized what developers were doing. Next, I iteratively designed a coding scheme for describing developers' activities. I coded 17 of the 18 sessions – one session did not include any implementation task. Each session was coded for activity at one-minute time granularity. Participants occasionally retrospectively described particularly memorable past tasks or talked about how they approached tasks in general which we did not include in the activity list, but mention in the discussion section. Developers were interrupted by replying to task-unrelated emails, by teammates dropping by, or discussions with the interviewer. All task-irrelevant activity was coded as an interruption and excluded from the analysis of time use. Due to equipment failure, 15 minutes of the recordings were lost out of a total of 962 minutes of task-related activity. In most cases, developers stopped working on their tasks once they had completed its implementation. But one developer reached the end of his task and conducted a code review. To be consistent, code reviews are not included in activity times.

7.3.2. Results

Developers spent a majority of their time understanding code by debugging (33%) or proposing changes and investigating the implications of the changes (28%). 9 of the 10 longest debugging and implication investigations were associated with a reachability question.

7.3.2.1. Activities

Figure 7.3 depicts the sequence of activities we observed and the time developers spent on each. When working on a bug they did not already understand, developers first sought to *reproduce* the problem by following steps in the bug report to confirm that the bug had not already been fixed, ensure that a fix could be tested, and provide a way to begin using the debugger. Developers faced with incorrect application behavior, either from the original bug or introduced by their fix, *debugged* to assign blame to specific program points exhibiting incorrect behavior. After determining the cause of a bug or when beginning a feature implementation task, developers began to propose fixes to solve the problem and *investigated* the implications of the proposals on program behavior. Developers then *edited* the code to implement the change. When editing, developers sometimes *reused* existing functionality and sought to learn its name and how to correctly reuse it. Developers *compiled* and built the application, sometimes producing compile errors they debugged. Finally, developers *tested* their changes, often revealing defects they then debugged.

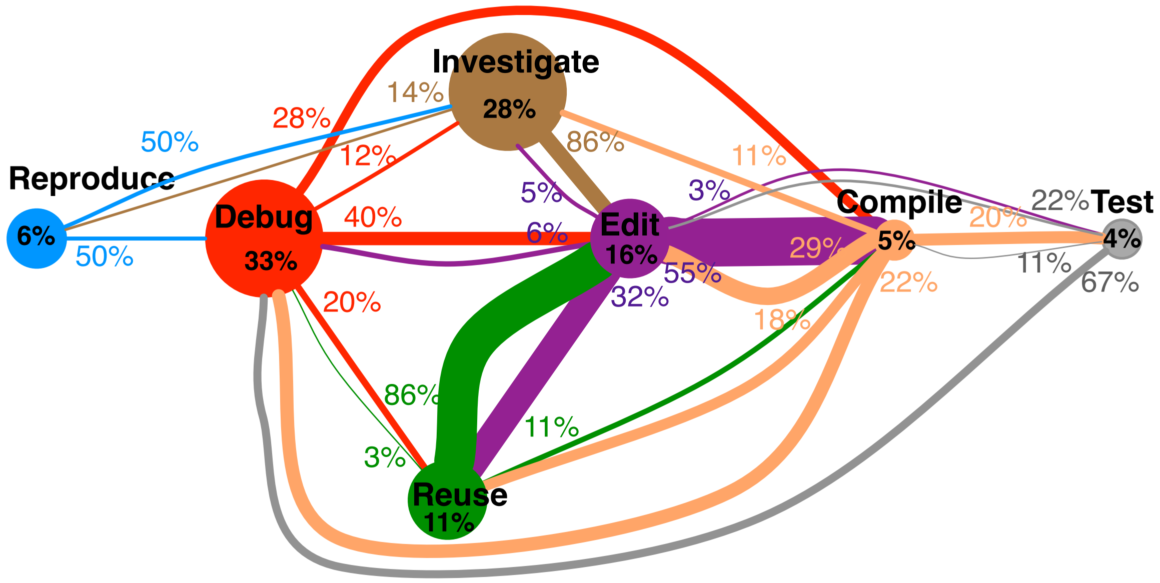


Figure 7.3. Developers' activities (circles with % of activity time) and transitions between activities (lines with % of transitions from activity). Transitions from an activity are in the activity's color, and left to right transitions are above right to left.

7.3.2.2. Time-consuming activities

While debugging and investigating code, we again saw that developers frequently asked reachability questions. In order to examine the relationship of these activities to reachability questions, we looked for reachability questions in the 5 longest debugging and 5 longest investigation activities. Each of these activities had a central, *primary question* developers tried to answer throughout the activity. Surprisingly, the primary question in 9 out of 10 of these activities was a reachability question (see Table 7.2). At the beginning of these activities, developers rapidly formulated a specific question expressing search criteria describing statements they wished to locate. For example, to debug a deadlock, a developer began at a statement and began traversing callees to try to find statements acquiring resources. 51 minutes later, this finally revealed the sequence of behaviors causing the deadlock.

Developer's primary question (Debugging activities)	Time (min)	Reachability question	Notes
Where is method *m* generating an error?	66	*find grep(errorText) in traces (p, m_{start}, m_{end}, ?)*	Finds the statement downstream from *m* outputting error text
What resources are being acquired to cause this deadlock?	51	*find ACQUIRE_METHODS in traces (p, o, d, ?)*	Finds calls to methods acquiring resources, including those leading to the deadlock.
"When they have this attribute, they must use it somewhere to generate the content, so where is it?"	35	*find reads(attribute) in traces(p, o, d, ?)*	Finds downstream uses of *attribute*, including those generating the content.
"What [is] the test doing which is different from what my app is doing?"	30	*compare(traces(p_{test}, o, d, ?), traces(p_{app}, o, d, ?))*	Finds differences in behavior between the test program and app program
How are these thread pools interacting?	19	*find methods(T) in traces(p, o, d, ?)*	Finds any calls into methods in thread pool types *T*.

Developer's primary question (Investigation activities)	Time (min)	Reachability question	Notes
How is data structure *struct* being mutated in this code (between *o* and *d*)?	83	*find writes(struct) in traces(p, o, d, ?)*	Finds all downstream statements mutating *struct*
"Where [is] the code assuming that the tables are already there?"	53	*compare((traces(p, o, d, tablesLoaded), (traces(p, o, d, tablesNot-Loaded))*	Finds different behaviors the code exhibits when tables are not loaded
"How [does] application state change when *m* is called denoting startup completion?"	50	*find writes(FIELDS) in traces(p, m_{start}, m_{end}, ?)*	Finds state changes caused by *m*
"Is [there] another reason why status could be non-zero?"	11	*find reachingDefs(status) in traces(p, ?, d, ?)*	Finds upstream statements through which values flow into *status*, including those creating its values

Table 7.2a (top) and 7.2b (bottom). The 5 of the 5 longest debugging activities and the 4 of the 5 longest investigation activities associated with a reachability question. For each activity, the developer's primary question during the activity, the length of the activity, and the related reachability question.

The one investigation activity that was not related to reachability questions dealt with understanding a section of code. The developer was reusing the code and exhaustively read it in order to identify design decisions that might not be compatible with the way in which he intended to reuse it.

When answering reachability questions, developers explored the code either dynamically using the debugger and logging tools or statically using source browsing tools. Interestingly, developers did not primarily use the debugger to debug and code browsing tools to investigate implications. Instead, like the lab study participants, developers often made use of both tools as they sought to answer multiple lower-level questions or tried alternative strategies for answering their primary question. Developers constantly dealt with uncertainty during their tasks both from generating and testing hypotheses and wondering about the correctness of results produced by their tools.

An example from the longest debugging activity helps illustrate several of these points. Observing an error message in a running application, one developer spent 66 minutes locating the cause of the error message in the code. Using knowledge of the codebase, he rapidly located the code implementing the command he had invoked in the application. But it was not obvious where it triggered the error. Hoping to "get lucky", he did a string search for the error message but found no matches. Unsure why he did not find any matches, he next began statically traversing calls from the command method in search of the error. But he rapidly determined he was unsure which path would be followed when the command was invoked. Switching to the debugger, he stepped through the code until learning his project was misconfigured and creating spurious results both in his debugger and code searches. After resetting his project configuration, he again did a string search for the error string and found a match. However, many callers called the method, any one of which might be causing his error. So he returned to stepping in the debugger. Finally locating code that seemed relevant, he quickly browsed through the code statically. Finally, he returned to the debugger to inspect the values of some variables.

7.3.3. Discussion

Participants worked on their everyday tasks that dealt with unfamiliar code. In these tasks, developers spent over half of their time debugging or reasoning about the implications of their changes. In 9 of the 10 most time-consuming activities, the developer's primary question was a reachability question. Developers were at a point in code and had specific search criteria describing the statements they wished to find. But finding these statements was hard and time-consuming as developers searched through large amounts of task-irrelevant code. In contrast to results from the Exploration Lab Study, the questions in the Reachability Observations were all questions developers explicitly asked.

Like all studies, these findings may have been influenced by the practices and tools that developers used that might differ in other organizations. In organizations with more extensive documentation or commenting processes, developers might rely on these more than the code itself. Developers did not have access to sophisticated UML reverse-engineering tools. None of our developers had unit tests extensive enough to rely on to test the correctness of their changes. Extensive unit tests might lead to more implementation of speculative changes, followed by testing, rather than extensive investigation prior to changes.

7.4. Additional examples

To gather additional examples of challenging reachability questions, several graduate students at Carnegie Mellon University were asked to report examples of reachability questions they had asked. These reports were not typical or easy to answer reachability questions, but some of the hardest, most tedious questions students experienced.

7.4.1. Debugging a null pointer exception

A PhD student at Carnegie Mellon reported a challenging debugging situation she was currently experiencing, and I observed some of her work. She was working in a codebase she had written herself that implemented a static analysis system called Fusion and had just raised a `NullPointerException`. Her code has a simple protocol: an `XMLRetriever` object was created, initialized, and later used. But, at the use site, there was a null pointer exception on one of its fields, indicating it had not been initialized. She wondered, how could an `XMLRetriever` object ever be created without being initialized?

To answer this question, she spent 40 minutes debugging, using the debugger to inspect values and inspecting code with code browsing tools. Her knowledge of the control flow structure of the application helped her to generate and test hypotheses about why the initialization method might not have been called. As she worked, she tried to understand how each of the places where the `XMLRetriever` is used were connected to the initialization call. Might there be some path to one of the places where it is used along which it is not first initialized? But these paths were numerous and long; there were 96 paths, some as long as 13 calls. Traversing these paths was hard.

She eventually determined the answer: there was a conditional along the path to the initialization call. The conditional was designed to select the cases in which another call should be made, which normally corresponded to when the initialization should be called. But there was a situation when these cases did not correctly correspond, and this caused the bug. To fix the bug, she moved the initialization call out from inside the conditional.

7.4.2. Considering a change

A PhD student at Carnegie Mellon was considering several possible alternative changes and wondered which one was the best design. To determine if one of the designs might work, a question emerged: would an object with a null constituent part break existing code? If the existing code supported this, a simple design could be used. To answer this question, she explored the code. From experience, she knew the method usually used to create such objects, which it did by calling into a factory used to create one of many possible sub-types. Much of the work of these methods was to set up the constituent parts. She hypothesized that maybe objects could already be created with a null constituent part. But since such objects would not need the initialization provided by the factory, they might be created along a different path that did not go through the factory. To test this hypothesis, she selected subtypes and searched for references to their constructors. Unfortunately, there were many subtypes. Selecting each one, searching for constructor references, and following the path to see if it went into the factory was tedious and time consuming.

7.4.3. Debugging incorrect results

Another Carnegie Mellon PhD attempted to debug code he wrote which used a framework (for which he had source in his workspace). Using the framework, the results of his code were mysteriously incorrect. Through extensive investigation, he narrowed the problem to a Java Collection object which contained incorrect values. He next tried to understand where the collection was being accessed and how it was being used. But control flowed back and forth between his code and the framework, and finding the correct corresponding method in each case was both tedious and made keeping track of the path challenging. He wished to search downstream across this path for places where the collection was being mutated.

7.5. General discussion

We found that reachability questions are frequent, often hard to answer, associated with false assumptions that lead to bugs, and asked by developers in many of the most time consuming debugging and investigation tasks. Several developers in the lab study became so overwhelmed investigating code that they gave up. Developers at work on actual tasks in the field often spent tens of minutes answering single reachability questions when debugging or investigating the implications of their changes. In all of these cases, developers asked questions and explored the code to search for statements answering their questions. Linking these diverse problems to reachability questions helps better explain their underlying causes and suggests common solutions.

7.5.1. Strategies for answering reachability questions

Developers may choose from among several classes of strategies for answering reachability questions: reasoning using facts they already know, communicating with teammates, or dynamically or statically exploring code. For code that developers know well, developers may already know the answer [FMH07]. But this level of understanding is difficult to achieve due both to the number of reachability relationships present in a codebase and because relationships often change as developers edit the code. One field study participant spent several minutes investigating code he had written himself a little over a year earlier because he was not certain of several important details unique to his task and he was concerned others might have edited the code. Conversely, even developers new to a codebase are able to generate hypotheses about reachability relationships by interpreting identifiers and using their knowledge about how they expect an application to work. Exploration Lab Study participants assumed that an EditBus was connected to edit events. But when developers wished to test these hypotheses, they used other strategies.

Developers communicate with their teammates both directly through face-to-face communication, instant message, or email and indirectly through documentation and comments. Where they exist, documentation diagrams such as UML sequence diagrams could help answer some reachability questions provided they anticipate the correct question. But nearly all of the questions we observed were highly specific to the developers' task, making it unlikely that such a diagram would exist. Developers occasionally made use of direct communication, often instant messaging teammates they thought might know all or part of an answer. But teammates often were not available to immediately respond. Moreover, for longer face-to-face interruptions, developers are sometimes expected to have already done due diligence to get a general understanding before asking a lengthy question of a busy and more knowledgeable teammate (see Chapter 4). Of course, teammates also eventually leave the team, may be otherwise unavailable, might have forgotten the answer, or might never have known the answer at all.

Thus, developers often answered reachability questions by exploring the code. In dynamic exploration, developers run the program and observe its output either directly or through tools such as a breakpoint debugger, logging statements, or logging tools. In some cases, generating the trace to be dynamically investigated was difficult or impossible because special hardware was required, it took a long time for the application to run and generate the trace, or it was unclear what application input was necessary to generate the trace. A developer in the Reachability Observations study working with a web application added logging statements before waiting a day for it to execute a lengthy batch job. Moreover, some reachability questions forced consideration of all possible traces. Developers sometimes randomly invoked application behavior in an attempt to generate desired traces. When possible, there were several advantages of dynamic exploration. Developers could inspect state and even mutate state to select the trace being followed. Breakpoints allowed developers to search for paths to a statement. But setting breakpoints was impractical when searching for many statements (e.g., any method in a type) or when developers did not know the statements for which they were searching (e.g., all statements related to scrolling).

Some of the problems we observed in the Exploration Lab Study could be attributed to a lack of knowledge of effective dynamic investigation strategies. Developers exploring upstream by iteratively setting breakpoints could have instead much more effectively inspected call stacks. However, developers devising and choosing strategies must simultaneously hypothesize answers to their questions, keep track of the question they are answering and information they have found, and deal with frequent interruptions from teammates [KDV07]. In these situations, developers may not have time to reflect at length on their strategies. Better educating developers about the types of questions they ask and the strategies they could use to answer them might help them devise more effective code exploration strategies. Of course, developers would also benefit from a tool that more directly supports these questions, as described in Chapters 9 and 10.

7.5.2. Challenges when statically exploring code

In static exploration, developers navigate the code by using source browsing tools such as a call graph exploration tool or textual searches for names. In contrast to dynamic exploration, static exploration does not require running the program. Call graph tools, such as the Eclipse call hierarchy, allow developers to follow chains of calls through the source. However, we observed many cases where these chains contained infeasible paths that could never execute. Through our direct observations and retrospective accounts from our participants, we discovered several idioms that created correlated conditionals with widely separated producers and consumers that were particularly difficult to statically explore. In an event bus architecture, messages are created by a producer, sent over a bus, and subscribed to by consumers. In COM, a pointer is initialized to a particular implementation of an interface (producer) and passed to call sites invoking methods on the interface (consumer). In frameworks, clients often register their implementations of framework interfaces with the framework (producer) which then uses dynamic dispatch (consumer) to transfer control back. In a property system, values referring to properties are created (producer) and used to access property getters or setters which look up the property (consumer).

Several, but not all, of these idioms often produce high branching factors in the control flow graph. A common interface (e.g., `IRunnable` in Java) may have many implementations, creating a large branching factor at dynamic dispatch. In an event bus, many methods call the bus send method and many bus receive methods are called by the bus, creating two high branching factor locations. For the developer, the effect of correlated conditionals is to create many possible edges to traverse, forcing the developer to guess which are feasible or attempt to manually simulate control flow by propagating data over control flow paths. We observed that performing path simulation manually was nearly impossible for statements with high branching factors as there were simply too many paths to consider.

7.6. Conclusions

Modern development environments provide developers with a debugger and source browsing tools for exploring code. The studies reported in this chapter found that these tools only indirectly answer reachability questions, which are a central part of many challenging coding tasks. The results suggest that developers could more quickly and accurately understand and explore code with tools that more directly support answering reachability questions. But one of the significant challenges developers face when exploring code is dealing with infeasible paths. A tool for helping developers explore code should also help filter their exploration to those paths that are actually feasible and relevant to the questions developers ask.

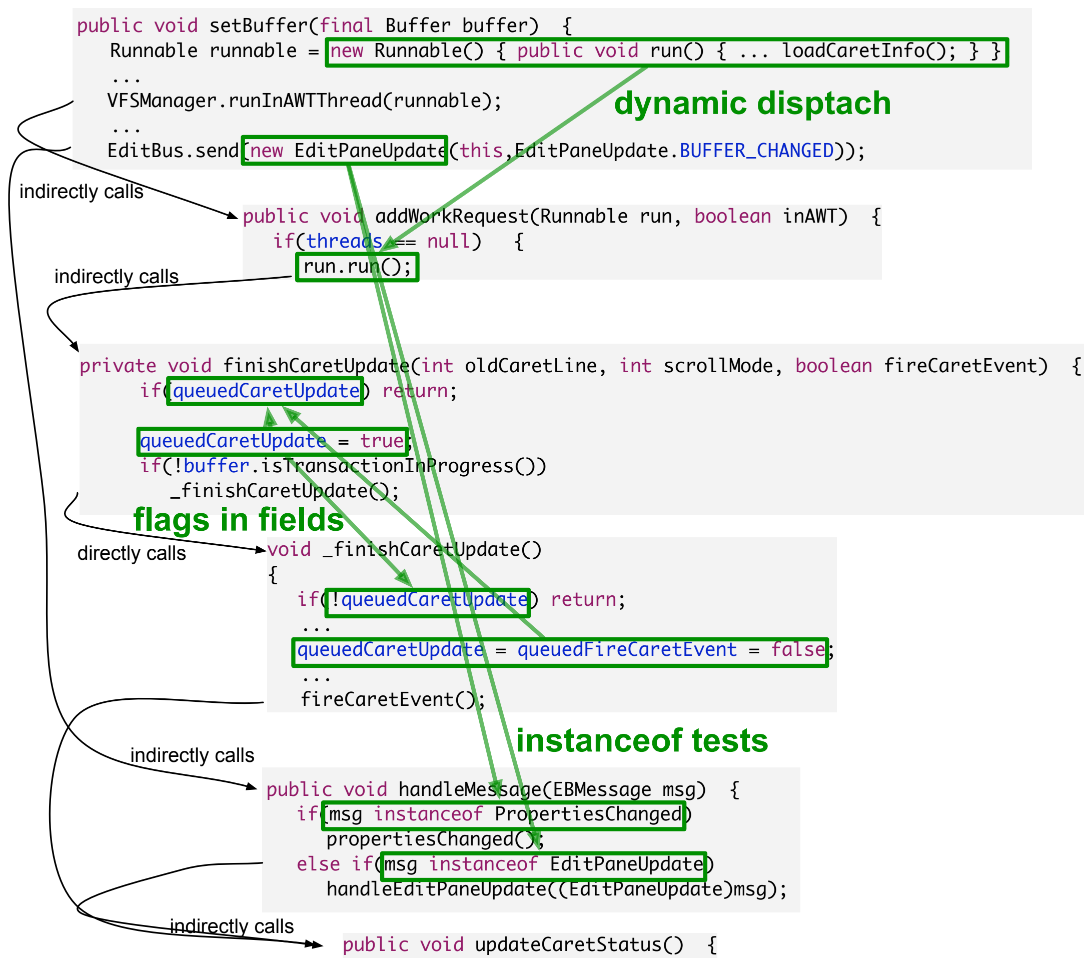
8. Fast Feasible Path Analysis

Consider the following code:

```
1       if (g)
2           h = false;
3       if (h)
4           foo();
```

Note that the path 1, 2, 3, 4 is infeasible. Recall that infeasible paths are caused by *correlations* between conditionals – the branch taken at one conditional determines the branch taken at a second conditional. For an execution path in which `g` is true and line 2 executes, `h` will be false and line 4 will not execute.

More complex infeasible paths often occur in codebases. Consider the example in Figure 8.1 taken from the Exploratory Lab Study (see Chapter 5 and Section 7.1). This example is more complex in two important respects. First, conditionals include not only Boolean flags but also dynamic dispatch and `instanceof` tests. An instance of an anonymous subclass of `Runnable` is created and `run` is called on this instance. Statically, at the call site to `run`, any method implementing `run` might execute. But, in an execution in which the object is of the type of the anonymous class in `setBuffer`, that implementation of `run` will execute. This also highlights the second important difference: parameters. The code that executes at the call to `run` in `addWorkRequest` depends on the value of the parameter `Runnable run`. When reached along a path from `setBuffer`, the value of `run` will be an instance of the anonymous class created in `setBuffer`. But, for other paths on which it is called, it could have other values. Which paths are feasible depends on the *context* in which it is called – the values that have been assigned to parameters and fields.



```
public void setBuffer(final Buffer buffer)  {
   Runnable runnable = new Runnable() { public void run() { ... loadCaretInfo(); } }
   ...
   VFSManager.runInAWTThread(runnable);
   ...
   EditBus.send(new EditPaneUpdate(this,EditPaneUpdate.BUFFER_CHANGED));

public void addWorkRequest(Runnable run, boolean inAWT)  {
   if(threads == null)   {
      run.run();

private void finishCaretUpdate(int oldCaretLine, int scrollMode, boolean fireCaretEvent)  {
     if(queuedCaretUpdate) return;

     queuedCaretUpdate = true;
     if(!buffer.isTransactionInProgress())
        _finishCaretUpdate();

void _finishCaretUpdate()
{
   if(!queuedCaretUpdate) return;
   ...
   queuedCaretUpdate = queuedFireCaretEvent = false;
   ...
   fireCaretEvent();

public void handleMessage(EBMessage msg)  {
   if(msg instanceof PropertiesChanged)
      propertiesChanged();
   else if(msg instanceof EditPaneUpdate)
      handleEditPaneUpdate((EditPaneUpdate)msg);

public void updateCaretStatus()  {
```

Figure 8.1 Examples of infeasible paths created by constant-controlled conditionals.

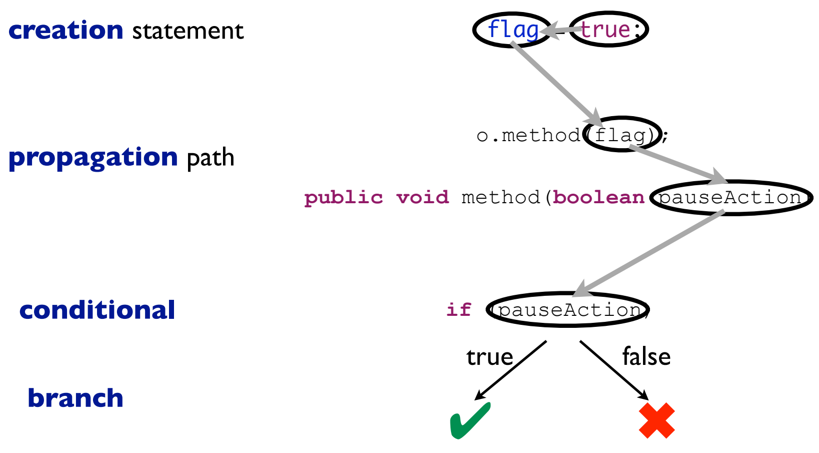


Figure 8.2. Constant controlled conditionals involve a creation statement, propagation path, conditional, and branch.

All of these are examples of infeasible paths caused by *constant-controlled conditionals*. A constant-controlled conditional is a conditional containing an expression that will always be one of several constants and whose possible values can be traced to one of several creation statements containing constants. These values may be propagated through variable assignments before reaching a conditional controlling the branch taken. Conditionals include both if statements and dynamic dispatch. Examples of constant-controlled conditionals include flags, anonymous classes and dynamic dispatch, and buses. Constant-controlled conditionals often capture how code behaves differently in different contexts. Through dynamic dispatch and flags, calling contexts control the behavior of code they invoke. Depending on the context, code may exhibit very different behavior.

For example, a bus lets a sender send a message to many recipients, who then decide, based on the type of the message, if any action will be taken. While many senders may send messages and many receivers may receive messages, individual senders may send a small number of messages that are acted on by a small number of receivers. Accurately distinguishing these connections greatly reduces the connectivity of the call graph – rather than all senders being connected to all receivers, only those senders and receivers that are actually related by sending and receiving a message are connected.

Infeasible paths make exploring code harder. In applications with extensive use of message passing or dynamic dispatch, this can be particularly problematic, implying that there are connections between portions of the codebase that are actually unconnected. When traversing through calls, developers must track the values of variables by hand to understand what is, in fact, connected, adding extra difficulty to the task. Moreover, developers often simply guess and make assumptions, resulting in incorrect beliefs about code (see Section 7.2.2.2).

This chapter describes a novel approach to eliminating some of the infeasible paths caused by constant-controlled conditionals quickly enough to be used in an interactive system: Fast Feasible Path Analysis (FFPA). FFPA constructs summaries describing possible paths through a method. When a search is executed, these summaries are used by an interprocedural analysis to partially path-sensitively propagate constants to determine which branches through conditionals are feasible. In my case studies, FFPA is able to generate call graphs using pre-computed summaries in 1 – 2 seconds of analysis time.

8.1. The static trace

FFPA represents feasible control flow paths through code as a *static trace.* A static trace is a graph in which each node is a statement or expression (subsequently referred to simply as statements) and directed edges denote possible successor statements. Like a control flow graph, statements may have multiple successor statements at conditionals or at loops. But when an analysis can determine that a path through a control flow branch is not feasible, this successor is not present. A static trace contains expression nodes only for the specific statements types of interest. Figure 8.3 shows a static trace for a toy example.

a)

```
public void r1() {
        a();
}

public void r2() {
        a();
}

public void a() {
        c();
        b(false);
        c();
        b(true);
}

public void b(boolean flag) {
        if (flag)
                Library.libraryCall();
}

public void c() {
         …
}
```

b)

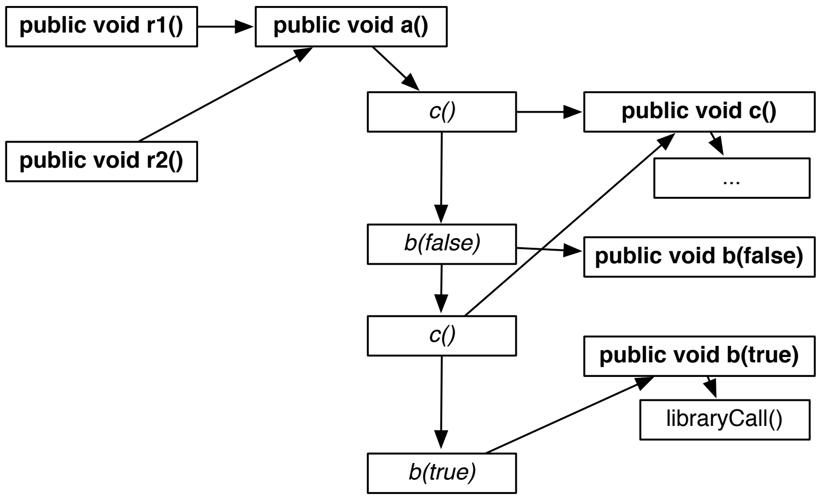


Figure 8.3. a) A short program and b) its static trace.

Like a dynamic trace, a static trace maps variables to values. Static traces can be constructed with variables mapped to values taken from a variety of possible sets. As FFPA constructs a static trace by propagating constants using a static analysis, static traces in FFPA map variables to either one of a small number of constants (true, false, or a runtime type) or the special value ⊤ ("top") signifying an unknown value. FFPA could easily be extended to include additional constants (e.g., string constants, numeric constants, enums). During construction (see Section 8.3.3), FFPA maintains a local store mapping variables to values, but this is not included in the final static trace. However, each statement in the static trace contains a reference to the corresponding Java statement or expression in the source.

Like a dynamic trace, a method may occur multiple times when reached along multiple paths. For example, in Figure 8.3, *b()* is present twice — once where flag is true and once where flag is false. However, when a method is called a second time with the same parameters, the previous trace is reused and an edge to the previous trace is created. For example, *c()* is called from two different call sites that share a single trace. Similarly, a recursive call to a method with the same parameters as a previous call reuses the trace. When the method has not previously been encountered with these parameters, the recursive call is unrolled, creating new method traces, until a previous method trace is encountered. Variables in a static trace may only take on a constant value. Thus, the recursion unrolling must terminate as there a finite number of parameters each of which may only take only one of a finite number of constant values (e.g., true, false, runtime types). In practice, this usually quickly terminates.

When FFPA cannot determine the path taken at a branch, the branch is flattened: statements in the first branch are appended, then the second. However, information is not arbitrarily shared between branches:

```
boolean flag = false;
if (x > 5)
{
      flag = m1();
      n(flag);
}
else
{
      m(flag);
}
```

is modeled as

```
n(m1());
m(false);
```

Even though the branch structure has been flattened, the value of flag when passed into m is (correctly) false, not the return value of m1() as it would be without the conditional.

Similarly, statements inside a loop are included once. The values of variables (and the paths these values make feasible) correspond to the values found after analysis of the loop has reached a fixed-point. Each value assigned to a variable in a loop represents an approximation of all values it might ever hold on any iteration of the loop.

A static trace describes feasible paths through a developer's code. At calls into library code for which a developer does not have source, a static trace contains a node for the call site but not the method itself. If the method returns a value, this value is $\top$.

Loops and recursion cause several complications. In a loop, a conditional *cond* determines whether the path into the loop or exiting the loop is taken. Loops create cycles in the CFG – statements inside a loop may execute many times. Like in data flow analysis [NNH04], a single, final context is computed for each statement, describing the statement's behavior across all iterations of the loop. Similarly, static traces include one method invocation per context for each method invocation in the source, with a computed context that approximates all possible contexts.

In a recursive call, a path of one or more method calls results in a method already on the stack being reentered and executed a second time. Concrete traces might traverse this cycle zero or more times. Static traces use two different devices to describe recursion. In some cases, the context in which the method is reentered may differ from the contexts already observed in the recursion. In this case, the static trace contains an additional copy of the method with executed statements in contexts from this new iteration. But the number of contexts in which a method can be observed is finite - there are a finite number of variables and a finite number of constant values which each may take (non-constant values are modeled as $\top$). Eventually, the method will be reentered in the same context in which it was entered on an earlier iteration. In this case, a back edge is created from the call site to an earlier portion of the static trace.

8.2. Analysis overview

FFPA generates a static approximation of paths that may execute. An alternative approach would be to use a dynamic analysis in which the user starts the program, enters input to demonstrate the situation of interest, and records an execution trace (c.f., [KM09]). A key advantage of a dynamic trace is that it is fully precise and there are no false positives – only statements that actually executed are included. But a static approach also enjoys several advantages. It permits reasoning about behavior that may not be evident in a single trace or even from many traces. Moreover, generating a dynamic trace is time-consuming for long running operations, difficult when special hardware or setup is required, or even impossible when it is unknown what input might cause the desired path to execute. For example, when debugging a failure reported from the field with only a stack dump as the indication of the problem, a developer may not know how to generate such a stack. Eliminating infeasible paths might enable developers to use static investigation in situations in which they are currently forced to use dynamic investigation.

FFPA consists of 3 phases: data flow analysis, summary construction, and static trace construction. In the dataflow analysis phase, it analyzes each method, finding possible intraprocedural values each variable might take. When a value cannot be determined, it is assigned the special value $\top$ ("top"), signifying no information. These values are then used in the next phase, summary construction, to build a graph of possible paths through each method, parameterized by the method's parameters and return values from callees [SP81]. After a developer invokes a search, an interprocedural analysis uses the summaries to construct a static trace, describing a call graph relative to the origin of the search.

In the first two phases, summaries are constructed and stored to disk for later use. Summaries only need be recomputed when a method's source changes. Thus, summary construction can be relatively slow. In contrast, the interprocedural analysis is executed after the developer invokes a search and before any results are shown. During this time, the user is waiting for a result. In order for the tool to feel interactive to the developer, a result must be obtained in a few seconds of analysis time. The results should also be precise and correspond mostly to feasible concrete traces. Thus, the two main goals of FFPA are performance and precision.

FFPA generates static traces for both downstream and upstream searches (see Section 9.2.1 for a description of the user interface for searching). Downstream searches begin at an origin method *o* and include all of the methods directly or indirectly called from *o*. FFPA terminates at methods invoked by the framework that work as cutpoints, isolating a portion of the call graph which executes in response to the call back. For upstream searches, roots (methods with no callers) which might cause *d* to execute are first identified by traversing a naïve call graph (without any infeasible paths eliminated). FFPA is then invoked for each of these roots, resulting in a set of static traces. Static traces that do not reach *d* are discarded.

8.3 Analysis approach

The goal of FFPA is to generate a static trace from a program *p* starting at a root statement *o*. FFPA approximates the results of a symbolic execution of *p* beginning at *o*. FFPA tracks constants through assignments and uses these values to determine which branch to take at a conditional. But FFPA does this modularly (per method), computing paths through a method symbolically, parameterized by the values of formal parameters and returns from method invocations. FFPA analyzes executed statements in different contexts (context sensitive), follows CFG paths (flow sensitive), and propagates distinct contexts to each branch at conditionals (branch sensitive). FFPA is partially path-sensitive as it sometimes eliminates infeasible paths.

FFPA tracks Boolean and type constants. Type constants are created at object allocation (i.e., `new Type()`) and at runtime type tests (i.e., `x instanceof Type`). FFPA could be extended to track other constants such as string constants, null, enumerated types, or numeric constants.

FFPA generates static traces in response to developer searches. One way to ensure performance would be to precompute static traces starting from every method in the program. However, after the developer edits the source, it would then be necessary to recompute all of the static traces. Thus, FFPA instead seeks to precompute as much as possible in method summaries, using only information local to the method. After an edit, only the affected summaries needs to be recomputed.

After FFPA has computed a static trace, this information may be reused. When a method is reached a second time in the same context, the previous results can be reused. Or when a second search is performed which includes parts of the same static trace, this information can be reused. FFPA maintains a cache of static traces indexed by method and context.

8.3.1 Dataflow analysis

Dataflow analysis and summary construction modularly build a summary for a method *m* which describes paths through *m*. Both phases are modular, using only information from *m* (i.e., intraprocedural information). Summaries are parameterized by the interprocedural sources through which values flow into *m* - formal parameters of *m* and return values of call sites in *m*. Sources and constants are propagated path-sensitively through assignment statements to sinks – actual parameters at call sites and return statements in *m*. Data is not tracked through fields – all reads from fields yield ⊤.

$v ::= t \mid f \mid T \mid \text{instanceof } T \mid \text{!instanceof } T \mid \top$	value
$s ::= m()_n \mid p$	source
$\gamma ::= a \mid \gamma \vee a \mid t$	path constraint
$a ::= \langle s, v\rangle \mid a \wedge \langle s, v\rangle$	and term
$le ::= v \mid \langle s, v\rangle \mid \bot$	lattice element
$\sigma ::= \{x \mapsto le\}$	tuple lattice element
$\omega ::= \{\langle\gamma, \sigma\rangle\}$	set lattice element
types T	
methods m	
formal parameters p	
program points n	
program variables x	

Figure 8.4. A syntax for data structures used in the data flow analysis.

Figure 8.4. lists the data structures used in the data flow analysis. FFPA tracks constant values, which include the Boolean constants t (true) and f (false). FFPA models the value of an object simply by its type. When an object allocation is seen (e.g., new T()), FFPA knows the exact type of the object, which it represents with the value T. In other cases, FFPA may only have a constraint on the type of the object, such as when a path goes through an instance of test. For these cases, FFPA uses the values instanceof T and !instanceof T. Finally, FFPA may see other values (e.g., 5) or may join two unequal values together. FFPA represents these other cases with the special value ⊤, which indicates that nothing is known about the value.

FFPA symbolically represents interprocedural information as sources. $m()_n$ indicates the return value of the call to the method m on line n. p is a formal parameter. In the analysis, FFPA builds path constraints γ describing the path along which information has originated. As a dataflow analysis, FFPA computes a map from edges in the control flow graph (points before and after statements) to a set lattice element ω. Each element in ω is a pair <γ, σ> indicating constraints γ on the path taken by information in the element and a tuple lattice element σ, which maps program variables to lattice elements. Each lattice element is either a value v, a source on which FFPA has computed a constraint of v, or ⊥, the initial value of variables before they have been assigned.

Figure 8.5 illustrates a simple example. At line 1, the source variable o is initialized to the source o and y is initialized to ⊤ as it is neither a Boolean variable nor an object. At line 2, the method call $n(f1, \top)_2$ is encountered, and the source for its return value is stored in f2. At line 3, the method call to l is encountered, but since it has no return value, there is no source to store.

```
1           public void a(Object o, int y) {
2                 boolean f2 = n(o, y);
3                 l(f2);
            }
```

ω = {< t, σ >} where σ is as follows after each line:

Line	o	y	f2
1	o	⊤	
2	o	⊤	$n(f1, \top)_2$
3	o	⊤	$n(f1, \top)_2$

Figure 8.5. A simple example of the dataflow analysis results for a method invocation.

The dataflow analysis is partially-path sensitive, tracking information for different paths in distinct σs. When the data flow analysis encounters a conditional *cond*, each σ is inspected to determine the branches to which it should be propagated. When *cond* contains an expression that σ maps to a Boolean source *s*, a *fork* occurs. This creates two new σs, replacing the existing σ, representing the case in which s is true and the case in which s is false. In the new σs, a constraint is added to s, mapping s to either <s, t> or <s, f>. To record this constraint on σ, the path constraint γ in < γ, σ> is conjoined with <s, t> (or as <s, f>), as appropriate.

```
1           public void a(Object o) {
2                 boolean r = false;
3                 if (o instanceof C)
4                       r = b();
5                 Object p = c();
6                 if (r)
7                       p = foo();
            }
```

1 ω = {< t, {o ↦ o>}
2 ω = {< t, {o ↦ o, r ↦ f>}
3 ω = {< <o, instanceof C>, {o ↦ <o, instanceof C>, r ↦ f >,
    < <o, !instanceof C>, {o ↦ <o, !instanceof C>, r ↦ f > >}
4 ω = {< <o, instanceof C>, {o ↦ <o, instanceof C>, r ↦ $b()_4$ }>}
5 ω = {< <o, instanceof C>, {o ↦ <o, instanceof C>, r ↦ $b()_4$, p ↦ $c()_5$ >,
    < <o, !instanceof C>, {o ↦ <o, !instanceof C>, r ↦ f , p ↦ $c()_5$ } >}
6 ω = {< <o, instanceof C> ∧ <$b()_4$, t>, {o ↦ <o, instanceof C>, r ↦ <$b()_4$, t>, p ↦ $c()_5$ >,
    {< <o, instanceof C> ∧ <$b()_4$, f>, {o ↦ <o, instanceof C>, r ↦ <$b()_4$, f>, p ↦ $c()_5$ >,
    < <o, !instanceof C> , {o ↦ <o, !instanceof C>, r ↦ f , p ↦ $c()_5$ } >}
7 ω = {< <o, instanceof C> ∧ <$b()_4$, t>, {o ↦ <o, instanceof C>, r ↦ <$b()_4$, t>, p ↦ $foo()_7$ >,

Figure 8.6. FFPA forks σs when sources are evaluated in conditional expressions.

Figure 8.6 illustrates the creation of forks. At line 3, FFPA encounters a conditional with the expression o instanceof C. FFPA forks the incoming σ, creating two new σs, one where the expression is true and the other in which it is false. This is stored both in path constraints on the new σs and in o. The σ where the expression is true is propagated into the true branch of the conditional. At 4, r is updated with the return value from b(). Before 5, the ωs from the true and false branch are combined. As the contents and path constraints differ for each σ, the resulting ω includes the σs from each. At 6, one of the σs maps r to false. This σ is not propagated into the conditional. The other σ maps r to a source, so this σ is forked on r, resulting in two new σs.

Other statements also evaluate expressions with Boolean values and may also fork contexts. At statements of the form *!x* or *x instanceof Type*, FFPA consults the context for the value of *x*. If *x* is a source, *x* is forked, creating a context in which *x* has a false constraint and a context in which *x* has a true constraint. If *x* has a value, this value is used.

As in standard dataflow analysis [NNH04], loops are analyzed iteratively until a fixed-point is reached and the results do not change. But forking loop guards poses a problem:

```
x = false;
while (foo())
     x = bar();
```

Forking `foo()` only in the loop's first iteration would create two σs – one with <foo(), t> and a second with <foo(), f>. However, the σ with <foo(), t> would never escape the loop. Alternatively, reforking foo() at every loop iteration solves this problem, but results in σs that traveled through the loop with path constraints including <foo(), t>. This causes it to not match the correct paths in the next phase. To solve this problem, loop guards are never forked. As a result, all loops are modeled as executing both at least once and zero times, with no fork distinguishing these cases.

```
        public void e() {
1               x = false;
2               while (foo())
3                       x = bar();
4               baz();
        }
```

1 $\omega = \{< t, \{x \mapsto f\}>\}$
2 $\omega = \{< t, \{x \mapsto f\}>\}$
3 $\omega = \{< t, \{x \mapsto \texttt{bar()}_2\}>\}$
2 $\omega = \{< t, \{x \mapsto \top\}>\}$
3 $\omega = \{< t, \{x \mapsto \texttt{bar()}_2\}>\}$
4 $\omega = \{< t, \{x \mapsto \top\}>\}$

Figure 8.7. As in standard dataflow analysis, loops are iterated until a fixed-point is reached.

Figure 8.7 illustrates FFPA's handling of loops. At line 2, foo() is not forked, as it is in the guard for a loop. Instead, one copy of σ is propagated into the loop (the true branch) and another out of the loop (the false branch). After analyzing 3, FFPA continues to iterate the loop until a fixed-point is reached. The ω after 3 is joined with the ω incoming into the loop, resulting in x being mapped to ⊤. Finally, after a fixed point is reached, FFPA analyzes line 4 and the outgoing ω from inside the loop is joined with the ω sent on the false branch, resulting in x again being ⊤.

Consider a series of uncorrelated conditionals (e.g., the values of $x_1 \dots x_n$ are unrelated):

```
if (x1)
        ...
...
if (xn)
     ...
```

When $x_1 \dots x_n$ are each source variables, each conditional will result in all of the contexts being forked. As a result, the final number of contexts will be exponential in the number of statements. To prevent this, FFPA joins contexts when the path-sensitivity provided by keeping them distinct no longer provides any benefit in precision. This occurs for variables that are dead and will never be read again in *m*. Contexts that differ only in dead variables always follow the same paths through *m*. FFPA employs a path-insensitive, flow-sensitive intraprocedural live variable analysis to compute a set of dead variables before every statement. Entries for dead variables are removed from all contexts, and identical contexts are joined. In the best case – a list of uncorrelated conditionals where the variables written by a conditional are dead before the next conditional – FFPA maintains at most two contexts at any statement.

```
1       public void d(boolean a, boolean b, boolean c) {
2               if (a)
3                       foo1(b);
4               if (b)
5                       foo2();
6               if (c)
7                       foo3();
        }
```

```
1       ω = {< t, {a ↦ a, b ↦ b, c ↦ c}>}
2       ω = {< <a, t>, {a ↦ <a, t>, b ↦ b, c ↦ c }>, < <a, f>, {a ↦ <a, f>, b ↦ b, c ↦ c }>} }
3       ω = {< <a, t>, {b ↦ b, c ↦ c }>} }
4       ω = {< <b, t>, {b ↦ <b, t>, c ↦ c }>, < <b, f>, {b ↦ <b, f>, c ↦ c }>} }
5       ω = {< <b, t>, {c ↦ c }> }
6       ω = {< <c, t>, {c ↦ <c, t> }>, < <c, f>, {c ↦ <c, f> }>} }
7       ω = {< <c, t>, {}> }
```

Figure 8.8. FFPA eliminates dead variables from σ, allowing identical tuple lattice elements to be combined.

Eliminating dead variables allows FFPA to only require a constant number of σs for a series of uncorrelated conditionals, as illustrated in Figure 8.8. At line 1, the parameters are initialized. At line 2, FFPA encounters a conditional with a Boolean variable. A fork occurs, and the current σ is split into two σs, one constraining a to be true and one constraining it to be false. The σ with a true is propagated along the true branch of the conditional and into line 3. At line 3, the call to foo1() occurs, but there is no return value, so nothing is updated. But the variable a is now dead, so it is dropped from σ. Before line 4, the ωs from the true and false branch are joined. As each has σs with different constraints, the new ω includes both. But, the σs are identical, so they are joined into a single context. Taking the disjunction of the path constraints <a, t> and <a, t> yields a true path constraint. But, the whole forking process begins again with b – two new σs are created with b true and b false. The σ with b true is propagated to 5, and the paths are recombined at 6. Finally, the process repeats on lines 6 and 7 with c. By removing the variables with constraints as soon as they become dead, the differences between σs are eliminated, allowing them to be joined together, and preventing an exponential blowup in the number of σs.

In practice, this approach usually results in a small number of σs. However, loops defeat the live variable analysis, as variables may be read on subsequent iterations of the loop. Thus, uncorrelated conditionals in a loop still result in an exponential number of tuple lattice elements. To ensure that code with deeply nested loops with many conditionals does not exhaust memory or time bounds, new contexts are not created after 1200 have already been created in a method. This number was chosen to limit memory usage. In this case, new conditionals with sources in the guards are treated as if they did not have sources (e.g., if (x) is modeled the same as if (x > 5)).

```
        public void f(boolean f1, boolean f2) {
1             Boolean x = false;
2             int i = 0;
3             while (i < 10) {
4                   if (f1)
5                         x = foo(x);
6                   if (f2)
7                         x = bar();
8                   i++;
9             };
        }
```

1 ω = {< t, {f1 ↦ f1, f2 ↦ f2, x ↦ f }>}
2 ω = {< t, {f1 ↦ f1, f2 ↦ f2, x ↦ f, i ↦ ⊤}>}
3 ω = {< t, {f1 ↦ f1, f2 ↦ f2, x ↦ f, i ↦ ⊤}>}
4 ω = {< <f1, t>, {f1 ↦ <f1, t>, f2 ↦ f2, x ↦ f, i ↦ ⊤}>}>,
< <f1, f>, {f1 ↦ <f1, f>, f2 ↦ f2, x ↦ f, i ↦ ⊤}>}>
5 ω = {< <f1, t>, {f1 ↦ <f1, t>, f2 ↦ f2, x ↦ foo()$_5$, i ↦ ⊤}>}>}>
6 ω = {< <f1, t> ∧ <f2, t>, {f1 ↦ <f1, t>, f2 ↦ <f2, t>, x ↦ foo()$_5$, i ↦ ⊤}>}>,
< <f1, t> ∧ <f2, f>, {f1 ↦ <f1, t>, f2 ↦ <f2, f>, x ↦ foo()$_5$, i ↦ ⊤}>}>,
< <f1, f> ∧ <f2, t>, {f1 ↦ <f1, f>, f2 ↦ <f2, t>, x ↦ f , i ↦ ⊤}>}>,
< <f1, f> ∧ <f2, f>, {f1 ↦ <f1, f>, f2 ↦ <f2, f>, x ↦ f , i ↦ ⊤}>}>}
7 ω = {< <f1, t> ∧ <f2, t>, {f1 ↦ <f1, t>, f2 ↦ <f2, t>, x ↦ bar()$_7$, i ↦ ⊤}>}>,
< <f1, f> ∧ <f2, t>, {f1 ↦ <f1, f>, f2 ↦ <f2, t>, x ↦ bar()$_7$, i ↦ ⊤}>}>}
8 ω = {< <f1, t> ∧ <f2, t>, {f1 ↦ <f1, t>, f2 ↦ <f2, t>, x ↦ bar()$_7$, i ↦ ⊤}>}>,
< <f1, t> ∧ <f2, f>, {f1 ↦ <f1, t>, f2 ↦ <f2, f>, x ↦ foo()$_5$, i ↦ ⊤}>}>,
< <f1, f> ∧ <f2, t>, {f1 ↦ <f1, f>, f2 ↦ <f2, t>, x ↦ bar()$_7$, i ↦ ⊤}>}>,
< <f1, f> ∧ <f2, f>, {f1 ↦ <f1, f>, f2 ↦ <f2, f>, x ↦ f , i ↦ ⊤}>}>}
3 ω = {< t, {f1 ↦ f1, f2 ↦ f2, x ↦ f, i ↦ ⊤}>,
< <f1, t> ∧ <f2, t>, {f1 ↦ <f1, t>, f2 ↦ <f2, t>, x ↦ bar()$_7$, i ↦ ⊤}>}>,
< <f1, t> ∧ <f2, f>, {f1 ↦ <f1, t>, f2 ↦ <f2, f>, x ↦ foo()$_5$, i ↦ ⊤}>}>,
< <f1, f> ∧ <f2, t>, {f1 ↦ <f1, f>, f2 ↦ <f2, t>, x ↦ bar()$_7$, i ↦ ⊤}>}>,
< <f1, f> ∧ <f2, f>, {f1 ↦ <f1, f>, f2 ↦ <f2, f>, x ↦ f , i ↦ ⊤}>}>}

4 ω = { < <f1, t>, {f1 ↦ <f1, t>, f2 ↦ f2, x ↦ f, i ↦ ⊤}>}>,
< <f1, f>, {f1 ↦ <f1, f>, f2 ↦ f2, x ↦ f, i ↦ ⊤}>}>
< <f1, t> ∧ <f2, t>, {f1 ↦ <f1, t>, f2 ↦ <f2, t>, x ↦ bar()$_7$, i ↦ ⊤}>}>,
< <f1, t> ∧ <f2, f>, {f1 ↦ <f1, t>, f2 ↦ <f2, f>, x ↦ foo()$_5$, i ↦ ⊤}>}>,
< <f1, f> ∧ <f2, t>, {f1 ↦ <f1, f>, f2 ↦ <f2, t>, x ↦ bar()$_7$, i ↦ ⊤}>}>,
< <f1, f> ∧ <f2, f>, {f1 ↦ <f1, f>, f2 ↦ <f2, f>, x ↦ f , i ↦ ⊤}>}>}

5 ω = {< <f1, t>, {f1 ↦ <f1, t>, f2 ↦ f2, x ↦ foo()$_5$, i ↦ ⊤}>}>,
< <f1, t> ∧ <f2, t>, {f1 ↦ <f1, t>, f2 ↦ <f2, t>, x ↦ foo()$_5$, i ↦ ⊤}>}>,
< <f1, t> ∧ <f2, f>, {f1 ↦ <f1, t>, f2 ↦ <f2, f>, x ↦ foo()$_5$, i ↦ ⊤}>}>}

6 ω = { < <f1, t> ∧ <f2, t>, {f1 ↦ <f1, t>, f2 ↦ <f2, t>, x ↦ foo()$_5$, i ↦ ⊤}>}>,
< <f1, t> ∧ <f2, f>, {f1 ↦ <f1, t>, f2 ↦ <f2, f>, x ↦ foo()$_5$, i ↦ ⊤}>}>,
< <f1, f> ∧ <f2, t>, {f1 ↦ <f1, f>, f2 ↦ <f2, t>, x ↦ bar()$_7$, i ↦ ⊤}>}>,
< <f1, f> ∧ <f2, f>, {f1 ↦ <f1, f>, f2 ↦ <f2, f>, x ↦ f , i ↦ ⊤}>}>}

7 ω = {< <f1, t> ∧ <f2, t>, {f1 ↦ <f1, t>, f2 ↦ <f2, t>, x ↦ bar()$_7$, i ↦ ⊤}>}>,
< <f1, f> ∧ <f2, t>, {f1 ↦ <f1, f>, f2 ↦ <f2, t>, x ↦ bar()$_7$, i ↦ ⊤}>}>}

```
8        ω = {< <f1, t> ∧ <f2, t>, {f1 ↦ <f1, t>, f2 ↦ <f2, t>, x ↦ bar()₇, i ↦ ⊤}>}>,
              < <f1, t> ∧ <f2, f>, {f1 ↦ <f1, t>, f2 ↦ <f2, f>, x ↦ foo()₅, i ↦ ⊤}>}>,
              < <f1, f> ∧ <f2, t>, {f1 ↦ <f1, f>, f2 ↦ <f2, t>, x ↦ bar()₇, i ↦ ⊤}>}>,
              < <f1, f> ∧ <f2, f>, {f1 ↦ <f1, f>, f2 ↦ <f2, f>, x ↦ f, i ↦ ⊤}>}>}

9        ω = { < t, {f1 ↦ f1, f2 ↦ f2, x ↦ f, i ↦ ⊤}>,
              < <f1, t> ∧ <f2, t>, {f1 ↦ <f1, t>, f2 ↦ <f2, t>, x ↦ bar()₇, i ↦ ⊤}>}>,
              < <f1, t> ∧ <f2, f>, {f1 ↦ <f1, t>, f2 ↦ <f2, f>, x ↦ foo()₅, i ↦ ⊤}>}>,
              < <f1, f> ∧ <f2, t>, {f1 ↦ <f1, f>, f2 ↦ <f2, t>, x ↦ bar()₇, i ↦ ⊤}>}>,
              < <f1, f> ∧ <f2, f>, {f1 ↦ <f1, f>, f2 ↦ <f2, f>, x ↦ f, i ↦ ⊤}>}>}
```

Figure 8.9. Uncorrelated conditionals in a loop result in an exponential number of tuple lattice elements being created.

Figure 8.9 illustrates what happens when uncorrelated conditionals occur in a loop, causing the number of σs to grow exponentially. At line 4, FFPA performs the first fork, resulting in two σs. But, at line 6, x is not dead, so it is not removed from the tuple lattice. As a result, the σs created by the fork on f1 still differ and are not combined when the paths join before 6. Similarly, FFPA forks again on line 6, resulting in 4 σs. After line 8, the ω that went through the loop is joined with the initial incoming ω into the loop, and FFPA continues until the results for statements in the loop reach a fixed-point.

While dead variables do not influence the path a context follows, dead variables still indicate the path already followed. This information is needed in summary construction (see 8.3.2) to determine the path to which the context should be matched. To maintain this information while preventing an exponential blowup in the number of contexts, each context includes a path constraint γ containing constrained sources in disjunctive normal form. Whenever a fork occurs, the constrained source is conjoined with each term in the path constraint. When identical contexts are joined at control flow merges, the new context's path constraint is the disjunction of tuple lattices, simplified where possible.

8.3.2 Summary construction

sn ::= [c] stmt_1 … stmt_i	summary node
c ::= <s1, v1>	constraint

stmt is a method call or return statement where all expressions have been substituted with lattice elements.

Figure 8.10. The data structures used in summaries.

Summaries model paths through a method, parameterized by formal parameters and the return values of method calls. Any information irrelevant to constructing a static trace is removed, reducing the number of statements along each path. Each path consists of a list of method call and return statements, where all expressions they contain have been substituted with lattice elements.

Paths are represented through summary nodes (see Figure 8.10). Each summary node consists of (optionally) a constraint on a source variable and a list of statements. When a fork is encountered, child summary nodes will be created with both true and false constraints.

Representing summaries as a tree of summary nodes would be highly inefficient, resulting in an exponential number of branches in the tree. Fortunately, many paths contain identical portions. Therefore, FFPA represents summaries as a directed acyclic graph of summary nodes. When two paths contain identical suffixes, the paths can share common summary nodes. The two paths are linked together by connecting to a common summary node.

As summary construction occurs, several pieces of data are tracked. The current leaf nodes of the summary graph are tracked. And, for each of these nodes, the complete path constraint from the root of the summary graph is tracked (but only the last constraint – the constraint on the summary node – is stored in the summary node). Summary nodes with constraints inherit their parents' constraints and add their own constraint. Common summary nodes inherit the intersection of their parents' constraints.

Summary construction begins with a single empty leaf node. For each method call or return statement, in textual order, FFPA finds the corresponding ω computed in the data flow analysis and continues as follows:

1. Each tuple lattice element pair <γ, σ> in ω is matched to compatible summary leaf nodes (leaf nodes with no children). A <γ, σ> is compatible with a summary node if γ does not conflict with the summary node's complete path constraint. Conflicts occur when they are mutually exclusive (e.g, <*x, f*> and <*x, t*>). Note that either context may contain additional constraints and still be compatible. Each leaf node may match zero or more <γ, σ> pairs. Leaf nodes that match at least one <γ, σ> pair are active leaf nodes.
2. Active leaf nodes are scanned for opportunities to create shared summary nodes. A shared summary node is created when a node and its sibling are active, match the same <γ, σ> pair, and the source variable in the constraint in which it differs from its sibling is dead (note: there can only be one source variable that differs as summary nodes are created and combined in pairs with a constraint on a single variable distinguishing them). This ensures that these summary nodes will not be recreated when processing a future statement, as the source variable will not be referenced further. When a shared node is created, FFPA returns to step 1.
3. Children for summary nodes may be created. If the γ in the <γ, σ> pair that matches a node contains a constraint that is not in the summary node's complete path constraint, this constraint is observed. Observing a constraint creates new true and false summary child nodes. When this occurs, FFPA returns to step 1.
4. When a <γ, σ> has matched a summary node, the statement is added to the summary node. For each summary node, all matching σs are joined, creating a new σ. Every expression in the statement is then substituted for its value in the new σ.

```
1      public void d(boolean a, boolean b, boolean c) {
2            if (a)
3                  foo1(b);
4            if (b)
5                  foo2();
6            if (c)
7                  foo3();
       }
```

1 ω = {< t, {a ↦ a, b ↦ b, c ↦ c}>}
2 ω = {< <a, t>, {a ↦ <a, t>, b ↦ b, c ↦ c }>, < <a, f>, {a ↦ <a, f>, b ↦ b, c ↦ c }>} }
3 ω = {< <a, t>, {b ↦ b, c ↦ c }>} }
4 ω = {< <b, t>, {b ↦ <b, t>, c ↦ c }>, < <b, f>, {b ↦ <b, f>, c ↦ c }>} }
5 ω = {< <b, t>, {c ↦ c }> }
6 ω = {< <c, t>, {c ↦ <c, t> }>, < <c, f>, {c ↦ <c, f> }>} }
7 ω = {< <c, t>, {}> }

Starting values: empty summary node sn_1

ANALYZING LINE 3

1. sn_1 compatible with < <a, t>, {b ↦ b, c ↦ c }>
2. No shared nodes created
3. Constraint on *a* observed. sn_2 = [<*a, t*>] sn_3 = [<*a, f*>]

1. sn_2 = [<*a, t*>] compatible with < <a, t>, {b ↦ b, c ↦ c }> sn_3 = [<*a, f*>] compatible with nothing
2. No shared nodes created.
3. No children created.
4. sn_2 = [<*a, t*>] foo1(b); sn_3 = [<*a, f*>]

ANALYZING LINE 5:

1. sn_2 compatible with < <b, t>, {c ↦ c }> sn_3 compatible with < <b, t>, {c ↦ c }>
2. Empty shared node sn_4 created as child of sn_2 and sn_3

1. sn_4 compatible with < <b, t>, {c ↦ c }>,
2. No shared nodes created
3. Constraint on b observed. sn_5 = [<*b, t*>] sn_6 = [<*b, f*>]

1. sn_5 compatible with < <b, t>, {c ↦ c }> sn_6 compatible with nothing
2. No shared node created.
3. No child nodes created.
4. sn_5 = [<*b, t*>] foo2(); sn_6 = [<*b, f*>]

ANALYZING LINE 7:

1. sn_5 compatible with < <c, t>, {} > sn_6 compatible with < <c, t>, { }>
2. Empty shared node sn_7 created as child of sn_5 and sn_6

1. sn_7 compatible with < <c, t>, {} > >
2. No shared nodes created
3. Constraint on c observed. sn_8 = [<*c, t*>] sn_9 = [<*c, f*>]

1. sn_8 compatible with < <c, t>, {} > sn_9 compatible with nothing
2. No shared node created.
3. No child nodes created.
4. sn_8 = [<*c, t*>] foo3(); sn_9 = [<*c, f*>]

Figure 8.11. The construction of a summary is illustrated, continuing the example from Figure 8.8.

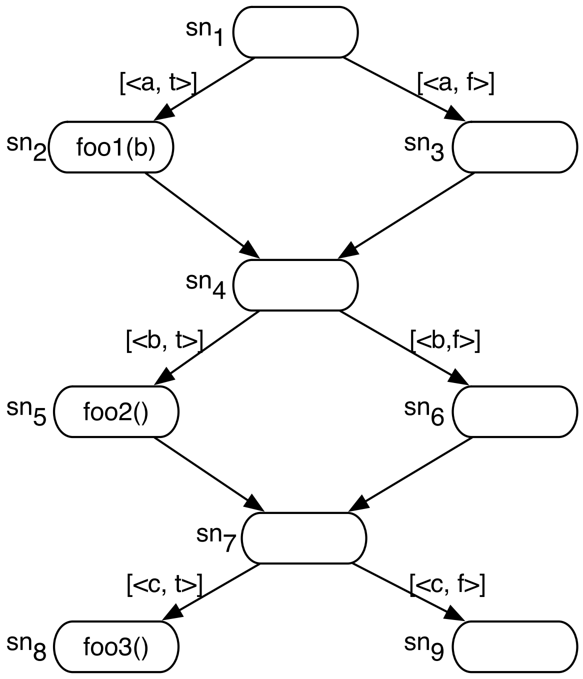


Figure 8.12. The final summary computed in the example from Figure 8.11.

Figure 8.11 illustrates summary construction. Summary construction begins with a single, empty, summary sn_1. FFPA first finds all method calls and return statements (at lines 3, 5, and 7) and analyzes each in turn. At line 3, FFPA uses the ω computed in the dataflow analysis to process the call to foo1(). sn_1 is found to be compatible with the only σ in ω and matches. No shared nodes are created, but the constraint on a in σ is observed, causing two new summary nodes to be created. As a result, FFPA returns to step 1 and matches the σ to sn_2. Finally, as only sn_2 matched a σ, foo() is added to this summary node and the parameter is substituted for its value in σ which is also b.

At line 5, FFPA processes the call to foo2() with the results computed by the dataflow analysis for line 5. The same σ matches both summary nodes. In the next step, FFPA examines if a shared node should be created. FFPA find that sn_2 and its sibling sn_3 are both active (they matched a σ), matched the same σ, and the variable whose constraints caused their creation (a) is now dead, indicating that the precision gained by having separate summary nodes for a will not be needed in the future. Thus, FFPA creates a new shared summary node sn_4. Returning to step 1, sn_4 is compatible with the only σ. No shared node is created. But, in step 3, a constraint on b is observed, resulting in two new child nodes being created. Thus, FFPA again returns to step 1. The σ is found to be compatible with sn_5. No shared nodes are created, and no children are created. FFPA reaches step 4. As only sn_5 matched a σ, foo2() is added to sn_5.

At line 7, FFPA processes the call to foo3(). As in line 5, FFPA again first creates a shared node, then two child nodes of the shared node (observing the constraint on c), and finally adds foo3() to sn_8. Figure 8.12 depicts the final summary.

8.3.3 Static trace construction

Static trace construction uses summaries to construct a static trace. FFPA works similarly to an interpreter: statements in summaries are executed, updating variables in local stores. When a method is invoked, actual parameters are first bound to formal parameters using the local store. The current stack frame is then pushed onto the call stack and a new stack frame created with a new local store. Interpretation continues in the callee with the new stack frame. After it returns to the caller, the caller's local store is updated with its return value.

As a method is interpreted, paths through the summary graph are traversed. At each step, the next statement to be executed is found. When a summary node with a constraint is encountered, the local store is examined. If the constraint is shown true, execution continues with the summary node. If it is false, execution continues with its sibling summary node (whose constraint must be true). Otherwise, execution continues with both summary nodes. When simultaneously executing multiple summary nodes, the same statement (labeled by source location) may occur in more than one summary node, but with different values. When this occurs, the values are joined: whenever they are not equal, they are $\top$.

FFPA may find recursive calls to methods currently on the stack. Whenever a method is invoked in the same context in which it is currently in the method stack, the call is not executed. Instead, a reference to the frame is appended and the return value is mapped to $\bot$ ("bottom") in the local store. $\bot$ is a special value denoting no information. While joining $\top$ with any value is $\top$, joining $\bot$ with a value is the value. FFPA does not iterate recursive calls to an interprocedural fixed-point, making it potentially unsound (it may not include some calls that are actually feasible). It is unknown how frequently this occurs in practice.

```
public void e() {
   d(true, false, true);
}
```

e’s summary is sn_e *= d(t, f, t);*

```
public void d(boolean a, boolean b, boolean c) {
      if (a)
            foo1(b);
      if (b)
            foo2();
      if (c)
            foo3();
}
```

d’s summary is shown in Figures 8.11 and 8.12

1. Analyze method e. Local store initialized as {}
2. Read statement *d(t, f, t);* from summary
3. Add to static trace for e. st_e *= d(t, f, t);*
4. Push onto stack frame. Go to d.

5. Local store initialized as {a ↦ t, b ↦ f, c ↦ t}
6. sn_1 traversed, no statements found.
7. children sn_2 and sn_3 found. sn_2 executes.
8. Read statement `foo1(b) from` sn_2.
9. Add to static trace for d. st_d = foo1(f);
10. foo1(f) is library call – no further action taken.
11. Encounter empty summary sn_4
12. children summaries sn_5 and sn_6 found. sn_6 executes.
13. Encounter empty summary sn_6
14. Encounter empty summary sn_7
15. children sn_8 and sn_9 found. sn_8 executes.
16. Read statement foo3() from sn_8.
17. Add to static trace for d. st_d = foo1(f); foo3();
18. foo3() is library call – no further action taken.
19. Done with constructing static trace d. Return to e.

20. Done with constructing static trace e.

Figure 8.13. An example of static trace construction.

Figure 8.13 illustrates constructing a static trace, continuing the summary construction example. The summary for method e, which calls d is shown. Starting at method e, FFPA first initializes the local store to the empty set and begins at the first summary node for e, sn_e. FFPA executes the first statement, d(t, f, t) and adds it to the static trace for e. As this is a method invocation, FFPA next pushes the analysis of e onto the stack frame and begins analyzing method d.

Using the actual parameters at the call site in d, the local store is initialized with values for the formal parameters a, b, and c. FFPA begins at the first summary node for d, sn_1, and looks for the next statement to execute. As the summary node is empty, FFPA next looks at its children, each of which have constraints on a. Examining the local store for a value of a, FFPA finds that it is true. Thus, FFPA continues execution at sn_2. FFPA now reads the statement foo1(b) from the summary. The local store maps b to f, so this is used for b's value. This is added to the static trace. As foo1() is a library call, FFPA does not execute foo1().

There are no additional statements in sn_3. Its child summary node sn_4 is empty. Next, FFPA encounters the children sn_5 and sn_6, which each contain constraints on b. Finding that b is false in the local store, FFPA continues execution with sn_6, which is empty. Its child, sn_7, is also empty. Its children sn_8 and sn_9 have constraints on c. FFPA examines the local store and finds that it is true, so it continues execution with sn_8. FFPA reads the statements foo3() from sn_8, executes it, and adds it to the static trace for d. As there are no more summary nodes, FFPA has finished constructing the static trace for d.

FFPA pops a stack frame from the stack, returning to the execution of the summaries for e. As it has reached the end of the summary node and there are no more summary nodes, FFPA completes the construction of the static trace for e. Figure 8.14 shows the final static traces produced.

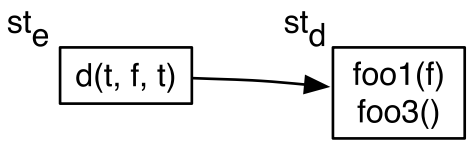


Figure 8.14. The final static trace produced by the example in Figure 8.13.

8.4. Implementation

FFPA is implemented as a static analysis for Java. It uses the Crystal static analysis framework[8], a data flow analysis framework for Java. Crystal relies on Eclipse's AST (abstract syntax tree) infrastructure. Summaries are stored to disk using the Java serialization framework[9].

[8] http://code.google.com/p/crystalsaf/

[9] http://docs.oracle.com/javase/6/docs/api/java/io/Serializable.html

8.5. Evaluation

FFPA is able to eliminate infeasible paths caused by constant-controlled conditionals when it is able to determine a variable's value at a conditional statement. More formally, FFPA is able to resolve a conditional *cond* evaluating a source *s* by determining which branch is followed when σ maps *s* to a constant *c*. Let $rd_1 ... rd_n$ be the reaching definitions of *s* (infeasible or feasible). FFPA determines *s* is a constant *c* at *cond* iff all reaching definitions rd_k of *s* are of one of three cases:

- FFPA found no feasible path from rd_k to *cond*
- rd_k assigns *c* to *s*
- rd_k assigns s_c to s and, at rd_k , s_c is mapped to c.

These conditions inductively describe propagation paths by which *s* acquires a value *c*. In the simplest case, a single assignment statement assigns *s* to *c* which is read by *cond*. But, more generally, *c* may be propagated from multiple creation statements through assignment statements to reach *cond*. One important requirement is that the creation statements must all occur after the origin *o*. This is not the case for propagation paths originating from a library call or callback.

The examples throughout Section 8.3 illustrate the potential precision benefits of FFPA. In these examples, FFPA is able to use both instanceof constraints and Boolean flags to eliminate paths through methods that are infeasible. More work remains to be done to evaluate the frequency of FFPA's benefits in practice.

An ideal way to evaluate the precision of FFPA would be to demonstrate its ability to reduce the number of infeasible paths found in a call graph. One way to measure this would be to take pairs of methods in a codebase and count the number of paths between them. Averaged over many methods and, perhaps, biased towards pairs of nodes that might be more typical of real searches (e.g., paths from user events to library calls), this would provide a quantitative measure of precision. Unfortunately, counting the number of paths between two nodes in an arbitrary graph is in the complexity class #P [V79], and thus there are no known polynomial algorithms for doing so. Thus, approximation algorithms, algorithms designed specifically for the types of graphs in call graphs, or heuristics (e.g., bounding the lengths of paths) would be required.

FFPA's performance has been established through use of REACHER in user studies. In the two lab studies of REACHER (see Chapter 10), participants ran FFPA throughout their tasks. Participants did not experience any interruptions waiting for FFPA: all queries completed in under two seconds; most finished significantly faster. Summaries were constructed for the entire codebase in tens of minutes before participants began and saved to disk.

FFPA employs several optimizations which may result in false negatives (not showing a call that could execute) and thereby make it unsound. FFPA does not iterate interprocedural call graph cycles to a fixed-point. And loops are always modeled as executing at least once. These optimizations may compromise FFPA's soundness (causing false negatives as calls that should be present are missed). While false negatives seem theoretically possible, no examples have been observed in practice.

Two of the most similar algorithms to FFPA are the Cartesian Product Algorithm (CPA) [A95] and the Simple Class Set algorithm (SCS) [GDD97]. CPA is a call graph construction algorithm which uses an interprocedural dataflow analysis. At call sites, CPA takes the Cartesian product of all subtypes of the receiver and actual parameters' static types. With each combination, CPA analyzes the callee, caching the result so that it can be reused at other call sites. The SCS algorithm works similarly, but instead uses the static types of the receiver and parameters. If CPA or SCS were to be extended to include Boolean flags, as well as types, the resulting algorithm would be similar to FFPA. Like FFPA, both would cache results when seen with identical parameters.

However, FFPA differs in several aspects. Most directly, CPA and SCS only track types, not Boolean flags. CPA and SCS are not partially-path sensitive. Thus, CPA and SCS never determine which branch to follow at a conditional other than dynamic dispatch sites, causing them to be less precise (include more false positives) than FFPA. CPA and SCS are interprocedural dataflow analyses, and do not employ FFPA's (possibly unsound) optimizations such as not iterating interprocedural call graph cycles to a fixed-point and modeling calls within a loop with a single calling context. In these cases, FFPA may have false negatives that CPA and SCS do not. Finally, CPA analyzes each method invocation using the Cartesian product of the receiver and parameter types, while FFPA and SCS do not. This may allow CPA to be more precise in these situations, although such examples have not been observed in practice. Moreover, this optimization reduces the performance of CPA, compared to FFPA, as each method invocation may generate many method contexts to analyze.

8.6. Conclusions

FFPA produces call graphs free of some of the common types of infeasible paths caused by constant-controlled conditionals. In contrast to slow but highly precise analyses such as model checkers, FFPA only tracks constants. But, as a result, FFPA can generate call graphs in only a few seconds of analysis time while eliminating some of the infeasible paths caused by constant-controlled conditionals. The next chapter describes REACHER, an interactive tool for exploring call graphs implemented using FFPA.

9. REACHER: SEARCHING ALONG CONTROL FLOW [10]

Studies of developers exploring and understanding code (Chapters 4-7) demonstrate that today's tools make answering reachability questions challenging. While today's tools let developers traverse across calls, developers are forced to guess which paths are feasible and lead to their targets. But what if developers instead used a tool that helped answer reachability questions?

This chapter describes the interaction design of REACHER, a tool for helping developers understand and explore code and answer reachability questions. Implemented as an Eclipse plugin for Java, REACHER lets developers search along call graphs, make sense of call graphs, and stay oriented as they navigate through code. See Figure 9.1 for an example of REACHER'S call graph visualization. Table 9.1 describes how REACHER'S design was shaped by my studies of code exploration. Section 9.4 analyzes the applicability of REACHER'S design to the observed reachability questions, highlighting several opportunities for additional features.

9.1. An example

To see REACHER in action, consider the challenging debugging task from Section 7.4.1. A developer debugging a null pointer exception tried to understand how `XMLRetriever.getStartContext()` could ever be called without `XMLRetriever.retrieveRelationships()` being called first. Working in a codebase she had written herself, she spent 40 minutes answering this question, using the debugger to inspect values and statically browsing. The task was hard because 96 paths connected these methods, some as long as 13 calls. Manually navigating and making sense of these paths was challenging.

REACHER makes this task easier by automating the search and visualizing the relevant portion of the call graph. We illustrate this with a scenario of how the developer might have instead worked using REACHER (see Figure 9.2). After opening `XMLRetriever.getStartContext()` in a Eclipse editor, she selects the method declaration and opens a context menu. She searches along paths to the selected method by selecting search upstream. Moving her cursor to the textbox in the REACHER Search view (upper right), she

[10] This chapter based on work previously published in [LM11].

searches for connections to the other method – `XMLRetriever.retrieveRelationships()` – by typing “retri”. As she types each character, REACHER lists matching statements below. Seeing `retrieveRelationship()` in the list, she clicks it, adding it to the call graph visualization below.

Study result	Design recommendation
Developers search for statements by attribute (e.g., field writes) and partial name.	Provide developers a configurable search dialog. As developers begin to enter search terms, immediately show matching statements.
Developers rapidly investigate, never returning to most methods.	Provide expandable details on demand, and let developers go back and forward with a browser style history navigation.
Developers explore huge call graphs, but the task relevant portion is small.	Only depict the (task relevant) methods developers select in a call graph visualization.
Developers reason about causality, class membership, ordering, choice, and repetition.	Provide developers an overview of this information in the call graph visualization.
Developers get lost and disoriented reading code in disparate places.	Let developers use the call graph visualization to navigate their editor view.

Table 9.1. A summary of findings from studies of code exploration and the resulting design requirements for tool support. REACHER incorporates all of these design recommendations in its design.

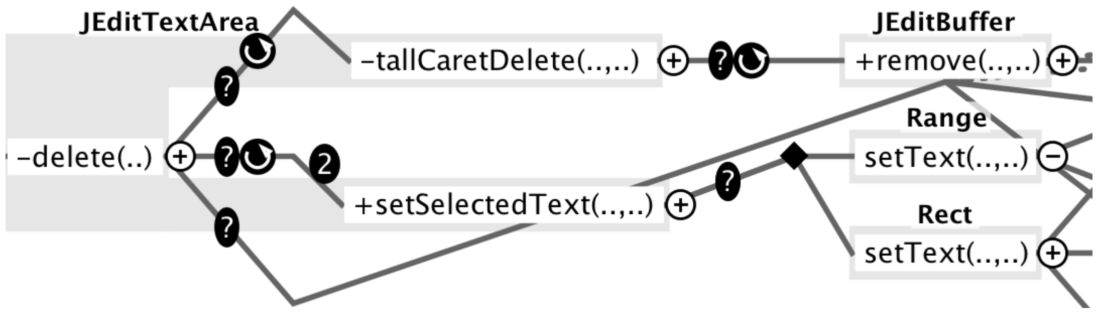
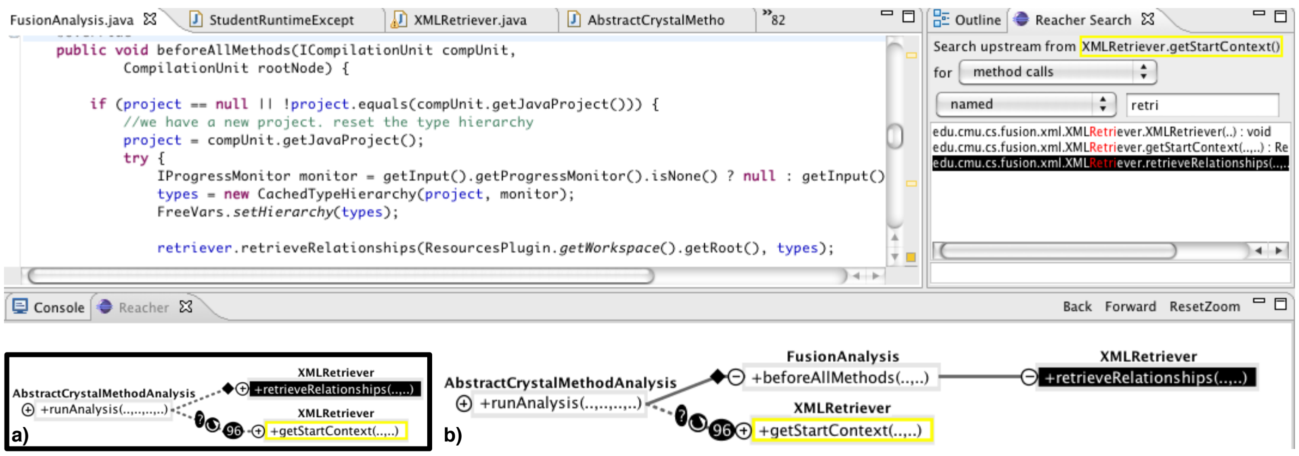


Figure 9.1. REACHER’s call graph visualization supports reasoning about interprocedural control flow. For example, this visualization illustrates that `JEditTextArea.delete(..)` – on the far left – may call `JEditTextArea.tallCaretDelete(.., ..)` several times in a loop before it may call `JEditTextArea.setSelectedText(..,..)` at two different call sites within a loop.



```
    public void beforeAllMethods(ICompilationUnit compUnit,
            CompilationUnit rootNode) {

        if (project == null || !project.equals(compUnit.getJavaProject())) {
            //we have a new project. reset the type hierarchy
            project = compUnit.getJavaProject();
            try {
                IProgressMonitor monitor = getInput().getProgressMonitor().isNone() ? null : getInput()
                types = new CachedTypeHierarchy(project, monitor);
                FreeVars.setHierarchy(types);

                retriever.retrieveRelationships(ResourcesPlugin.getWorkspace().getRoot(), types);
```

Search upstream from XMLRetriever.getStartContext()
for method calls
named retri

```
edu.cmu.cs.fusion.xml.XMLRetriever.XMLRetriever(..) : void
edu.cmu.cs.fusion.xml.XMLRetriever.getStartContext(..,..) : Re
edu.cmu.cs.fusion.xml.XMLRetriever.retrieveRelationships(..,..
```

Figure 9.2. Can `XMLRetriever.getStartContext()` ever be called without `XMLRetriever.retrieveRelationships()` being called first? To answer this question in REACHER, a developer first opens `XMLRetriever.getStartContext()` in Eclipse. She right clicks the method declaration and invokes an upstream search. In REACHER'S search view (upper right), she types "retri". As she types, REACHER lists matching statements below. Clicking the third result adds it to REACHER'S call graph visualization (a). Looking at the visualization, she sees that all calls to `getStartContext()` are preceded by a call to `retrieveRelationships()`. But maybe there is a conditional somewhere on the path to `retrieveRelationships()`? Double clicking the path expands it (b), showing the method `beforeAllMethods()` which was previously hidden. Hovering over the call from `beforeAllMethods()` to `retrieveRelationships()` shows a popup describing the call (this edge is missing a ? due to a bug in REACHER). Clicking it opens the file in an Eclipse editor. Reading the code, she sees that the call is guarded by a conditional.

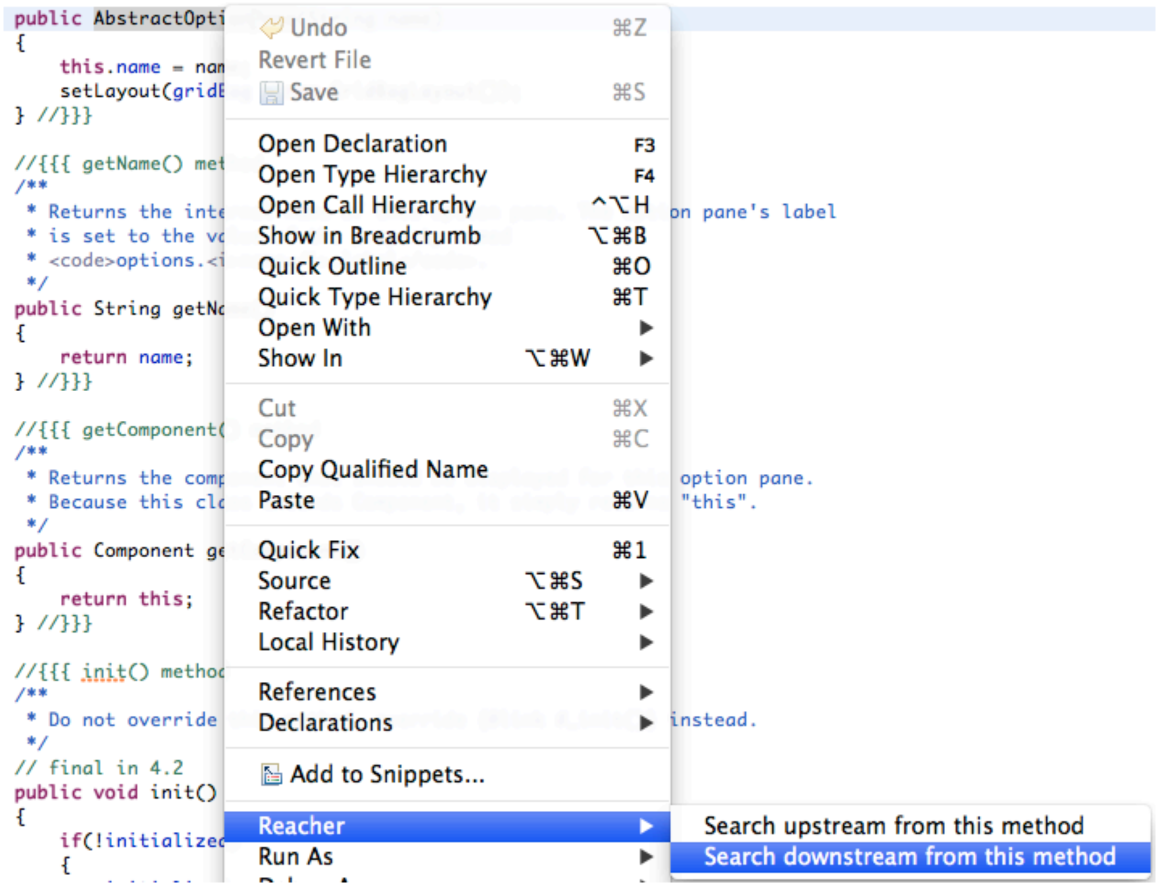
The call graph now contains 3 methods – `XMLRetriever.getStartContext()` (the origin method), `XMLRetriever.retrieveRelationships()` (the method she searched for), and `AbstractCrystalAnalysis.runAnalysis()` (Figure 9.2a). As this was an upstream search, REACHER looked for a common method calling both `retrieveRelationships()` and `getStartContext()` and found `runAnalysis()`, adding it to the call graph. Two edges emerge from `runAnalysis()` – one to `retrieveRelationships()` and a second to `getStartContext()`. The edge to `retrieveRelationships()` leaves `runAnalysis()` above the edge to `getStartContext()`, indicating it executes first. Inspecting the call graph, the developer learns that, in fact, all paths to `retrieveRelationships()` are preceded by a path to `getStartContext()`. But perhaps there is a conditional guarding the path to `getStartContext()` that might cause it not to be called? The dashed edge from `runAnalysis()` to `retrieveRelationships()` indicates that some of the path is hidden, so she double clicks to expand the path, revealing the previously hidden method `beforeAllMethods()` which connects `runAnalysis()` to `retrieveRelationships()` (Figure 9.2b). Hovering over the edge between `beforeAllMethods()` and `retrieveRelation-`

`ships()`, she sees a popup describing the call (see Section 9.4 for a discussion of why the call is missing a may execute icon). Clicking the edge navigates the Eclipse editor to the callsite. She then sees the cause of the bug – eight lines above the callsite is a conditional guarding the call. While correct for the rest of the body, it should not guard this call. Moving the call to `getStartContext()` outside the conditional block fixes the bug.

9.2. User interface design

9.2.1. Searching along control flow

Developers begin interacting with REACHER by starting a new search (see Figure 9.3). Searches can be invoked from anywhere in a method's declaration (i.e., from either its signature or statements in its body). All searches are performed relative to the beginning of the method (before the first statement in the method's body).



```
public AbstractOpti
{
    this.name = nam
    setLayout(gridB
} //}}}

//{{{ getName() met
/**
 * Returns the inte                                    on pane's label
 * is set to the va
 * <code>options.<i
 */
public String getNa
{
    return name;
} //}}}

//{{{ getComponent(
/**
 * Returns the comp                                    option pane.
 * Because this cla                                    "this".
 */
public Component ge
{
    return this;
} //}}}

//{{{ init() method
/**
 * Do not override                                     instead.
 */
// final in 4.2
public void init()
{
    if(!initialized
    {
```

Figure 9.3. Developers begin interacting with REACHER by right clicking in the editor window and invoking a new search.

REACHER supports both upstream and downstream searches along control flow paths. An upstream search begins at a destination method and traces along paths by which it may be reached. Downstream searches begin at an origin method and trace paths through its callees (and methods they call). Downstream and upstream searches are not symmetric (Figure 9.4). A downstream search captures what a target method does – all of the causality relationships resulting from a control flow path from the target method. Upstream searches correspond to what happens before the target method and include not only methods which directly or indirectly call the target method, but also other methods that a common ancestor calls. Including methods that executed before allows developers to ask about what should already have happened, such as the question about a missing call to an initialization method causing a null pointer exception (see Figure 9.2).

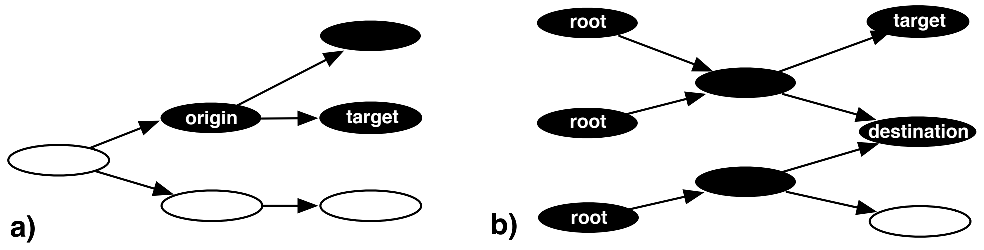
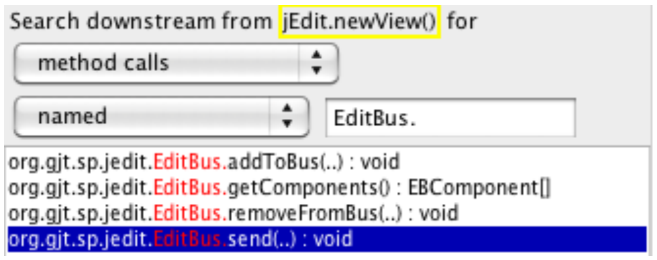


Figure 9.4. (a) A downstream search from an origin method finds methods (shaded ovals) on paths from origin, but does not find methods on paths returning from origin (unshaded ovals). (b) An upstream search from a destination method finds methods on paths terminating at destination and beginning at any root, including methods on paths from roots that occur before destination (shaded ovals).


Search downstream from jEdit.newView() for
method calls
named EditBus.

org.gjt.sp.jedit.EditBus.addToBus(..) : void
org.gjt.sp.jedit.EditBus.getComponents() : EBComponent[]
org.gjt.sp.jedit.EditBus.removeFromBus(..) : void
org.gjt.sp.jedit.EditBus.send(..) : void

Figure 9.5. As users search, REACHER displays a list of matching statements. Double clicking pins a search result, assigning the corresponding search a unique color and persistently adding it to the visualization.

After the user invokes a search and the FFPA generates paths, REACHER provides a window for searching along paths (Figure 9.5). REACHER indicates if the search is downstream or upstream and the method started from. The Reachability Observations study found several examples of searches scoped to a specific type of method or statement (Table 7.3). REACHER directly supports these searches by allowing users to select the type of method or statement to search for (method, library, or constructor calls; field read, writes, or accesses; or any of these). Additionally, REACHER lets developers select the portion of the name to match (package name, type name, or type and method name), supporting searches for any functionality in a type or package. As developers enter searches, search text is matched against any portion of the identifier (e.g., `str` matches the `str` in `ClassName.newString()`). The matching portion of the result is highlighted in red. These features make it easy to find a target by knowing just a fragment of a name or relevant concept while also minimizing typing. Selecting a result adds it to the call graph. Selections are ephemeral, supporting quick scrubbing to visualize each result in turn. Double clicking a result *pins* the item, persisting it in the visualization.

REACHER lists search results – methods and fields – with their fully qualified name and type. We experimented with instead showing a portion of all matching statements. For example, searching for `foo()` might display multiple callsites such as `a.foo()` and `b.foo()`. Searching for fields included every access and assignment statement. This provided more context and made it possible to select individual callsites and access statements. But this context made the text for each result much longer, making the result list wider and occupying more space. Additionally, result lists were far longer – methods called frequently could be included tens or hundreds of times rather than once. And forcing users to choose a specific callsite or field access statement was more distracting than helpful. So REACHER lists each method or field only once, no matter how many different places it was called or accessed. After selecting a result, users see the context in the call graph visualization.

9.2.2. Methods and expressions

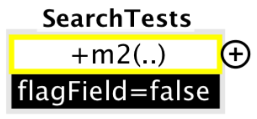


Figure 9.6. REACHER'S depiction of the method `SearchTests.m2(boolean)` and the field write `flagField = false`.

REACHER visualizes call graphs as graphs of method nodes and call edges. Following the UML's conventions [RJB99] public, protected, and private methods are prefixed by +, #, and -, respectively. The identifiers of static methods are italicized. To help distinguish overloaded methods, each parameter is indicated with a "..", and parameters are separated by commas. Including the parameter names and types would be unambiguous, but, even for common cases, names become several times longer, with a corresponding reduction in the numbers of methods shown in a fixed space. When a selected search result is a field access or a library call, REACHER displays the field access expression or call site statement below the method in which it is located (see Figure 9.6). The method the user started from is highlighted with a yellow box, corresponding to the yellow box in the search window (see Figure 9.5).

Previous research and studies in this dissertation show that developers often get disoriented when trying to explore the control flow to and from a method [AM06][KAM05](see Sections 5.1.5. and 10.3.2.). These navigations can be challenging in conventional tools. Developers are either limited to back and forward navigations or must remember the file and location of methods to navigate there. REACHER lets developers build up a working set of relevant methods shown in the visualization by pinning relevant methods. Clicking a method in the call graph opens the code in an Eclipse editor.

9.2.3. Causality

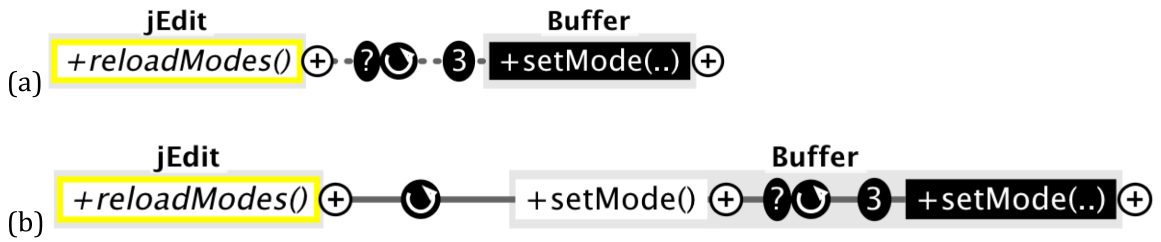


Figure 9.7. (a) Indirect calls (dashed lines) expand into (b) one or more paths of direct calls (solid lines).

Causality is a central part of reachability questions – what does this do and when does it happen? REACHER'S call graph is designed to help developers reason about causality. When a method node is created in the call graph, REACHER finds all of the control flow paths connecting it to existing nodes in the call graph, showing all of the ways it might be triggered. Knowing there *is* a causal relationship is often sufficient, so REACHER displays these control flow paths as a single indirect call edge (Figure 9.7.a). These paths are often long, complex, and uninteresting; hiding them significantly reduces irrelevant clutter. When the path is interesting, developers can double click it, expanding it to show the previously hidden methods in the path (Figure 9.7.b). Clicking a call edge navigates the editor to the corresponding call site.

While searching helps to locate distant methods, developers sometimes explore a method's immediate callers and callees. For downstream searches, REACHER depicts a circled plus icon ⊕ when a method has hidden *callees*. Clicking the icon expands all of the callees, changing the icon into a circled minus icon. Clicking the minus icon hides the callees. Similarly, for upstream searches, REACHER provides a plus icon to the left of the method indicating that there are hidden *callers*.

9.2.4. Ordering

In most of the exploration tasks we observed, developers used information about the *order* of calls. Therefore, unlike existing call graph visualizations, REACHER visually encodes the call order, sorting outgoing edges in execution order from top to bottom (see Figure 9.8). This unambiguously orders paths through the call graph. To distinguish incoming from outgoing edges, edges exit a method from the right and enter from the left. When there are multiple incoming edges, all but the first enter from the bottom to help disambiguate multiple incoming edges.

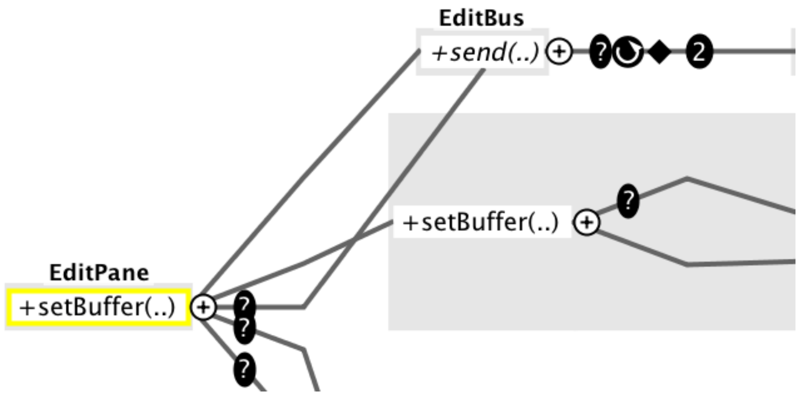


Figure 9.8. Outgoing calls execute from top to bottom.

Upstream searches cause additional complexity when a user adds a method *m* that executes before any visible methods. As REACHER's edges denote indirect or direct calls and no currently visible method calls *m*, no edges connect it, and its order is not visible. To solve this problem, REACHER computes the least upper bound method between *m* and currently visible methods. A least upper bound must exist for *m* to be upstream. The least upper bound is then added to the call graph. For example, after adding `getStartContext()` and `retrieveRelationships()`, REACHER adds the least upper bound `runAnalysis()` (see Figure 9.2), showing that `getStartContext()` executes before `retrieveRelationships()`.

REACHER uses a single node for methods along all paths by which they are reached, connecting each path after the first with backward edges. For example, in Figure 9.9, `tallCaretDelete()` and `Range.setText()` both call `remove()`, with a backward edge to `remove()` denoting `setText()`'s call. Backward edges increase visual complexity, introducing non-tree edges that overlap and cross. We considered instead creating a tree structure by replicating repeatedly called methods, except for recursive calls. However, replicating not only replicates the method itself but also its entire subtree of direct and indirect callees. Replicating subtrees greatly increases the call graph's dimensions. For example, expanding with replication the path in Figure 9.2 between `runAnalysis()` and `getStartContext()` increases the number of rows from 8 to 97 (see Figure 10.4 for a portion of a call graph built in this fashion). Furthermore, replication makes understanding subtrees more challenging by forcing developers to manually compare nodes between similar subtrees to identify differences.

However, using a single node for each method increases visual complexity, creating overlapping and crossing edges that can be challenging to untangle. To help solve this problem, REACHER lets developers mouse over an element to see its connections (see Figure 9.9). Entering a node highlights incoming and outgoing edges; entering an edge highlights incoming and outgoing nodes. One study participant commented, “It kinda reminds me of a magician, that if they want to see if there are any wires around they move their hand.”

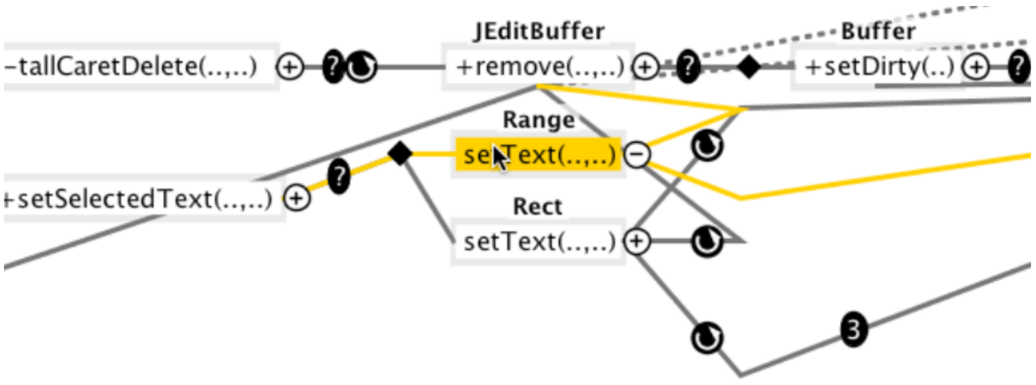


Figure 9.9. Mousing over a method highlights incoming and outgoing calls.

9.2.5. Type membership

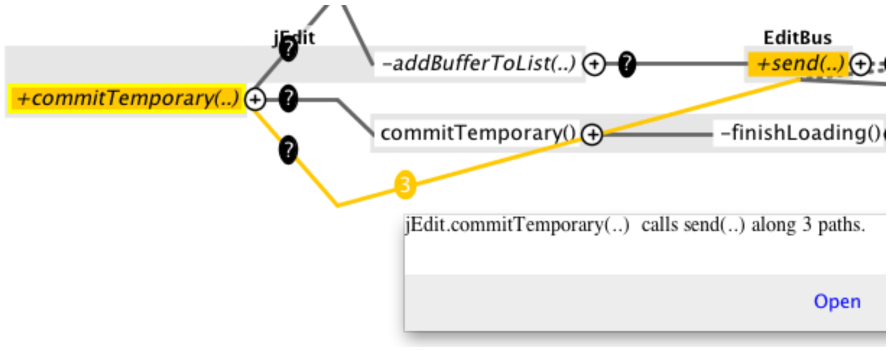
Types (e.g., classes) express a developer’s intention that the methods and fields they contain are related. REACHER visually encodes type membership with shadows grouping adjacent methods with a common type (see Figures 9.1, 9.7, and 9.10).

9.2.6. Layout

REACHER uses an automatic layout to assign each method a position. REACHER’S layout technique begins at root methods – methods with no visible callers. Call graphs produced by upstream searches may have multiple roots. From each root, REACHER computes a spanning tree. For methods with multiple incoming edges, the spanning tree includes the edge which executes first. REACHER then walks the spanning trees in-order to compute positions for each method, assigning positions from top to bottom and left to right. For methods with a single callee, both are assigned to the same row, with the caller to the left of its callee. For methods with multiple callees, each callee is given its own row from top to bottom. This process hierarchically computes a row and column assignment for each method. Row height and column width are then assigned using the maximum vertical and horizontal dimensions, respectively, of their cells. Finally, REACHER stacks each spanning tree vertically, with backward edges linking trees.

9.2.7. Repetition and choice

Realizing that a call is guarded by a conditional or may execute repeatedly can be important for answering reachability questions. REACHER alerts developers to the presence of these constructs by visualizing repetition and conditionals with call edge icons. Question marks indicate a conditional guarding a call’s execution; loop icons indicate call sites in a loop. When a call could be to one of several overriding methods because of dynamic dispatch, edges to these callees begin with a single shared line and branch into separate lines at a diamond icon. REACHER condenses repeated edges to the same method into a single edge, indicating the edge count with a number icon. But when an edge to a different method is interleaved between the repetitions, the repeated edges are shown separately before and after the interleaved edge, showing ordering. For example, in Figure 9.8, the repeated calls to `send()` are shown before and after the interleaved call to `setBuffer()`. Hovering over an icon displays a descriptive popup (see Figure 9.10).



jEdit.commitTemporary(..) calls send(..) along 3 paths.

Figure 9.10. Hovering over an icon or edge displays a descriptive popup.

9.2.8. Supporting rapid exploration

As developers work, they construct a *working set* of task relevant code [KAM05]. This code is often widely scattered across many classes in a codebase and tangled amongst other unrelated code. Developers often switch back and forth between elements in the working set, incurring a significant cost in the mechanics of identifying which tab to click and determining the position of each snippet within each file [KAM05]. Moreover, this navigational overhead encourages developers to remember and guess about the contents of their working set, as checking the code adds more time and cost. Finally, in extreme cases, developers may even become lost and disoriented. Several participants in the Exploration Lab Study gave up, becoming too overwhelmed trying to keep track and make sense of all of the relevant code (see Section 5.1.5).

REACHER helps developers stay oriented by providing a task-relevant overview of the code. REACHER provides a variety of additional interactive features for rapidly expanding details and then hiding them again if the user decides they are not relevant. REACHER depicts relevant methods and let developers navigate among them simply by clicking. “Back” and “forward” commands traverse a web-browser style navigation stack of visualization states. Pan and zoom commands lets users focus on specific areas or get an overview. Clicking and dragging the background pans the display, and using the mouse scroll wheel controls the zoom level. To help users track the location of methods as new methods are added and layout positions change, REACHER smoothly animates transitions. Showing the callers or callees of a method anchors the method’s position, moving other nodes relative to it, so that the part of the visualization at which the user was looking stays stable.

9.3. Applicability

To understand the scope and generality of REACHER'S design, the applicability of REACHER to the 19 observed reachability questions developers asked (Tables 7.1 and 7.2 in Sections 7.1 and 7.3; Section 7.4) was examined. Table 9.2 lists the 11 of the 19 (58%) observed reachability questions REACHER directly supports and the corresponding steps to apply REACHER to each of these questions. Table 9.3 lists the 8 observed reachability questions for which additional features are required and describes the additional features required. Section 11.1 describes possible designs for many of these features.

Question	**Steps to use REACHER**
What resources are being acquired to cause this deadlock?	Search downstream for each method which might acquire a resource, pinning results to keep them visible
When they have this attribute, they must use it somewhere to generate the content, so where is it?	Search downstream for the attribute, scoping search to field reads.
How are these thread pools interacting?	Search downstream for thread pool class, scoping search to matching type names.
How is data structure *struct* being mutated in this code (between *o* and *d*)?	Search downstream for *struct* class, scoping search to matching type names and searching for field writes.
How [does] application state change when *m* is called denoting startup completion?	Search downstream, leaving search text blank, and scoping search to field writes
Method *m* is fast enough that it does not matter that it is called more frequently.	Search downstream for library calls. Inspect calls to determine which are potentially slow.
Why is calling *m* necessary?	Search downstream for library calls. Inspect calls, revealing one that refreshes the screen.
Method *m* need not invoke method *n* as it is only called in a situation in which *n* is already called.	Search upstream from *m* for *n*.
The scroll handler *a* does not need to notify *b*, because *b* is unrelated to scrolling.	Search downstream from *b* for “scroll”.
Is the initialization method always called before the use site?	Search upstream for the initialization method. Inspect path for ordering and conditionals.
How is the collection instance that is getting passed around being mutated?	Search downstream for writes to the collection class.

Table 9.2. Steps to use REACHER for the observed reachability questions supported by REACHER.

Question	**Additional features required**
Where is method *m* generating an error?	Support searching for values of variables (e.g., error message string constant).
What [is] the test doing which is different from what my app is doing?	Support comparing traces of similar code snippets.
Where [is] the code assuming that the tables are already there?	Support comparing traces before and after a change (e.g., after table initialization code commented out).
Is [there] another reason why status could be non-zero?	Support searching along data flow to see what values may reach status and where these originate.
From what callers can the guards protecting statement *d* in method *m* be true?	Support searching relative to a statement inside a method.
Removing this call in *m* does not influence behavior downstream.	Support comparing traces before and after a change.
What situations currently trigger this screen update in *m?*	Support searching upstream for cut points (methods that are never called) and scoping upstream to only paths by which a method is reached rather than everything that execute before.
Do all paths to constructor calls to subtypes of a class go through a factory class?	Support searching from a set of methods to allow developers to search upstream from all constructors in subtypes of a class.

Table 9.3. Additional features required for REACHER to support the remaining observed reachability questions.

9.4. Implementation

REACHER is implemented as an Eclipse plugin for Java. REACHER'S visualization is implemented using the Prefuse visualization toolkit [HCL05]. REACHER uses FFPA to generate static traces (see Chapter 8), executing an FFPA query whenever the user invokes a search. When REACHER is started, it attempts to load saved method summaries from disk. If no summaries are found, FFPA runs its first two phases to generate method summaries.

After FFPA generates static traces, REACHER next constructs a visual graph. A visual graph differs from static traces in several important respects. A visual graph contains only the methods in which the developer has expressed an interest, including origin and destinations of searches, selected search results, expanded callers and callees, and the least upper bounds of visible methods (see Section 9.2.4). These methods are *displayed* methods. All other methods in the static traces are hidden, replaced with indirect (dashed) calls denoting hidden methods when they occur on paths between displayed methods. Second, each method occurs at most once in the visual graph, across all contexts. When there are multiple contexts, they are merged.

Upstream searches generate more than one static trace. Each root that can reach the destination is a separate static trace. But static traces overlap when a portion is reached in the same context. Thus, while each static trace can be viewed as a separate trace – an execution trace beginning at a root – the union of all static traces generated by a search can also be viewed as a graph. This graph is the *active trace graph*. For a downstream search, the active trace graph is simply the single static trace produced from executing FFPA from the origin.

To construct a visual graph for an upstream or downstream search, REACHER first uses a breadth-first traversal of the active trace graph to mark displayed methods and all paths by which they may be reached. Marks describe the portion of the trace graph that will be rendered into the visual graph. Any nodes that have not been marked will not be rendered and can be skipped during the rendering pass. This speeds the rendering pass by reducing the number of traces to be visited. Note that marks themselves have no representation in the visualization but are only to determine which traces need to be visited when rendering.

The marking traversal begins at the root of each static trace. Each node in the active trace graph is visited exactly once (even if it occurs in many static traces). When a displayed method is encountered, the node itself and all methods on the path by which it was reached are marked. When a marked method is encountered, the path by which it was reached is marked. Paths from the marked node are not traversed. Figure 9.11 (1) shows an example of a marked trace graph.

REACHER next uses the marked trace graph to render a visual graph. REACHER traverses the marked nodes of the active trace graph, beginning at each root. In contrast to the marking traversal, this traversal visits each trace node once along every path by which it may be reached. For example, in Figure 9.11, `a()` and `b(true)` are each visited twice (once from `r1()` and once from `r2()`) while `c()` is visited 4 times. However, as `b(false)` was not marked, it is not visited. When a displayed node is encountered, a corresponding visual graph node is created. As additional displayed nodes are encountered, both a node and an edge (either direct or indirect as appropriate) are created. But when the same method appears twice in a row from the same caller, the methods are combined into a single node. Finally, REACHER performs a consolidation pass to combine identical visual trees (Figure 9.11 (2) and (3)). Each pair of visual trees is compared and identical pairs combined.

Finally, REACHER computes information for the edge icons. To determine the number of times a call happens, REACHER counts the number of call traces that correspond to the call in the visual graph. To determine if a call edge occurs in a loop, REACHER checks if any of the corresponding traces have call sites that are located inside a loop. To determine if a call edge might not execute, REACHER checks to see if the call does not execute on some of the paths FFPA followed through the method summary. However, in some situations, this implementation will not identify a call as might not executing.

When a call is guarded by a conditional and this conditional is not modeled by FFPA, REACHER does not show a “may execute” icon, when in fact it should. For example, in Figure 9.2, there is no "may execute" icon on the call to `retrieveRelationships()`. This is because the guard contains the expression `project == null`, which FFPA does not model. FFPA does not model expressions that could never be controlled by constant controlled conditionals such as those involving Boolean operators such as arithmetic. FFPA could be extended to record the presence of these conditionals in the summaries, to allow REACHER to accurately report when a call could or could not execute. But this extension has not been implemented in the current design.

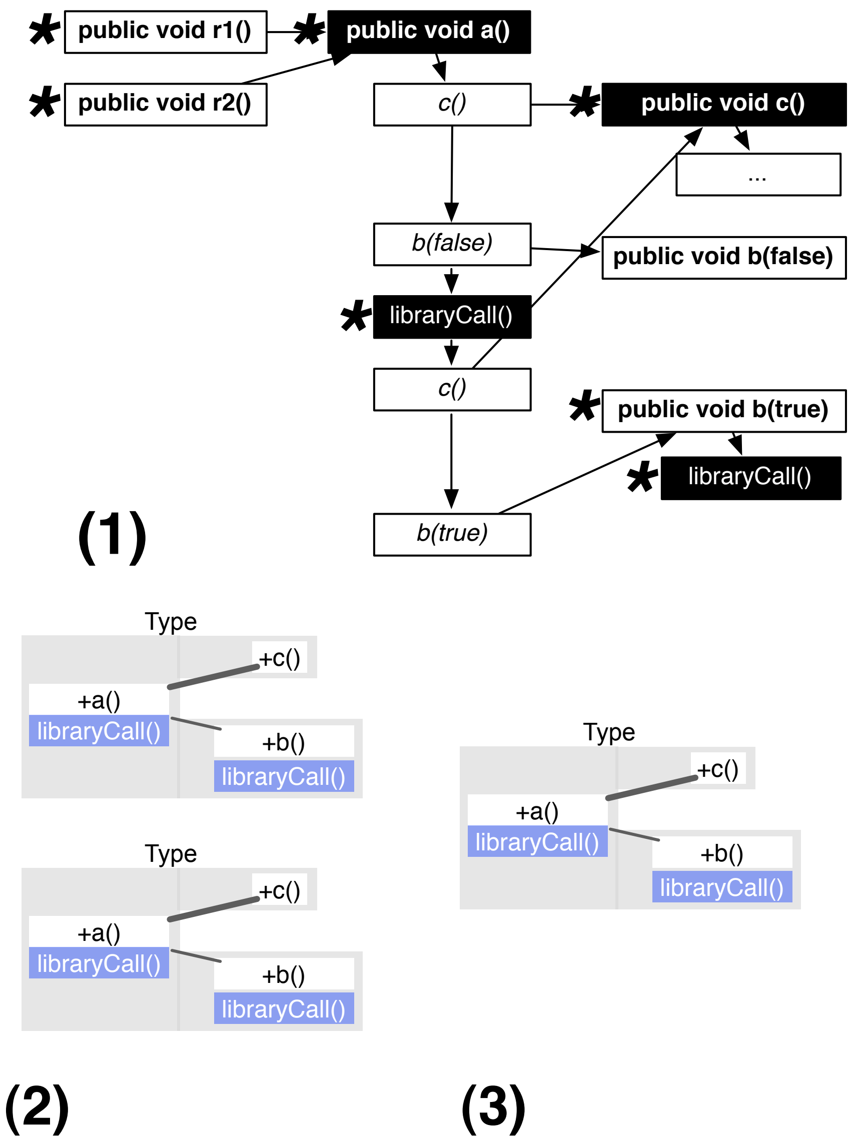


Figure 9.11. REACHER renders visual graphs by (1) marking (*s) trace nodes on paths to displayed nodes (black fill), (2) rendering visual graph trees, and (3) combining identical trees.

9.5. Conclusions

REACHER is designed to help developers understand and explore code more effectively by helping them answer reachability questions. Rather than manually traverse across long and complex paths, developers can simply search. To more effectively reason about causality, ordering, and type membership, developers can simply inspect a task-specific diagram. And to stay oriented, developers can use the diagram as a navigation aid, using it to rapidly navigate the code editor through their working set. The next chapter describes several evaluations of REACHER.

10.

STUDIES OF REACHER

To improve REACHER's initial design and evaluate the final design, several studies of REACHER were conducted. Early in its design and implementation, a small pilot paper prototype study was conducted. This study was designed to test the main premise of REACHER's design – that a combination of search and call graph visualization would help to answer reachability questions – and identify potential usability problems early. Following this study, a number of aspects of REACHER's design were improved and implemented in an initial prototype. A lab study evaluation was then conducted, which both confirmed REACHER's potential and identified a number of areas for improvement. After further iterating the design, I conducted a final evaluation, which found that REACHER can help developers answer reachability questions significantly faster and more successfully.

10.1. Paper prototype pilot study

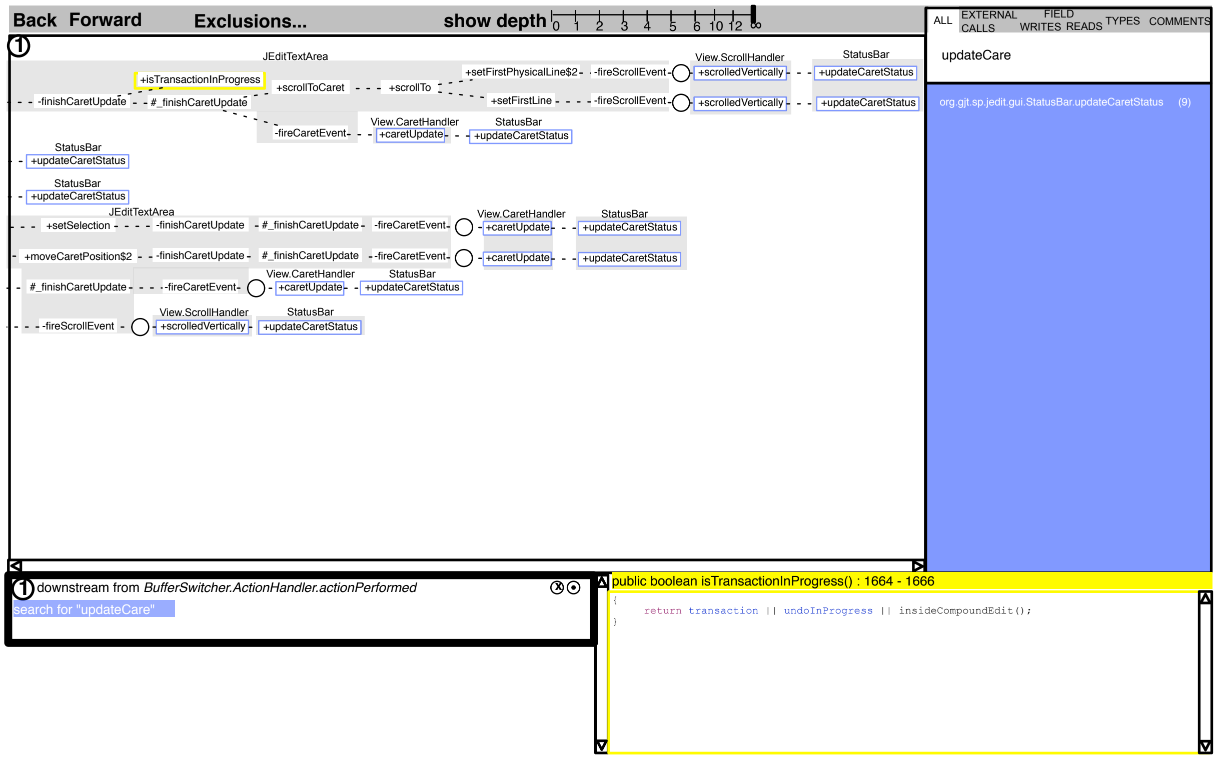
I conducted a small pilot paper prototype study of REACHER in which a single participant performed a task with a mockup of REACHER. Several of the problems identified in this study influenced REACHER's subsequent design iterations.

10.1.1 Method

A task was selected from a previous study of code exploration (see Section 5.1.3). The **FOLDS** task involved investigating complex code in a large codebase in order to propose a design fix. A central part of the task was understanding how the existing design worked, which involved understanding control flow relationships in the code. Based on the questions participants in the original lab study had asked, mockup screenshots of REACHER were constructed in a drawing program (see Figure 10.1) depicting how REACHER might depict answers to these questions. The mockups included both REACHER's call graph visualization and the interface elements for interacting with REACHER. Using the other task from the same previous study (**CARETS**), a tutorial was also designed, walking through the use of REACHER to answer questions relevant to this task, illustrated with mockups. See Appendix 5 for the complete materials.

A participant was recruited from graduate students at Carnegie Mellon who had experience with both Java and Eclipse. After working through the tutorial, the participant was given 1 hour to work on the **FOLDS** task. The participant was encouraged to think aloud as she worked, and her work was recorded with the Camtasia screen capture software[11] and an audio recorder.

The mockups were presented as pages in an Omnigraffle[12] document. When the user wished to invoke a command, a (physical) paper overlay was used to indicate the available commands (commands were recorded by the audio recorder). After executing a command, the appropriate REACHER mockup was opened in Omnigraffle by the experimenter. In all, 10 mockups were created for the tutorial and 12 for the **FOLDS** task (see Appendix 5).



```
public boolean isTransactionInProgress() : 1664 - 1666
{
    return transaction || undoInProgress || insideCompoundEdit();
}
```

Figure 10.1. A mockup of REACHER used in the paper prototype study.

10.1.2 Results

The participant was able to successfully use the REACHER mockup to help explore code, but was unable to finish the task. The challenges that the participant faced highlighted several limitations of this first design of REACHER.

One of the biggest barriers to making progress with REACHER was asking ordering questions. In particular, the participant wanted to know about the ordering relationship between two methods. But this version of REACHER could not answer this question directly, as this version

[11] http://www.techsmith.com/camtasia.html

[12] http://www.omnigroup.com/products/omnigraffle/

could only find methods on a path to the origin or downstream, not all methods before or after. This did not match what the participant wished to ask.

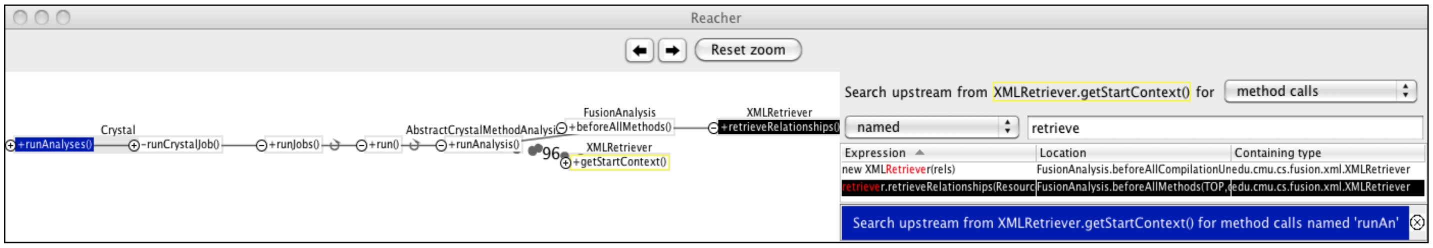
Dealing with origins and directions was also confusing. Switching between downstream and upstream questions required the participant to explicitly set an origin and a direction. Understanding when the direction had to be switched, what it meant, and what the current state was proved too confusing. Moreover, there was a separate search cursor that controlled which portion of the visualized call graph would be searched.

The participant also disliked the horizontal orientation of the call graph. She felt that developers were far more familiar with call stacks in the debugger and would thus more naturally understand that methods with call relationships were arranged from top to bottom. But while she found the orientation unexpected, she still was able to understand it.

Finally, the participant felt that REACHER should make it easier to understand when something definitely happens. While this version of REACHER distinguished control flow paths that may execute from those that must execute, the participant wished to have more direct support for investigating the circumstances under which a path would execute.

10.1.3 Subsequent improvements to REACHER

Following the paper prototype study, several aspects of REACHER's design were changed to address the observed problems. Figure 10.2 illustrates the new design (this design still differed substantially from REACHER'S final design described in Chapter 9 – see Section 10.2.4). Most fundamentally, the semantics of the paths REACHER searches and depicts were changed. Upstream searches were changed to include all methods executing before (for any origin). This lets developers search for ordering relationships without having to switch to a downstream search. However, the user still had to search for a common root between the search origin and search target in order to see ordering relationships between them.



Search upstream from XMLRetriever.getStartContext() for method calls

named retrieve

Expression	Location	Containing type
new XMLRetriever(rels)	FusionAnalysis.beforeAllCompilationUn	edu.cmu.cs.fusion.xml.XMLRetriever
retriever.retrieveRelationships(Resourc	FusionAnalysis.beforeAllMethods(TOP,d	edu.cmu.cs.fusion.xml.XMLRetriever

Search upstream from XMLRetriever.getStartContext() for method calls named 'runAn'

Figure 10.2. Redesigned REACHER interface. The user has searched upstream from `getStartContext` for "retrieve," which finds the method `retrieveRelationships` which executes before `getStartContext`, even though it is not on the path from the root `runAnalysis`.

To simplify and reduce confusion with downstream and upstream searches, several other changes were made. The command "Set as origin" was removed and replaced with "Search downstream" and "Search upstream," replacing the need to flip between upstream and downstream modes. Support for restricting searches to a portion of the visible call graph was eliminated. The origin of the visible call graph was made more prominent with a highlight in the call graph visualization. And the search pane added a description of the currently active search (e.g., Search upstream from `XMLRetriever.getStartContext()` in Figure 10.2).

10.2. Lab study 1

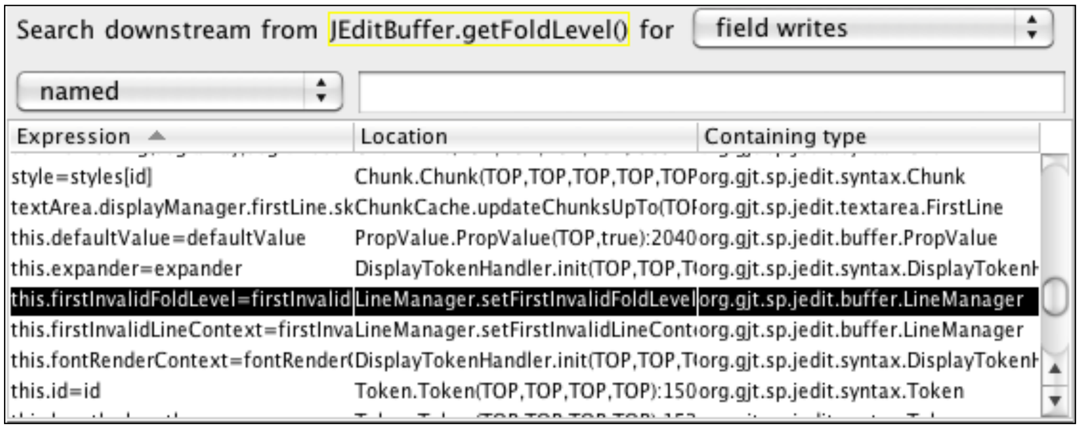
After implementing REACHER based on this design, a lab study was performed to evaluate REACHER's ability to help developers understand and explore code more effectively. Participants worked on two challenging tasks: one based on the **FOLDs** task (see Section 5.1.3) and one based on a debugging task I had observed (described in Section 7.5).

10.2.1. Method

12 participants were recruited from students and staff at Carnegie Mellon University. All reported being comfortable programming in Java and using Eclipse (mean = 6.1 years experience), had professional software development experience (mean = 2 years), and knew an average of 6 programming languages. None had previously seen REACHER. Participants were randomly assigned to either a control condition in which they used standard Eclipse or an experimental condition in which they used Eclipse and REACHER.

We designed two tasks (**MUTATION** and **PROTOCOL**) that both involved answering a reachability question and were very challenging. Participants worked in two unfamiliar and complex codebases. The **MUTATION** task was in jEdit, an open source text editor (see also Section 5.1.3). The **PROTOCOL** task was in Fusion, 50 KLOC static analysis tool for Eclipse and involved Eclipse plugin debugging, with which most participants were not familiar. Developers were not given any domain knowledge about either application, but only a short description of an issue and were directed to several relevant locations in the code.

We adapted the **MUTATION** task from an earlier lab study (see Section 5.1.3) in which developers had to investigate and fix a design problem in which a getter (`getFoldLevel()`) was being called despite its return value being ignored. One hypothesis several of the participants had in the previous study was that the method was being called in order to mutate state by assigning fields. In the current study, developers were asked in the **MUTATION** task to find all of the statements downstream from the method that assigned a field. This task was challenging as there were 67 such field writes connected by 599 method calls. Figure 10.3 lists a small portion of these field writes.


Search downstream from JEditBuffer.getFoldLevel() for field writes

named

Expression	Location	Containing type
style=styles[id]	Chunk.Chunk(TOP,TOP,TOP,TOP,TOP	org.gjt.sp.jedit.syntax.Chunk
textArea.displayManager.firstLine.sk	ChunkCache.updateChunksUpTo(TO	org.gjt.sp.jedit.textarea.FirstLine
this.defaultValue=defaultValue	PropValue.PropValue(TOP,true):2040	org.gjt.sp.jedit.buffer.PropValue
this.expander=expander	DisplayTokenHandler.init(TOP,TOP,T	org.gjt.sp.jedit.syntax.DisplayTokenH
this.firstInvalidFoldLevel=firstInvalid	LineManager.setFirstInvalidFoldLevel	org.gjt.sp.jedit.buffer.LineManager
this.firstInvalidLineContext=firstInva	LineManager.setFirstInvalidLineCont	org.gjt.sp.jedit.buffer.LineManager
this.fontRenderContext=fontRender(	DisplayTokenHandler.init(TOP,TOP,T	org.gjt.sp.jedit.syntax.DisplayTokenH
this.id=id	Token.Token(TOP,TOP,TOP,TOP):150	org.gjt.sp.jedit.syntax.Token

Figure 10.3 Some of the many field writes in the MUTATION task.

We adapted the **PROTOCOL** task from an actual debugging task we observed in the field (see Section 7.4.1.). Participants were given the actual code at the time of the bug, given steps to reproduce the bug, and instructed in how to use the debugger to debug an Eclipse instance. To simulate a small portion of the knowledge the original developer had, participants were instructed that the exception was caused because `retrieveRelationships()` was not being called before `getStartContext()` (see Figure 10.2), leading to missing state and a null pointer exception. Participants were asked to determine the conditions under which this might occur.

To ensure all participants were familiar with Eclipse's many code navigation features, all participants were first given a tutorial on these features (also used in Chapters 5 and 7.2; see Appendix 6 for the complete materials). Participants in the REACHER condition then completed a short tutorial describing the visualization notation and interactive features of REACHER. Participants were next given each of the task descriptions in turn and then had 30 minutes to work on each task. Several participants gave up (see Table 10.1); other participants ran out of time and were stopped. Participants were not told if their answer was correct until after they had finished working on both tasks. Participants used Eclipse 3.5 and were allowed to use any feature they wished. Participants worked on a 2.8 Ghz computer with 8 GB of memory, a large 30" monitor, and an additional laptop screen. To understand why developers used the approaches they did, participants were asked to think aloud as they worked. We then recorded audio and the screen using Camtasia.

10.2.2. Results

In the **MUTATION** task, participants using REACHER finished the task in less than half the time (a significant difference, $p < .05$) and were significantly more successful ($p < .05$). None of the participants using only Eclipse succeeded while half of the participants using REACHER were successful. In the **PROTOCOL** task, one (17%) of the participants using only Eclipse succeeded while two (33%) of the participants using REACHER succeeded. This difference was not significant ($p = .27$). Participants using Eclipse only were slightly faster than those using REACHER (23.2 vs. 25.3 minutes), but this difference was not significant (p = .32). Table 10.1 summarizes the task performance results.

		ECLIPSE	REACHER
MUTATION	succeeded	0%	50%
	gave up	17%	0%
	avg time ±std dev (mins)	N/A	7.0 ± 5.0
PROTOCOL	succeeded	17%	33%
	gave up	33%	0%
	avg time ±std dev (mins)	9.6	20.5 ± 6.5

Table 10.1. REACHER Lab Study 1 task performance.

Participants in the Eclipse-only condition experienced a number of problems that often prevented them from succeeding. In the **MUTATION** task, participants used the call hierarchy to traverse downstream calls. Most participants browsed the immediate callees or one level deeper, while a few browsed "way down somewhere" and reported "being half lost." Navigating calls to listeners was challenging for many participants, as it required leaving the call hierarchy, finding types that implement the listener interface, deciding which might actually be called at the callsite, finding the relevant method, and then doing a new search on its callees. Thus, only one of the participants spent the time to navigate through these calls. However, most of the field writes were happening past these calls. In contrast, REACHER treats such situations like other conditionals, either determining that a call to a particular runtime type is infeasible or including all possible alternatives.

In the **PROTOCOL** task, all but one of the Eclipse-only participants failed. The successful participant enjoyed a unique advantage amongst participants in either condition: he correctly guessed the diagnosis for the problem while reading the task description. Using the debugger, he was then able to set a breakpoint on the call to `retrieveRelationships()`, browse up through its callers, and rapidly identify the offending conditional. The remaining, unsuccessful participants instead tried to understand the paths connecting `getStartContext()` to `retrieveRelationships()`. Developers used the call hierarchy to traverse paths upwards from both, but were unsuccessful in locating their intersection. None of the participants ever learned that `retrieveRelationships()` is called along a single path before the many paths to `getStartContext()` (see Figure 10.4 for a small selection of the paths). Moreover, even though several found the conditional in `beforeAllMethods()` that caused the bug (see Figure 9.1), most participants had no idea of its significance since they lacked a hypothesis about why it was relevant. This replicates a previous finding that developers viewing highly task-relevant portions of code without the proper background knowledge may not benefit from it [RMC04]. Several participants became disoriented in the many paths by which `getStartContext()` is reached, and a few attempted to deal with this problem by writing down calls on paper. One-third of the participants thought the task would be too difficult to complete and gave up.

The biggest barrier participants using REACHER experienced was realizing that searching with REACHER would help. The successful participants generally came to this realization quickly, after spending a minimal amount of reading the code connected to the starting point. In the **MUTATION** task, these developers realized that searching downstream from `getFoldLevel()` for field writes would finish the task. The unsuccessful participants in the REACHER condition either ignored REACHER entirely, used REACHER to traverse calls, which they manually inspected, or searched downstream only for setter methods they already expected wrote a field. Another participant reported that she used the call hierarchy initially, rather than REACHER, because she had been using the call hierarchy for some time and that was what she was familiar with doing.

In the second task (**PROTOCOL**), more of the participants used REACHER, both because of the task's difficulty and perhaps because they were slightly more comfortable with REACHER. Many of the participants were quickly able to search and determine that `retrieveRe-lationships()` is called along a single path that is always before `getStartContext()` (see Figure 10.3), something that none of the participants without REACHER learned. But hypothesizing that the bug was caused by a conditional guarding the call to `retrieveRelation-ships()` was challenging, and none of the unsuccessful REACHER participants did this.

Despite these difficulties, our study demonstrated that searching along control flow paths can make developers dramatically more effective. In the **MUTATION** task, a third of the REACHER participants successfully completed the task in 4.5 minutes or less; none of the participants without REACHER were successful. Two of the three participants who successfully completed the **PROTOCOL** task used REACHER, and unlike any of the other participants, found the connection between `retrieveRelationships()` and `getStartContext()` before hypothesizing that a conditional was relevant. Finally, the successful developers using REACHER expressed excitement about REACHER or were interested in when REACHER might be available for them to use in their everyday work.

10.2.3. Discussion

The user study revealed that getting developers to express their questions as reachability questions is more challenging than we expected. Some of the developers immediately "got" REACHER and quickly adapted their strategies. But others, despite a tutorial explaining how to use REACHER, chose to use Eclipse's call hierarchy because it was more familiar. Others, despite having done searches in a tutorial, persisted in manually traversing across calls for tens of minutes for well-defined search targets specified in a task. Due to not using REACHER in these situations, these developers were far less successful than those that used it.

There are several possible reasons for this. In some cases, developers may just be unaware of the problems and limitations of their existing strategies. For example, in the **MUTATION** task, developers may simply think that inspecting a few callees is sufficient to find all of the mutations happening, not realizing that several events trigger far more mutations to occur. In other cases, developers fit REACHER into their existing workflow without really adapting their workflow to take advantage of the novel aspects of REACHER. For example, one participant simply used REACHER to search through a method they already thought mutated the fields, not ever thinking to use REACHER on methods they did not suspect would write fields. One of the primary benefits of REACHER, compared to existing exploration approaches, is that it can replace guesswork with certainty. Rather than having to rely on guesses about what control flow relationships probably look like, REACHER can quickly find the actual paths. But, when developers are overconfident about their existing knowledge of control flow, they may not realize that using REACHER to test their beliefs has benefits.

Finally, developers may simply be attached to their existing tools. Despite the many additional features modern integrated development environments provide, many developers still use simple text editors (see Chapter 4). It may remain challenging to convince such developers to use tools such as REACHER. More research is necessary to better understand how developers decide to adopt tools.

10.2.4 Subsequent improvements to REACHER

The study also revealed several problems with REACHER that were subsequently addressed. Having REACHER display ordering relationships between methods still required first manually searching for a common route before next searching for the target method. To address this issue, REACHER was augmented to automatically find root methods. Whenever there are disconnected methods in the visualization, REACHER looks for a least upper bound on paths connecting these methods. If one exists, it is shown in the visualization.

Another problem participants faced was navigating and making sense of call graphs. In some cases, call graphs contain subgraphs that are reached by many different paths. This version of REACHER visualized these call graphs by replicating each subgraph for each path by which it was reached. This greatly increased the size of the call graph, making it unwieldy to navigate and make sense of (Figure 10.4). To address this problem, REACHER was redesigned to only show one copy of each method and introduced a variety of new affordances for expressing ordering (see Section 9.2.4). This also converted the visualization from a tree to a directed graph. Interactive features were added to make making sense of these edges easier (see Section 9.2.4).

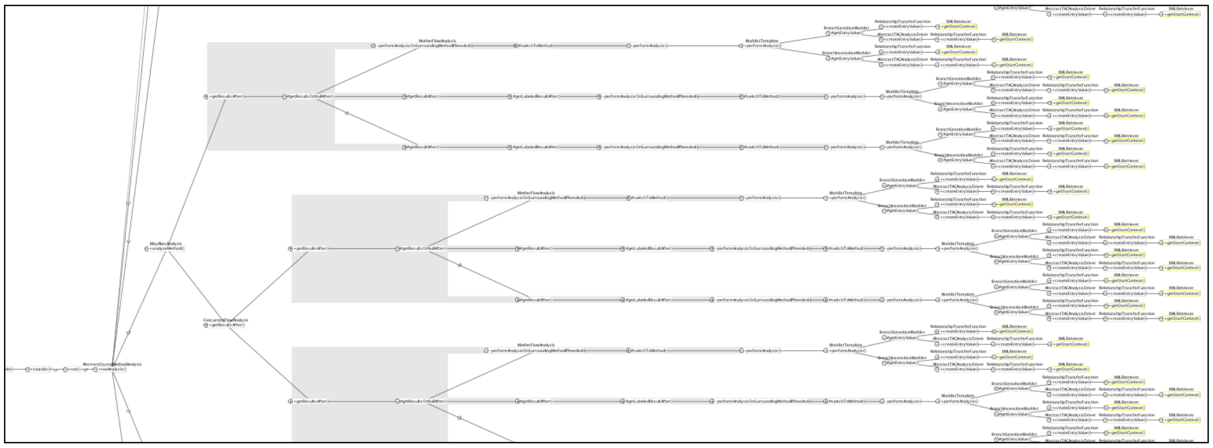


Figure 10.4. This version of REACHER displayed copies of methods for each path by which they were reached. When searching for portions of the call graph which are repeated many times, this created a call graph visualization containing much redundancy. The current version avoids this problem by displaying only a single copy of each method (see Section 9.2.4).

10.3 Lab study 2 [13]

Lab study 1 revealed the potential of REACHER: participants were substantially faster and more successful in the **MUTATION** task. But it also revealed several problems with REACHER, both in its design and in communicating its potential use. Therefore, REACHER was improved and then a second lab study was conducted to further evaluate REACHER's potential for helping developers more effectively understand and explore code.

10.3.1. Method

12 new participants were recruited from students and staff at Carnegie Mellon University. All participants reported being comfortable programming in Java (median = 4.5 years experience), had professional software development experience (median = 1.1 years), and knew an average of 4 programming languages. None had previously used REACHER.

Participants performed 6 tasks and were given 15 minutes to complete each task. Each task posed a reachability question and involved finding and understanding control flow between events. Table 10.2 lists each of the tasks' actual questions (only an excerpt of tasks 5 and 6 are listed—see Appendix 7 for the complete materials). To test if participants were able to understand the visualization notation, each task was designed to require understanding a particular aspect of the notation. Tasks 1 and 2 dealt with ordering, tasks 3 and 4 dealt with conditions, and tasks 5 and 6 dealt with repetition. All participants performed all 6 tasks and did half of the tasks with Eclipse alone and half with Eclipse and REACHER. Participants

[13] This section describes work previously published in [LM11].

were randomly assigned to conditions. The order of the tasks, whether they received the 3 Eclipse only tasks or the 3 REACHER tasks first, and which tasks were used in each condition were all counterbalanced.

All tasks were performed in the jEdit codebase, a 55 KLOC open source text editor used in several previous studies of code exploration (including the Code Exploration Study described in Chapters 5 and 7). Several of the tasks dealt with jEdit's EditBus which provides a publish / subscribe mechanism for sending and receiving messages. Understanding connections through the EditBus was a key challenge participants had faced in the Code Exploration Study tasks. In this study, participants were asked questions such as what events were sent on the bus or to trace messages through the bus.

To ensure all participants were familiar with Eclipse's many code navigation features, all participants were first given a tutorial on Eclipse (the same as used in previous studies). Before performing tasks with REACHER, participants completed a second tutorial that explained REACHER'S notation and interactions that could be used to answer reachability questions. Participants were given task instructions on paper and allowed to take notes. Participants used Eclipse 3.6.1 and were allowed to use any feature they wished. Participants worked on a 2.8 Ghz computer with 8 GB of memory, a large 30" monitor, and an additional laptop screen. To understand why developers used the approaches they did, participants were asked to think aloud, and we recorded audio and the screen with Camtasia.

Task 1. When a new view is created in `jEdit.newView(View)`, what messages, in what order, may be sent on the `EditBus` (`EditBus.send()`)?
Task 2. When text is deleted (`JEditTextArea.delete()`), what is the first message that may be sent on the `EditBus` (`EditBus.send()`)?
Task 3. Does setting the buffer in `EditPane.setBuffer()` cause the caret status on the status bar to be updated at least once (`StatusBar.updateCaretStatus()`)?
Task 4. Other than the check that the `firstLine` has changed from the `oldFirstLine` in `setFirstLine()`, are there other conditionals that might cause `JEditTextArea.setFirstLine()` not to update the scroll bar (`JEditTextArea.updateScrollBar()`)?
Task 5. How many messages may `jEdit.commitTemporary()` send to the `EditBus`? (i.e., how many times might it invoke `EditBus.send()`?) …
Task 6. How many messages may `jEdit.reloadModes()` send to the `EditBus`? (i.e., how many times might it invoke `EditBus.send()`?) …

Table 10.2. Participants were asked to answer a series of six reachability questions.

10.3.2. Results

Participants completed tasks 5.6 times more successfully with REACHER (78%) than with Eclipse alone (14%). Averaged across all tasks, participants' mean task time was 11.1 minutes with Eclipse alone and 7.2 minutes with REACHER. This is a conservative estimate of the time difference, because we used a time of 15 minutes (the maximum) for tasks on which participants ran out of time, whereas they would likely have taken much longer. Figure 10.5 shows success and task time per task. Participants were significantly faster with REACHER in tasks 1, 2, 4, and 6 ($p < .05$), but not tasks 3 ($p = .6$) or 5 ($p = .25$). Participants succeeded too infrequently with only Eclipse to compare times between just those who succeeded.

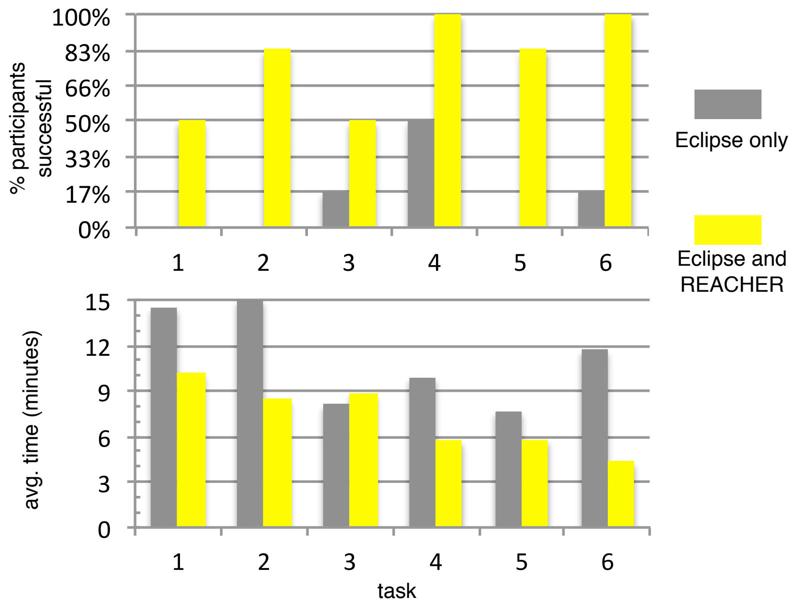
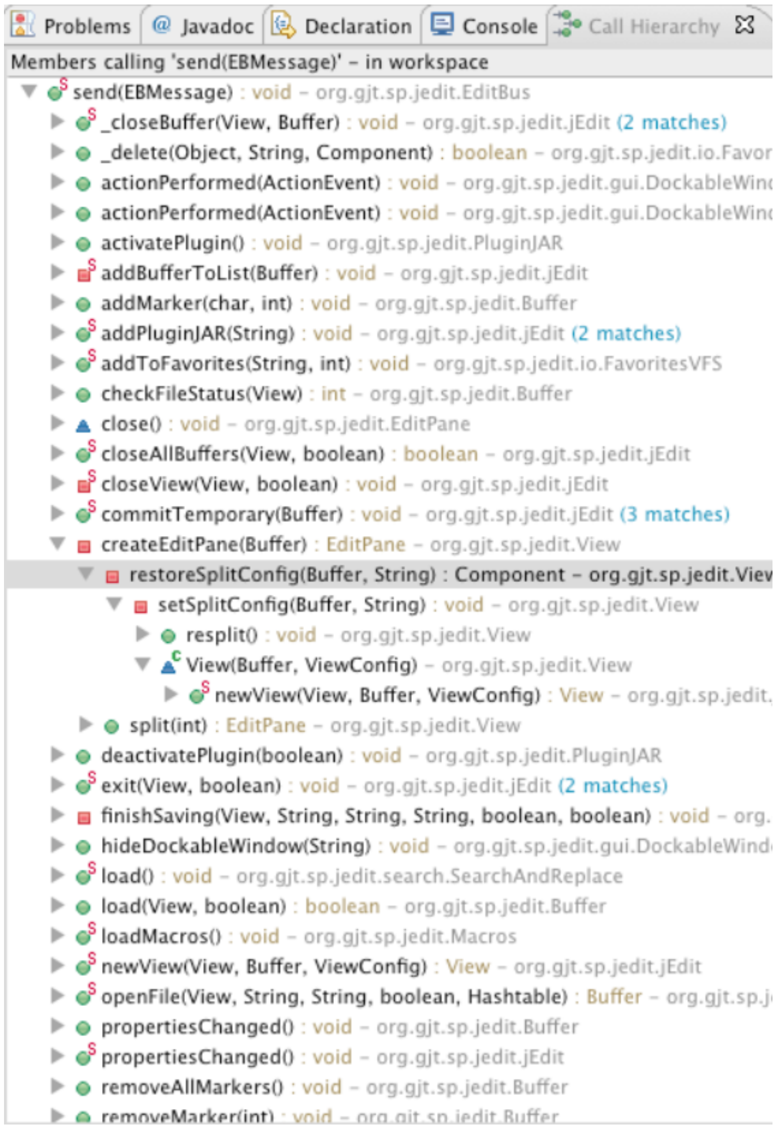


Figure 10.5. Success and average task time. Task time includes participants that failed. Participants who ran out of time received 15 minutes.

Participants with only Eclipse used a number of static exploration strategies. When reading a method, participants relied heavily on the "scent" [LBB10] of method names at call sites to decide which methods to open and read. For example, to find paths to EditBus messages, participants reasoned about which methods might be likely to do something requiring an EditBus message to be sent. Some participants tried to methodically traverse many paths, while others guessed which would be most likely to lead to the target. Many participants explicitly debated whether it was better to guess or methodically explore. Most participants also navigated to the target statement to get a sense for what it did and when it might be likely to happen.

Most participants with only Eclipse used the call hierarchy to traverse paths of calls. But, due to the huge fanout of methods, most realized the hopelessness of finding their target method in this view (see Figure 10.6). Several participants did bidirectional search, navigating call hierarchy paths both forwards and backwards and trying to pick methods to traverse based on similarity to calls from the other direction. A significant barrier to static traversal were event listeners, implemented using the Observer Pattern. To determine which methods were actually called, participants would have to determine which classes implemented the interface and then begin new traversals from these methods. This forced them to perform new call hierarchy searches, losing their place. Participants sometimes said that trying to discover a listener was disheartening, as it signified there was much more to understand.



```
Members calling 'send(EBMessage)' – in workspace
▼ send(EBMessage) : void – org.gjt.sp.jedit.EditBus
  ▶ _closeBuffer(View, Buffer) : void – org.gjt.sp.jedit.jEdit (2 matches)
  ▶ _delete(Object, String, Component) : boolean – org.gjt.sp.jedit.io.Favor
  ▶ actionPerformed(ActionEvent) : void – org.gjt.sp.jedit.gui.DockableWin
  ▶ actionPerformed(ActionEvent) : void – org.gjt.sp.jedit.gui.DockableWin
  ▶ activatePlugin() : void – org.gjt.sp.jedit.PluginJAR
  ▶ addBufferToList(Buffer) : void – org.gjt.sp.jedit.jEdit
  ▶ addMarker(char, int) : void – org.gjt.sp.jedit.Buffer
  ▶ addPluginJAR(String) : void – org.gjt.sp.jedit.jEdit (2 matches)
  ▶ addToFavorites(String, int) : void – org.gjt.sp.jedit.io.FavoritesVFS
  ▶ checkFileStatus(View) : int – org.gjt.sp.jedit.Buffer
  ▶ close() : void – org.gjt.sp.jedit.EditPane
  ▶ closeAllBuffers(View, boolean) : boolean – org.gjt.sp.jedit.jEdit
  ▶ closeView(View, boolean) : void – org.gjt.sp.jedit.jEdit
  ▶ commitTemporary(Buffer) : void – org.gjt.sp.jedit.jEdit (3 matches)
  ▼ createEditPane(Buffer) : EditPane – org.gjt.sp.jedit.View
    ▼ restoreSplitConfig(Buffer, String) : Component – org.gjt.sp.jedit.Viev
      ▼ setSplitConfig(Buffer, String) : void – org.gjt.sp.jedit.View
        ▶ resplit() : void – org.gjt.sp.jedit.View
        ▼ View(Buffer, ViewConfig) – org.gjt.sp.jedit.View
          ▶ newView(View, Buffer, ViewConfig) : View – org.gjt.sp.jedit.
    ▶ split(int) : EditPane – org.gjt.sp.jedit.View
  ▶ deactivatePlugin(boolean) : void – org.gjt.sp.jedit.PluginJAR
  ▶ exit(View, boolean) : void – org.gjt.sp.jedit.jEdit (2 matches)
  ▶ finishSaving(View, String, String, String, boolean, boolean) : void – org.
  ▶ hideDockableWindow(String) : void – org.gjt.sp.jedit.gui.DockableWind
  ▶ load() : void – org.gjt.sp.jedit.search.SearchAndReplace
  ▶ load(View, boolean) : boolean – org.gjt.sp.jedit.Buffer
  ▶ loadMacros() : void – org.gjt.sp.jedit.Macros
  ▶ newView(View, Buffer, ViewConfig) : View – org.gjt.sp.jedit.jEdit
  ▶ openFile(View, String, String, boolean, Hashtable) : Buffer – org.gjt.sp.j
  ▶ propertiesChanged() : void – org.gjt.sp.jedit.Buffer
  ▶ propertiesChanged() : void – org.gjt.sp.jedit.jEdit
  ▶ removeAllMarkers() : void – org.gjt.sp.jedit.Buffer
  ▶ removeMarker(int) : void – org.gjt.sp.jedit.Buffer
```

Figure 10.6. Participants disliked Eclipse's call hierarchy because it did not provide search and did not help them to think as visually as in REACHER.

One participant tried to use dynamic, rather than static, investigation, and faced different challenges. To use the debugger to investigate a method, he first had to find a user command which invoked it, and he statically traversed upstream using the call hierarchy. After finding a command, he ran the program and invoked it, but found that conditionals prevented the path he wanted to see from executing. Returning to static investigation, he tried to find when they were true. But even after figuring out how to invoke the functionality, he faced a further challenge. To find paths from an origin to the target, he breakpointed the target, repeatedly hit the breakpoint, and investigated the paths. But as the target was widely called by methods other than the desired origin, many of the times that the breakpoint was hit were not paths from the origin. While he tried to only investigate those paths from the origin, he occasionally forgot to check and investigated the wrong paths.

All participants began using REACHER by opening the origin method described in the task, invoking a downstream search, and expanding the resulting paths (all of the tasks were about downstream searches, as it was possible to hang or crash REACHER with a few upstream searches). While participants often had a correct answer early in the task, they then spent most of their time better understanding the code to be sure of their answer, using REACHER to navigate to callsites along the path and discover what the calls were doing. Thus, the time differences in Figure 10.5 under-represent the real differences. Several attempted to more precisely determine in which situations different paths may execute by inspecting conditionals and trying to understand when they might be true by tracing the data that flowed into them.

As participants read methods in the editor, REACHER'S call graph provided context and helped them to stay oriented:

> *I like it a lot. It seems like an easy way to navigate the code. And the view maps to more of how I think of the call hierarchy.*
> *It seems pretty cool if you can navigate your way around a complex graph.*

Without REACHER, participants were often disoriented:

> *Now I'm getting a little confused*
> *Where am I? I'm so lost.*
> *I think I lost where I am in this silly tree.*
> *These callstacks are horrible.*
> *Where was I?*
> *There was a call to it here somewhere, but I don't remember the path.*
> *I'm just too lost.*

All participants reported that tasks with REACHER were easier; most had strongly positive impressions:

> *REACHER was my hero. … It's a lot more fun to use and look at.*
> *It's very cool actually. You don't have to ... go through many, many files.*
> *Oh, this is really great, how do you find this stuff [methods along paths]?*
> *It seems really useful.*
> *It's pretty cool.*
> *You don't have to think as much.*

Many felt that tasks without REACHER were very difficult:

> *Ah, this is going to get miserable isn't it.*
> *This is crazy.*
> *This is pretty ugly.*
> *My intuition about what might send a message to EditBus are probably wrong. So if I start drilling down into calls, I may miss something.*

Participants were able to successfully use the notation on the call graphs to understand paths. We did not observe any difficulties developers experienced understanding the notation.

Failing tasks while using REACHER was infrequent (22%) but not absent. 6 of the 8 failures were in tasks 1 and 3. Some of these failures were caused by failing to find all of the paths due to overlapping edges or paths that zigzagged through the graph. Others were caused by participants focusing on part of a path and missing an icon on the rest of the path. For example, one participant failed task 3 because they missed a ? icon at the end of a long path. Even for participants that succeeded, following paths was hard. One participant suggested highlighting the path from the current node to a root.

Our study revealed a number of other usability problems that still remained with REACHER. Edges that passed through methods or overlapped were initially confusing until users discovered the highlighting feature. Some participants found it difficult to visually locate targets in the call graph. While these methods are already rendered using a distinctive black fill and white text, participants suggested making them even more easily recognizable. Participants failed to notice that incoming and outgoing edges intersect nodes at different positions but instead relied on popups to disambiguate the direction of backward edges. One participant suggested indicating edge direction with arrows. A few participants wished to disentangle cluttered visualizations by dragging methods and manually overriding their layout positions.

10.3.3. Limitations

This study has several limitations. By phrasing the task instructions as reachability questions for the participants, we did not include the surrounding debugging or investigation task context which normally motivates users to ask these questions. While participants felt that searching along control flow was representative of their actual work, several felt that questions about path attributes (e.g., how many times…) were contrived. We included these questions to make sure that our visualization was clear and usable. Our tasks only included downstream reachability questions and did not assess the usability of REACHER for upstream reachability questions. Unlike most developers in the field, our participants had no experience in the codebase. Developers with more knowledge might more successfully predict where they should navigate.

10.4 Conclusions

Studies of REACHER demonstrated that it helps developers to explore code more easily and effectively, transforming a tedious, frustrating, disorienting, guess-work-filled task into one which most participants finished successfully. Our tasks replicated the challenges in exploring code that our and other studies have repeatedly found that developers face – finding methods, staying oriented, and understanding paths – and demonstrated that a combination of search, task-specific visualization, and IDE integration makes code exploration significantly easier. When faced with highly branching, long, and confusing paths, REACHER'S most significant benefit appears to be the ability to search, which helped developers to more quickly locate far-away methods and statements. But REACHER also helped support the subsequent work of understanding and reasoning about the path. Participants traced call graph paths to identify properties of paths. Participants ultimately wanted to see the code behind these paths, and used REACHER to quickly jump among the methods on the paths.

11. CONCLUSIONS AND FUTURE WORK

Understanding and exploring code is a central part of what makes developing software time-consuming and challenging. Seeing software developers as users of their tools and languages leads to questions about how effectively these tools and languages help developers with their work. Studying how developers work and the problems they face suggests ways that tools can be improved to help them work more effectively. While software engineering researchers have worked for decades on approaches to help developers understand and explore code, studying how developers work brings a new perspective, revealing questions developers ask and strategies developers use that can be more directly supported. This dissertation found that one of the key challenges developers face is searching along control flow and that a tool that better supports this strategy can help developers answer their questions more quickly, stay oriented, and more effectively reason about software. But there are also many opportunities for future work.

11.1. Extensions to REACHER

There are a variety of ways in which REACHER could be extended. On an engineering level, REACHER could be implemented for additional development environments and languages and made more robust. But it could also be extended to directly support a larger variety of questions and situations.

11.1.1. Filtering paths

Developers wondering how code behaves in a specific situation or context ask *filter* questions (e.g., what happens when this is true?)(see Section 7.3). Several of developers' reported hard-to-answer questions dealt with behavior in specific situations (see Chapter 6), such as when an exception occurs or an operation times out.

REACHER could directly support filter questions by filtering the paths REACHER considers and visualizes. For example, a developer investigating the behavior after an operation fails might filter to see only those execution paths that occur in this case. Selecting the method's call site in the code editor, the developer could select an option to "Show paths when this is" and select the constant indicating a failure. REACHER would then generate a new search beginning at the current point in code and filter the displayed call graph paths to those that execute when the operation fails. The code editor could also be scoped to those paths through conditionals that execute in this case. Of course, methods might execute in different contexts, with different resulting feasible paths. REACHER's call graph visualization could allow these alternative contexts to be selected by selecting call edges corresponding to different contexts. Developers could further refine the situation of interest by filtering additional expressions.

Implementing filtering poses several challenges. For expressions that sometimes contain constants and are modeled with a symbolic value in the summary, filtering could be implemented by simply assigning the constant value to the expression. The summary could then be used, as normal, to build a static trace and a call graph. However, developers might also wish to filter expressions that do not contain constants. In the summary, these expressions are modeled as $\top$. Unlike expressions containing constants, no paths through the summary are forked when these expressions occur in guards. This helps FFPA reduce the paths through the summary. As a result, filtering these expressions requires a new summary to be created. But only the summary for the method containing the filtered expression needs to be recomputed.

11.1.2. Comparing paths

As developers propose and implement changes, they consider their implications. For example, in the Exploration Lab Study, developers introduced a bug by setting a flag in a particular situation, causing a control flow path to no longer execute, and preventing a required update. In the Reachability Observations Study, a developer asked which paths would change if a table were not initialized at startup. And developers might wish to understand how methods behave differently when called in different contexts or situations. All of these are examples of *compare* questions.

REACHER could more directly support compare questions by using color to associate paths with the situations in which they execute. For example, a method called by three different callers might, depending on the caller, behave differently. This could be visually encoded by assigning the incoming edge from each caller a unique color; outgoing edges could then be assigned the colors corresponding to the contexts in which they execute (see Figure 11.1). These colors could be used further downstream to associate behavior with calling context.

Colors could also encode behavior changes resulting from code edits. After an edit and file save, REACHER might update the call graph visualization to reflect the new behavior. But, rather than simply update, REACHER could also show changes to the call graph, using colors to encode calls that have been added or removed. Such a design might have helped the developer in the Exploration Lab Study realize that his change had unintentionally changed how the code was working.

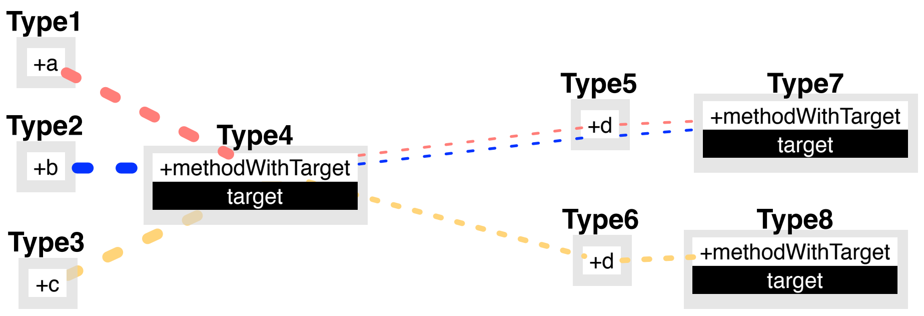


Figure 11.1. A mockup showing how color might be used to distinguish the behavior of methods in different situations.

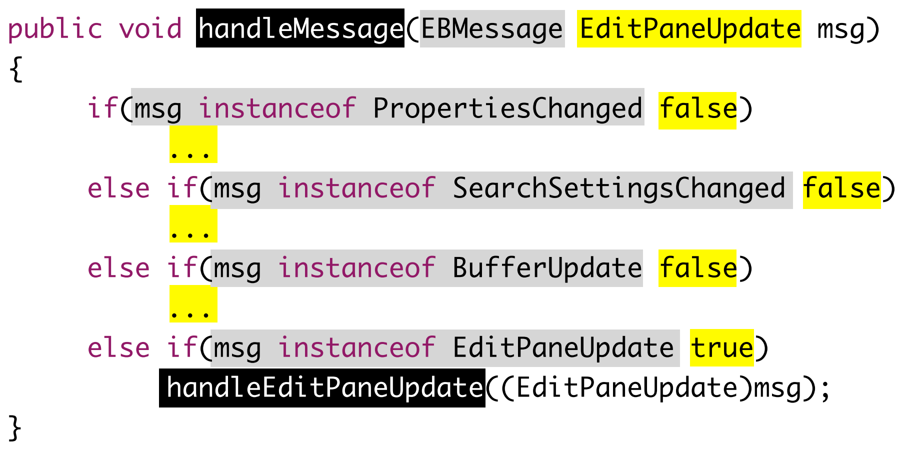
Compare could also be used with filter to compare situations. For example, REACHER could show how code behaves differently when a table is not initialized. A developer might first invoke a filter command on the call to initialize the table – "What happens if this doesn't execute?" REACHER's call graph visualization could then show which calls downstream would no longer execute.

Implementing compare requires determining a method of computing differences between static traces. Equivalent methods in each static trace must be matched and additions and removals found. Traditional algorithms for comparing versions of source code might be able to be adapted to this task. However, static traces differ in including methods in multiple contexts.

11.1.3. Depicting paths in a source view

REACHER's call graph visualization provides a high-level overview of code, relying on a code editor to let developers see the details of code. Some of the questions developers ask involve reasoning about these details. For example, a developer trying to understand how a method might behave differently in contexts with differing flags might try to follow and understand the paths *within* the method in each case. Or a developer reasoning about how code behaves when an operation fails might wish to see the paths through a method executing in this situation.

REACHER could more directly answer these questions by providing a view of source annotated and filtered to reflect REACHER's current view of the source. For example, in a context in which a method is called with a specific parameter, REACHER could show how this parameter's value is passed through the method and how it influences which branch is taken (see Figure 11.2.). In conjunction with filter, this view could be used to understand what a section of code does in a specific situation, such as an operation timing out, an error code being returned, or for a particular type of input. For example, the developer reasoning about how code behaves differently when a table is not initialized might first identify some of the differences in calls being made using the REACHER's call visualization and then navigate to the call site in the code editor. Rather than reconstruct why the calls are different, the source view could directly show how, in different situations, the values of parameters differ, causing different paths to be followed, and resulting in different calls executing.



```
public void handleMessage(EBMessage EditPaneUpdate msg)
{
     if(msg instanceof PropertiesChanged false)
          ...
     else if(msg instanceof SearchSettingsChanged false)
          ...
     else if(msg instanceof BufferUpdate false)
          ...
     else if(msg instanceof EditPaneUpdate true)
          handleEditPaneUpdate((EditPaneUpdate)msg);
}
```

Figure 11.2. A mockup of how REACHER might depict paths through code. In this example, REACHER sees `handleMessage()` execute in a context in which the parameter `EBMessage` is an instance of the type `EditPaneUpdate`. This allows REACHER to resolve the branch taken at a series of conditionals. The code view depicts this by showing the value of expressions and collapsing portions of the source that do not execute in this context.

Implementing this feature poses several design challenges. Most importantly, the code editor must clearly differentiate expression values from the expressions themselves. The expression's code could be replaced with its value, but this would make understanding how the value was obtained more difficult. Displaying both (as in Figure 11.2.) might be challenging for long lines of code with little extra space and needs to clearly visually differentiate expressions from their values.

Depicting paths in code could also be used with dynamic (rather symbolic) values independently of REACHER. For example, a logging tool could record values of expressions as code executes and then let developers inspect the source with these values embedded. Methods could be viewed in each of their contexts. Compared to inspecting a log file or using the debugger to step and hover to see values, such a view might provide a more effective overview, making it easy to see, at a glance, how code is behaving and ensure it is behaving as expected. Such a view might even be useful in computer science education to help students understand complex algorithms more easily.

11.1.4. Tracking values through fields

FFPA conservatively models values read from a field as $\top$. This reduces the precision of FFPA: FFPA can never determine the branch taken at conditionals dependent on field reads. But fields often store constants, such as flags reflecting the abstract state of an object. For example, one of the bugs created in the Exploration Lab Study was caused by a flag being read from a field which encoded the current state of the object.

An earlier prototype of FFPA tracked values through fields by assuming that there was only a single instance of each type. Making this assumption greatly simplifies the problem, essentially converting every field into a global variable. But this assumption also creates both false positive and false negatives when eliminating infeasible paths. When there are multiple instances of a type, values written into one overwrite another, resulting in arbitrary and spurious data flow.

A more accurate approach could use a points-to analysis (e.g., [S96]) to compute a set of (abstract) objects to which each field reference might refer. Unfortunately, points to analyses are relatively slow interprocedural analyses. After a method has been edited, the slow interprocedural analysis might need to be run again, unless a more complex incremental analysis could be used. And, even given such a slow and elaborate analysis, it is unclear how precise such an analysis might be.

Alternatively, REACHER might instead allow the developer to interactively adding annotations to describe which abstract objects might be pointed-to at each reference. In combination with following data flow paths (see 11.1.6), this might allow the developer to interactively describe what objects might exist.

11.1.5. Following paths through frameworks

FFPA treats paths through code for which the developer does not have source in their workspace (e.g., frameworks, libraries) as cut-points (see Section 8.3). As a result, the call graph visualization does not show call graph paths through framework methods, and developers cannot search for paths through frameworks. In some situations, this might be beneficial for the developer: this limits the scope of the code being examined to their own source code. But in other situations, developers wish to understand paths through a framework. For example, developers attempting to understand how and when call-backs occur might wish to search for paths from their code, into the framework, and back into their code through call backs.

FFPA could be extended to include framework paths. In many cases (e.g., the Java Standard Libraries), the source code is readily available on the developer's computer, and often even indexed by Eclipse. FFPA could simply use this source code. However, this might make most static traces far longer and could lead to many more connections between code. Moreover, paths that enter into event dispatch code could be potentially problematic if FFPA is not precise enough to determine which path is followed in a specific context, resulting in many more infeasible paths.

11.1.6. Searching along data flow

REACHER only currently supports searches along control flow. However, reachability questions also include searches across data flow (see Section 1.2 and Tables 7.1 and 7.2). Developers search along data flow to understand how variables get their value or understand where a value might be used. Current development environments partially support such questions by providing reference searches. However, to traverse data flow from variable to variable, developers must manually perform new reference searches, adding extra overhead and making it easy to get lost. And, just as in control flow searches, developers must guess which data flow leads to statements of interest.

Developers might also ask data flow questions to deal with REACHER's imprecision. Like all static analyses, REACHER produces a conservative approximation of paths that might execute, using a ? icon on calls that might or might not execute. Developers may wish to determine the conditions in which these paths are feasible. REACHER indirectly supports answering these questions through code inspection – developers can use REACHER to navigate to methods along the paths and look for conditionals. However, REACHER could more directly support these questions by letting developers see which paths lead to specific values.

REACHER's design could be extended to more directly support searching along data flow and viewing data flow paths in the call graph visualization. After selecting an expression in the code editor, developers might invoke an upstream or downstream search on this expression. After first generating upstream or downstream control flow, REACHER would next identify all the statements data dependent on the expression. These statements could then be shown in the call graph visualization and highlighted in the code editor. Figure 11.3 shows three designs, with varying levels of detail, for incorporating data flow into the call graph visualization.

However, while showing all data dependent statements might be appropriate when their numbers are small, in other cases this might involve a substantial fraction of a codebase. To deal with this issue, REACHER might limit the number of data dependencies shown, perhaps bounding them to a specific number of intermediate assignments statements or length in the call graph. Developers could then search along data flow paths for statements of interest using an additional mode in the search pane. Alternatively, developers might wish to search for paths with specific values or to group paths by value.

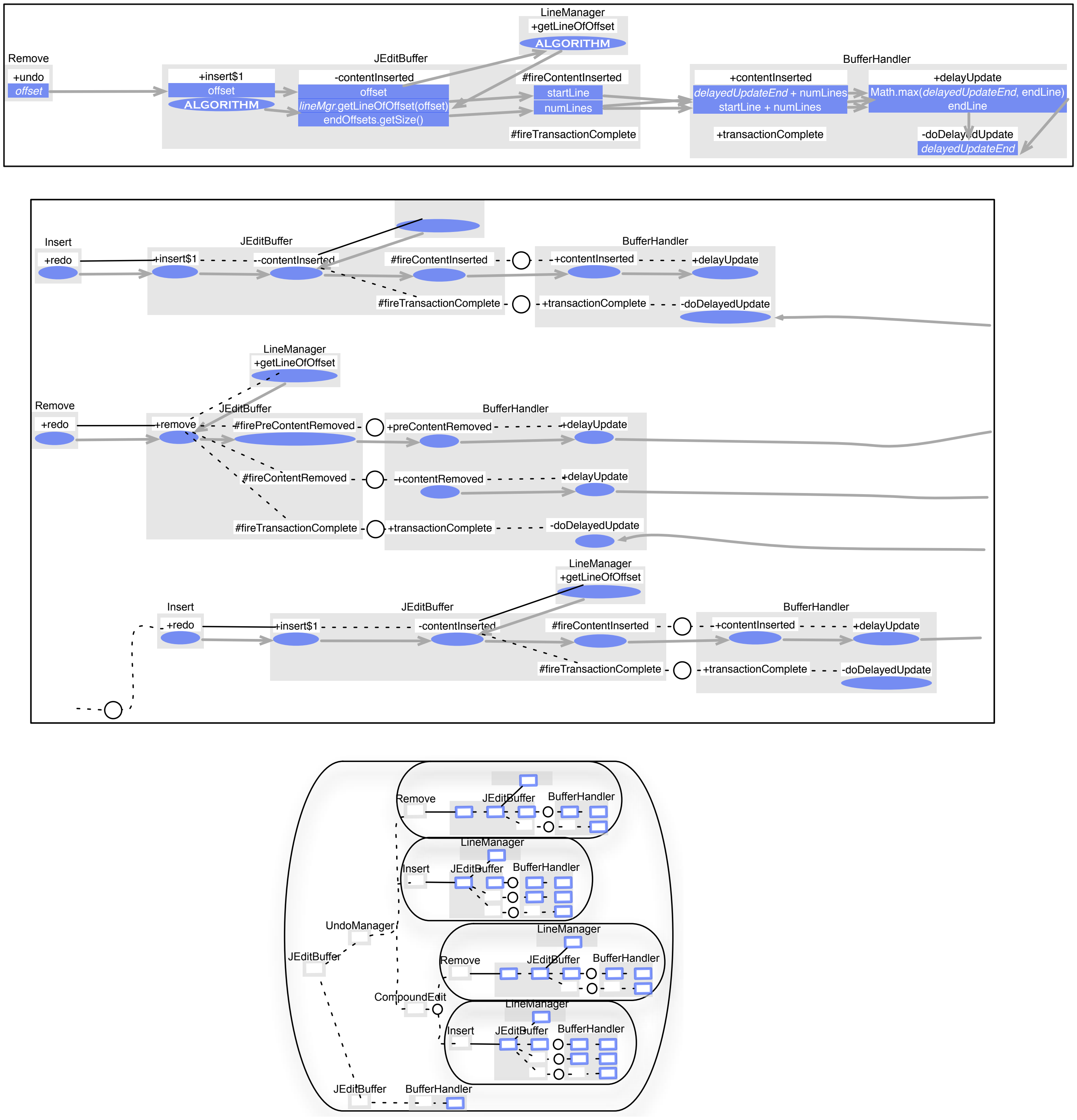


Figure 11.3. Three alternative designs for depicting data flow. In the first, each data-dependent expression is included. In the second, an oval indicates that a method is data dependent. In the third, data dependent methods are indicated with an outlined box.

11.1.7 Directly supporting higher-level questions

Developers sometimes ask questions that require them to aggregate, select, and make sense of search results. For examples, developers ask questions such as "In what situations or user scenarios is this called?" or "What parts of this data structure are modified by this code?" These questions are only indirectly supported by REACHER, as they ask about "parts of a data structure" or "situations or user scenarios" that are higher-level than the individual methods or fields REACHER depicts.

One way to more directly support these questions is by aggregating items in the list of search results. For example, rather than seeing a list of methods related to many user scenarios, a developer might instead see these grouped into a small number of user scenarios. This would allow the developer to see the user scenarios or situations in which something is called. Similarly, a user might wish to see method calls or field writes they are searching for aggregated or grouped by the parts of the data structure they deal with or perhaps by the classes containing the search results. For example, to understand what data is being mutated, developers might want to see that some of the data is mutated in functionality related to a text editor window (through calls through the text edit class to other classes) and other data is being mutated in code for file handling (through calls through a file class to other classes). After clicking on an aggregate search result and adding it to the visualization, REACHER could display the aggregate as a single box, reducing the amount of clutter, and only expand it if the user asks for it to be expanded. This would allow developers to use REACHER at a higher level of abstraction to reason about relationships between groups of methods or fields.

In other cases, developers may be interested in reasoning about methods or fields based on their properties. For example, some methods in a collection class do not modify the collection while others do. A developer might wish to see only those methods that modify a collection that is defined in a framework (for which field writes are not directly visible). For the FOLDS task, developers were interested in calls to framework methods that are slow and expensive rather than all calls into a framework. Annotations could be added to methods to reflect these properties, either directly in the source or through REACHER'S own representation of the method. REACHER could then let developers search for methods that have been marked as slow or that have some other property of interest.

11.2. Answering rationale questions

The studies reported in Chapter 4 – 7 revealed a number of important problems beyond those addressed by REACHER. One of the most frequently reported hard-to-answer questions about code is why it is implemented the way that it is. For example, a developer wished to uncomment a code block to re-enable some functionality, but was concerned that there might be unpredictable consequences [LM10-3]. She analyzed the surrounding code, but could not see any obvious consequence to re-enabling the block. She found the commit comment, surprised to find she had herself commented it out two years ago, but she had not indicated the reason for the change and could not remember it. Uncommenting it, she ran the unit tests, which passed. Hours later, she heard from her teammates that the re-enabled code was actually broken for some rare input values used by others. Finally confident she understood what was happening, she implemented a fix and committed the change.

Developers ask rationale questions such as "*Why was it done this way?*" or "*Why wasn't it done this other way?*" (see Section 6.2.1). Two-thirds of developers rated answering rationale questions as a serious problem (see Table 4.2). Difficulties answering rationale questions waste developer time, lead to code decay, and cause bugs.

Developers struggle to answer rationale questions because the reasons for decisions are not captured the moment they are made. And with good reason: there is no short-term incentive to writing down the rationale. Many systems have investigated complex notations for capturing design rationale at the level of high-level design or design documents [MC96] or even at the level of code through IDE integration [BB08]. While systems have demonstrated promise in helping developers to *answer* rationale questions [BB08], no system has demonstrated that developers will use it to write rationale down. While some have proposed ideas for incentives (e.g., capturing design rationale exchanged in code reviews [SV09]), these ideas have not made their way into actual design rationale systems. As a result, developers who might use a design rationale system currently have little short-term incentive to write it down.

There are a number of ways that developers might be incentivized to express rationale. One solution might be to provide a lightweight notation for expressing rationale in comments, providing both (1) short-term benefits to expressing rationale by leveraging the rationale to support programming and program navigation and (2) long-term benefits in helping developers more easily answer design rationale questions. To realize this goal, approaches must be found to express design decisions and then incentivize developers to express this rationale. There are many open questions confronting such an approach. Do developers making decisions think about the decisions that future developers later ask rationale questions about? How can these decisions be captured without imposing an undue burden on the original developers? How much information needs to be captured?

11.3. Using questions to evaluate contract specifications

Another opportunity for future research is in the area of contracts and specifications. This has long been a major topic of research in software engineering (see Section 2.2.1). In practice, many developers use assertions to express contracts. Yet, developers still report hard-to-answer questions such as "*What is the intent of this code?*", "*What does this do in this case?*", and "*What assumptions about preconditions does this code make?*" How effectively do contracts help developers answer these questions? Do these problems reflect situations where developers have not expressed any contract? How well does the information developers can express in contracts and specifications match the questions developers later ask? In what ways can such systems be designed to more effectively express and capture the information developers later want to know? Studies could investigate these questions by evaluating the effectiveness of contracts to answer developers' hard-to-answer questions, potentially revealing design implications for future contract systems that better match the questions developer ask.

11.4. Conclusions

Designing a useful tool requires understanding how a tool affects work and ensuring that it addresses an important problem that developers face. This dissertation demonstrates that a successful tool can be designed using data about how developers work. A series of studies investigated how developers understand and explore code and found that reachability questions are an important problem that developers face. Observations of developers demonstrated that problems caused by reachability questions were often of long duration, routinely take tens of minutes to answer, and impacted software quality by causing bugs. Surveys and observations of developers in the field demonstrated that reachability questions are a frequent part of developers' work. REACHER was designed to solve these problems by directly supporting developers' work and successfully helps developers answer many reachability questions faster and more successfully. Studying developers' information needs is a key future direction for software engineering research. Designing tools that help developers to more effectively satisfy their information needs can help to improve software development.

Appendix 1: Materials from Activities Survey (Chapter 4)

This survey is being conducted by the Human Interactions in Programming team in Microsoft Research. It will take about [estimate] minutes to complete. All information will be reported anonymously and typically in aggregate form. Your response is important to us – you are one of a small number of randomly selected MS employees and thus represent many of your peers.

Our goal is to learn more about the tools, techniques, and problems faced by people developing software at Microsoft. The results of this survey will be used to inform the design of tools to improve the software development process.

Each employee who submits a completed survey will be entered into a drawing for one of ten $50 Amazon gift certificates. The drawing is open only to first-party recipients of the email invitation. Surveys must be completed by 7/8/2005 to be eligible. See [URL elided] for additional details on the drawing.

For more information about our project or for any questions about the survey contact Gina Venolia or Thomas LaToza. Thank you for your help!

Demographics

1. What best describes your primary job function? [Radio: SDE, SDET, STE, PM, Dev lead, Test lead, Architect, Other] **JobFunction**

1. Primary job function (if you answered "Other" above) [Free text] **JobFunction**
2. What best describes your role? [Radio: Individual contributor, Lead, Manager, Executive, Other] **Role**
3. Role (if you answered "Other" above) [Free text] **Role**
4. Gender [Radio: Female, Male, Decline to state] **Gender**
5. Age [Radio: Teens, 20's, 30's, 40's, 50's, 60's, 70's, Decline to state] **Age**
6. Education in computer science, software engineering, or related field [Radio: None, Some college, Undergrad degree, Some grad school, Masters degree, PhD] **Education**
7. Years [Grid vs. Less than 1, 1, 2, 3-5, 6-10, 11-15, 16-20, 21-25, 26+]
   a. Programming **YearsProgramming**
   b. At Microsoft **YearsMicrosoft**
   c. In current team **YearsTeam**
   d. In current codebase (source tree or depot) – forks are not considered new codebases **YearsCodebase**
8. Number of codebases enlisted in over my entire time at Microsoft [Radio: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10+] **CodebasesEnlisted**

Team Demographics

9. Business unit [Radio: Windows Client, Information Worker, Business Solutions, Servers and Tools, Mobile and Embedded Devices, MSN, Home and Entertainment, Other] **BusinessUnit**
10. Business unit or organization (if you answered "Other" above) [Free text] **BusinessUnit**
11. Product [Free text] **Product**
12. Most of the code in my current codebase is written in [Radio: C, C++, C#, T-SQL, Other] **ProgrammingLanguage**
13. Programming language (if you answered "Other" above) [Free text] **ProgrammingLanguage**
14. My current codebase has a clear architecture [Radio: Strongly disagree, Disagree, Somewhat disagree, Neutral, Somewhat agree, Agree, Strongly agree] **ClearArchitecture**
15. Improving the architecture of our codebase is a priority for our team [Radio: Strongly disagree, Disagree, Somewhat disagree, Neutral, Somewhat agree, Agree, Strongly agree] **ArchitectureImprovement**
16. My team has useful information (schedules, architecture diagrams, bug lists, …) displayed in common areas (hallway, lounge, kitchen, …) [Yes/No] **WallDisplays**
17. When I use Visual Studio, I use [Radio: Visual Studio 6.0, Visual Studio 2002, Visual Studio 2003 (Everett), Visual Studio 2005 (Whidbey), Other version of visual studio, I never use Visual Studio] **VSVersion**
18. I am proud of the way my team builds software [Radio: Strongly disagree, Disagree, Somewhat disagree, Neutral, Somewhat agree, Agree, Strongly agree, Decline to state] **TeamMorale**

Activities

For the remainder of this survey the following definitions will be relevant. Code related activities may involve shipping product code, test infrastructure, internal tools, or any other work related code. Activities may use more than one type of tool (e.g. editor, debugger, whiteboard, web search), and tools may be used for more than one activity.

- Designing code – Analyzing a new problem and mapping out the broad flow of code which will be used to solve the problem. This includes drawing pictures on a whiteboard or using Visio. This does not include communication of any sort such as writing up a design in a design doc or email, having a design review meeting, or doing design with a teammate.
- Writing new code – Creating a new method, source file, or script and getting it to a compilable state.
- Understanding existing code – Determining information about code including the inputs and outputs to a method, what the call stack looks like, why the code is doing what it is doing, or the rationale behind a design decision. This includes using a profiler to find frequently executed functions, a debugger to localize a bug, an editor to find all references to a variable, or a search tool to find a method definition. This

does not include communication of any sort such as reading a design document or asking a teammate a question.

- Editing existing code – Editing existing code and returning it to a compilable state.
- Unit testing code – Ensuring that code is behaving as expected. This includes writing and running automated, ad hoc, or other tests, using a debugger to verify a change, or tracing.
- Communicating about code – Any computer mediated or face to face communication about information relevant to a coding task. This includes reading or writing design docs, specs, bug descriptions, checkin comments, or email. It includes finding example code written by others on the web. It also includes talking in your teammate's office, conducting a code review, instant messaging a teammate, or planned team meetings.
- Code related overhead – Any other code related activities including building, synchronizing code, or checking in changes.
- Activities not related to code – Any other activities included in your work time.

19. Time since last M0 or initial planning period of your team's primary deliverables [Radio: 0, 1-3 months, 4-6 months, 7-12 months, < 2 years, < 3 years, < 4 years, < 5 years; 5 years or more] **M0Distance**
20. Time until next RTM or release of your team's primary deliverables [Radio: 0, 1-3 months, 4-6 months, 7-12 months, < 2 years, < 3 years, < 4 years, < 5 years; 5 years or more] **RTMDistance**
21. Percent of work time last week that I spent (for all percentage questions, percentages need not sum to exactly 100%) [Grid vs. 0%, 1%, 2%, 5%, 10%, …, 90%, 100%]
    a. Fixing bugs entered in a bug tracking tool (e.g Product Studio, …) **Corrective**
    b. Writing new features **Adaptive**
    c. Making code more maintainable **Perfective**
22. Percent of work time last week that I spent [Grid vs. 0%, 1%, 2%, 5%, 10%, …, 90%, 100%]
    a. Designing code **Designing**
    b. Writing new code **Writing**
    c. Understanding existing code **Understanding**
    d. Editing existing code **Editing**
    e. Unit testing code **UnitTesting**
    f. Communicating about code **Communicating**
    g. Code-related overhead **Overhead**
    h. Other code-related activities **OtherCodeActivities**
    i. Activities not related to code **NonCodeActivities**
23. Other code-related activities last week [Free text] **OtherCodeActivities**

Communicating About Code

If you did not communicate about code last week, please skip this section.

24. Of the time I spent communicating about code last week, the percent of time I used: [Grid vs. 0%, 1%, 2%, 5%, 10%, …, 90%, 100%]
    a. Face-to-face in planned meetings
    b. Face-to-face in unplanned meetings
    c. Email
    d. Instant messaging
    e. Phone
    f. Bug database
    g. Internal documentation (design documents, specs, team sharepoints)
    h. External documentation (MSDN, help files)
    i. The web
    j. Other
25. Other tools used last week (if you answered "other" above) [Free text]
26. This tool was effective for communicating about code: [Radio: Strongly disagree, Disagree, Somewhat disagree, Neutral, Somewhat agree, Agree, Strongly agree]
    a. Face-to-face in planned meetings
    b. Face-to-face in unplanned meetings
    c. Email
    d. Instant messaging
    e. Phone
    f. Bug reports
    g. Internal documentation (design documents, specs, team sharepoints)
    h. External documentation (MSDN, help files)
    i. The web
    j. Other (same as above)
    k. All tools I used, taken together

47. Of the time I spent communicating about code last week, the percent of time it was with [Grid vs. 0%, 1%, 2%, 5%, 10%, …, 90%, 100%]

a.. SDEs on my team

b. SDETs on my team

c. STEs on my team

d. PMs on my team

e. Other team members

f. Other SDEs, SDETs, or STEs at Microsoft

g. Other PMs at Microsoft

h. Other people at Microsoft

i. Other people not at Microsoft

Designing Code

If you did not design code last week, please skip this section.

27. Of the time I spent designing code last week, the percent of time I used: [Grid vs. 0%, 1%, 2%, 5%, 10%, ..., 90%, 100%]
    a. Whiteboard
    b. Paper
    c. Word processor
    d. Visio
    e. Visual designers or development tools
    f. Source code editor
    g. Other
28. Other tools used last week (if you answered "other" above) [Free text]
29. This tool was effective for designing code: [Radio: Strongly disagree, Disagree, Somewhat disagree, Neutral, Somewhat agree, Agree, Strongly agree]
    a. Whiteboard
    b. Paper
    c. Word processor
    d. Visio
    e. Visual designers or development tools
    f. Source code editor
    g. Other (same as above)
    h. All tools I used, taken together\

Writing New Code

If you did not write new code last week, please skip this section.

30. Of the time I spent writing new code last week, the percent of time I used: [Grid vs. 0%, 1%, 2%, 5%, 10%, ..., 90%, 100%]
    a. Visual Studio editor
    b. Vi (vim, gvim, ...)
    c. Emacs
    d. SlickEdit
    e. Source Insight
    f. Notepad
    g. A SQL editor
    h. Other
31. Other tools used last week (if you answered "other" above) [Free text]
32. This tool was effective for writing new code: [Radio: Strongly disagree, Disagree, Somewhat disagree, Neutral, Somewhat agree, Agree, Strongly agree]
    a. Visual Studio editor
    b. Vi (vim, gvim, ...)
    c. Emacs
    d. SlickEdit
    e. Source Insight
    f. Notepad
    g. A SQL editor

   h. Other (same as above)
   i. All tools I used, taken together

Understanding Existing Code

If you did not understand existing code last week, please skip this section.

33. Of the time I spent understanding existing code last week, the percent of time I used: [Grid vs. 0%, 1%, 2%, 5%, 10%, …, 90%, 100%]
    a. Visual Studio editor
    b. Visual Studio debugger
    c. Vi (vim, gvim, …)
    d. Emacs
    e. SlickEdit
    f. Source Insight
    g. Notepad
    h. A SQL editor
    i. Other debuggers (WinDbg, CorDbg, ntsd, kd, …)
    j. A profiler
    k. Diff tool (windiff, bcdiff, …)
    l. Other
34. Other tools used last week (if you answered “other” above) [Free text]
35. This tool was effective for understanding existing code: [Radio: Strongly disagree, Disagree, Somewhat disagree, Neutral, Somewhat agree, Agree, Strongly agree]
    a. Visual Studio editor
    b. Visual Studio debugger
    c. Vi (vim, gvim, …)
    d. Emacs
    e. SlickEdit
    f. Source Insight
    g. Notepad
    h. A SQL editor
    i. Other debuggers (WinDbg, CorDbg, ntsd, kd, …)
    j. A profiler
    k. Diff tool (windiff, bcdiff, …)
    l. Other (same as above)
    m. All tools I used, taken together
36. Of the time I spent understanding existing code last week, the percent of time I spent: [Grid vs. 0%, 1%, 2%, 5%, 10%, …, 90%, 100%]
    a. Examining source code
    b. Examining source code check-in comments and diffs
    c. Examining high-level views of source code (UML diagrams, class hierarchies, call graphs, …)
    d. Running the code and looking at the results
    e. Running the code and examining it with a debugger
    f. Using debug or trace statements
    g. Other
37. Other techniques used last week (if you answered “other” above) [Free text]

38. This technique was effective for understanding existing code: [Radio: Strongly disagree, Disagree, Somewhat disagree, Neutral, Somewhat agree, Agree, Strongly agree]
    a. Examining source code
    b. Examining source code check-in comments and diffs
    c. Examining high-level views of source code (UML diagrams, class hierarchies, call graphs, …)
    d. Running the code and looking at the results
    e. Running the code and examining it with a debugger
    f. Using debug or trace statements
    g. Other (same as above)
    h. All techniques I used, taken together

Editing Existing Code

If you did not edit existing code last week, please skip this section.

39. Of the time I spent editing existing code last week, the percent of time I used: [Grid vs. 0%, 1%, 2%, 5%, 10%, …, 90%, 100%]
    a. Visual Studio editor
    b. Vi (vim, gvim, …)
    c. Emacs
    d. SlickEdit
    e. Source Insight
    f. Notepad
    g. A SQL editor
    h. Other
40. Other tools used last week (if you answered “other” above) [Free text]
41. This tool was effective for editing existing code: [Radio: Strongly disagree, Disagree, Somewhat disagree, Neutral, Somewhat agree, Agree, Strongly agree]
    a. Visual Studio editor
    a. Vi (vim, gvim, …)
    b. Emacs
    c. SlickEdit
    d. Source Insight
    e. Notepad
    f. A SQL editor
    g. Other (same as above)
    h. All tools I used, taken together

Unit Testing Code

If you did not unit test code last week, please skip this section.

42. Of the time I spent unit testing code last week, the percent of time I spent: [Grid vs. 0%, 1%, 2%, 5%, 10%, …, 90%, 100%]
    a. Running test cases
    b. Ad-hoc testing
    c. Using the Visual Studio debugger
    d. Using other debuggers (WinDbg, CorDbg, ntsd, kd, …)

   e. Using trace statements
   f. Other
43. Other tools or techniques used last week (if you answered “other” above) [Free text]
44. This tool or technique was effective for unit testing: [Radio: Strongly disagree, Disagree, Somewhat disagree, Neutral, Somewhat agree, Agree, Strongly agree]
   a. Running test cases
   b. Ad-hoc testing
   c. Visual Studio debugger
   d. Other debuggers (WinDbg, CorDbg, ntsd, kd, …)
   e. Trace statements
   f. Other (same as above)
   g. All tools or techniques I used taken together

Problems

45. This is a serious problem for me: [Radio: Strongly disagree, Disagree, Somewhat disagree, Neutral, Somewhat agree, Agree, Strongly agree]
   a. Understanding code that someone else wrote
   b. Understanding code that I wrote a while ago
   c. Having to switch tasks often because my current task gets blocked
   d. Having to switch tasks often because of requests from my teammates or manager
   e. Finding the right person to talk to about a bug
   f. Finding the right person to talk to about a piece of code
   g. Finding the right person to review a change before check-in
   h. Finding code related to a bug
   i. Finding the bugs related to a piece of code
   j. Understanding the history of a piece of code
   k. Understanding the rationale behind a piece of code
   l. Understanding who “owns” a piece of code
   m. Finding out who is currently modifying a piece of code
   n. Understanding the impact of changes I make on code elsewhere
   o. Being aware of changes to code elsewhere that impact my code
   p. Getting enough time with senior developers more knowledgeable about parts of code I’m working on
   q. Convincing developers on other teams within Microsoft to make changes to code I depend on
   r. Convincing managers that I should spend time rearchitecting, refactoring, or rewriting code
   s. Finding all the places code has been duplicated
46. When I couldn’t make progress on my current development task last week, the percentage of time it was because: [Grid vs. 0%, 1%, 2%, 5%, 10%, …, 90%, 100%]
   a. Someone “broke the build” (I couldn’t try out my changes because the system wouldn’t compile or run)
   a. I was waiting on a reply to an email, instant message or phone call from a teammate
   b. I was waiting on a reply to an email, instant message or phone call from someone on another team

c. I was waiting for a teammate to check in changes I needed
d. I was waiting for someone else to make a technical decision (team to decide on design change, …)
e. I was waiting for someone else to make a nontechnical decision (PM to make spec decision, triage to decide to take the fix, …)
f. I was looking for information that I couldn't find

Final Questions

47. Please notify me when the survey results are available [Checkbox]
48. I would be willing to participate in future user studies to improve the Microsoft software development process [Checkbox]
49. Other comments or suggestions [Free text]

APPENDIX 2: MATERIALS FROM ACTIVITIES INTERVIEWS (CHAPTER 4)

1. **Overview**
    a. Can you tell us at a very high level what type of work you do?
    b. How long have you been on this team?
2. **Team**
    a. Regular team meetings, informal team meetings
    b. How many people on your team?
        i. -How many people do you talk to regularly?
    c. How does ownership / load balancing work?
    d. What is the boundary between your team and other teams?
    e. Wall displays
    f. Ever use IM?

3. Followup

a. Do you ever make changes to code that you weren't already modifying to add a feature or fix a bug?

b. Does your team ever budget time to cleanup code?

c. Does your team do any high level M0 planning of how all of the changes will fit together in changing the system?

3. **Tools**
    a. VS, source insight, etc.
        i. If multiple, when do you use each?
4. **Design documents**
    a. How many / content / how updated?
        i. -If not, why not, comments instead?
5. **Rationale**
    a. Tell us about a time when you had a hard time understanding a piece of code
        i. -Bug fix / new feature, how start investigating, look at comments / docs, talk to people, etc.
    b. Email / face to face decisions
    c. What are you doing when you have problems understanding rationale
        i. Big picture or small picture
        ii. Would comments have helped? Or design docs?
6. **Code duplication**
    a. Is there code duplication on your codebase?
        i. -How find?
        ii. -Bugs because of?
        iii. -Code clones or co changes?
    b. Do you ever intentionally create duplicates?
7. **Interruptions**
    a. How often are you interrupted?

            i. -Work to manage, ….

8. **Understanding impact of change elsewhere on your code**
    a. Tell us about a time when some change elsewhere caused a bug in your code
        i. -Checkin emails
9. **Final thoughts**

APPENDIX 3: MATERIALS FROM FOLLOW-UP SURVEY (CHAPTER 4)

Developer Patterns Survey

This survey is being conducted by the Human Interactions in Programming team in Microsoft Research. It will take about 20 minutes to complete. The results of this survey will help us design tools to improve Microsoft's software development process. Any published data will be anonymized. Your response is important to us as you are one of a small number of randomly selected MS employees and thus represent many of your peers. Each employee who submits a completed survey will be entered into a drawing for one of five $50 Amazon gift certificates. The drawing is open only to first-party recipients of the email invitation. Surveys must be completed by 8/31/05 to be eligible. See ____ for additional details on the drawing. For more information about our project or for any questions about the survey contact Gina Venolia (ginav) and Rob DeLine (rdeline). Thank you for your help!

After taking the survey click "Submit" to save your changes.

This survey is **not** anonymous

Demographics

1.	What best describes your primary job function?	○	SDE
		○	SDET
		○	STE
		○	PM
		○	Dev lead
		○	Test lead
		○	Architect

		○ Other...
2.	Primary job function (if you answered "Other" above) (Max Characters: 256)	
3.	What best describes your role?	○ Individual contributor ○ Lead ○ Manager ○ Executive ○ Other...
4.	Role (if you answered "Other" above) (Max Characters: 256)	
5.	Gender	○ Female ○ Male ○ Decline to state
6.	Age	○ Teens ○ 20's ○ 30's ○ 40's ○ 50's ○ 60's+ ○ Decline to state

Feature Team

Terms used in the survey:

- *Feature team* – The core group of developers that you work with.
- *Teammates* – The people in your feature team.
- *Collocated* – A person is collocated with you if their primary work location is within a minute (or so) walk of yours.

Please take a moment to think about exactly who makes up your feature team.

7.	How many people are there in your feature team? (Include yourself.) (Min Number: 1 - Max Number: 99)	
8.	How many of your teammates report to your immediate manager? (Include yourself.) (Min Number: 1 - Max Number: 99)	
9.	How many of your teammates report to a different manager? (Min Number: 1 - Max Number: 99)	
10.	How many of your teammates have SDE as their primary job function? (Include yourself as appropriate.) (Min Number: 0 - Max Number: 99)	
11.	How many of your teammates have SDET as their primary job function? (Include yourself as appropriate.) (Min Number: 0 - Max Number: 99)	
12.	How many of your teammates are collocated with you? (Include yourself.) (Min Number: 1 - Max Number: 99)	

Communication Patterns

In your last work week, approximately how many …

13.	… regularly-scheduled meetings about code did you have? (Min Number: 0 - Max Number: 999)	
14.	… scheduled-but-not-recurring meetings about code did you have? (Min Number: 0 - Max Number: 999)	
15.	… unplanned face-to-face discussions about code did you have with ***teammates***? (Min Number: 0 - Max Number: 999)	
16.	… unplanned face-to-face discussions about code did you have with ***non-teammates***? (Min Number: 0 - Max Number: 999)	
17.	… emails about code did you send to ***collocated*** team-mates? (Min Number: 0 - Max Number: 999)	
18.	… emails about code did you send to ***non***-collocated teammates? (Min Number: 0 - Max Number: 999)	
19.	… emails about code did you send to ***non-teammates***? (Min Number: 0 - Max Number: 999)	

Code Ownership

20. Please answer…

	Strongly agree	Agree	Somewhat agree	Neutral	Somewhat disagree	Disagree	Strongly disagree	N/A
There is a clear distinction between the code my feature team "owns" and the code owned by other tea	○	○	○	○	○	○	○	○
There is a clear distinction between the code that I "own" and the code owned by my teammates	○	○	○	○	○	○	○	○
There is code that my functional team owns that none of my team-mates (including me) owns	○	○	○	○	○	○	○	○
I find it more difficult to fix bugs in code that I don't own than code that I do own	○	○	○	○	○	○	○	○
I find it more difficult to do new feature work in code that I don't own than code that I do own	○	○	○	○	○	○	○	○

21. Do you think your team's code ownership practices are…

	Strongly agree	Agree	Somewhat agree	Neutral	Somewhat disagree	Disagree	Strongly disagree	N/A
… too strict?	○	○	○	○	○	○	○	○
… too loose?	○	○	○	○	○	○	○	○

Design Documents

Approximate number of design documents that were written by you or your teammates in the last year...

22.	... that were relevant to issues isolated within your team? (Min Number: 0 - Max Number: 999)	
23.	... that were relevant to issues that affected other teams? (Min Number: 0 - Max Number: 999)	
24.	... that were relevant to issues isolated within your team and were design reviewed? (Min Number: 0 - Max Number: 999)	
25.	... that were relevant to issues that affected other teams and were design reviewed? (Min Number: 0 - Max Number: 999)	
26.	... that were relevant to issues isolated within your team and are kept up-to-date? (Min Number: 0 - Max Number: 999)	
27.	... that were relevant to issues that affected other teams and are kept up-to-date? (Min Number: 0 - Max Number: 999)	

28. Do you often refer back to design documents that...

	Strongly agree	Agree	Somewhat agree	Neutral	Somewhat disagree	Disagree	Strongly disagree	N/A
... you've written?	○	○	○	○	○	○	○	○
... your teammates have written?	○	○	○	○	○	○	○	○

... non-teammates have written?	○	○	○	○	○	○	○	○

29. What do you think are the important benefits of design documents?

	Strongly agree	Agree	Somewhat agree	Neutral	Somewhat disagree	Disagree	Strongly disagree	N/A
They force the developer to think through the design	○	○	○	○	○	○	○	○
They build consensus around the design within the team	○	○	○	○	○	○	○	○
They build consensus around the design between teams	○	○	○	○	○	○	○	○
They document the rationale behind the design for devs encountering the code later	○	○	○	○	○	○	○	○
Other...	○	○	○	○	○	○	○	○

30. What are other important benefits of design documents (if you answered "Other" above)?
(Max Characters: 1000)

31. Do you think your team is good at using design documents…

	Strongly agree	Agree	Somewhat agree	Neutral	Somewhat disagree	Disagree	Strongly disagree	N/A
... for within-team issues?	○	○	○	○	○	○	○	○
... for cross-team issues?	○	○	○	○	○	○	○	○

Understanding Unfamiliar Code

32. Do you often find it difficult to understand...

	Strongly agree	Agree	Somewhat agree	Neutral	Somewhat disagree	Disagree	Strongly disagree	N/A
... what unfamiliar code is trying to accomplish?	○	○	○	○	○	○	○	○
... why unfamiliar code is taking a particular approach?	○	○	○	○	○	○	○	○

Code Duplication

33. In your last work week have you made repeated or related changes in code that was...

	Strongly agree	Agree	Somewhat agree	Neutral	Somewhat disagree	Disagree	Strongly disagree	N/A
... duplicated (by copy-paste or other means) in the source code?	○	○	○	○	○	○	○	○
... duplicated because of architectural constraints such as language, calling convention, etc?	○	○	○	○	○	○	○	○
... redundantly implemented independently by developers not aware of each others work?	○	○	○	○	○	○	○	○

... where an individual design decision got spread into multiple places in the source?	○	○	○	○	○	○	○	○
... in multiple product branches?	○	○	○	○	○	○	○	○
... in a large body of code imported from another team into my team?	○	○	○	○	○	○	○	○

34. Is your team's code difficult to manage because of code that was...

	Strongly agree	Agree	Somewhat agree	Neutral	Somewhat disagree	Disagree	Strongly disagree	N/A
... duplicated (by copy-paste or other means) in the source code?	○	○	○	○	○	○	○	○
... duplicated because of architectural constraints such as language, calling convention, etc?	○	○	○	○	○	○	○	○
... redundantly implemented independently by developers not aware of each others work?	○	○	○	○	○	○	○	○
... where an individual design decision got spread into multiple places in the source?	○	○	○	○	○	○	○	○
... in multiple product branches?	○	○	○	○	○	○	○	○
... in a large body of code imported from another team into my team?	○	○	○	○	○	○	○	○

Unit Testing

35.	What best applies to your team?	○ My team currently uses unit testing ○ My team tried unit testing but abandoned it ○ My team has not tried unit testing
36.	How many months has/had your team tried unit testing? (Min Number: 1 - Max Number: 99)	

37. Does your team...

	Strongly agree	Agree	Somewhat agree	Neutral	Somewhat disagree	Disagree	Strongly disagree	N/A
... currently use unit testing EXTENSIVELY?	○	○	○	○	○	○	○	○
... write unit tests BEFORE implementing the tested functionality?	○	○	○	○	○	○	○	○
... write the unit tests AT THE SAME TIME AS implementing the tested functionality?	○	○	○	○	○	○	○	○
... write the unit tests AFTER implementing the tested functionality?	○	○	○	○	○	○	○	○
... make SDEs responsible for writing unit tests?	○	○	○	○	○	○	○	○
... make SDETs responsible for writing unit tests?	○	○	○	○	○	○	○	○
... make SDEs responsible for maintaining unit tests?	○	○	○	○	○	○	○	○
... make SDETs responsible for maintaining unit tests?	○	○	○	○	○	○	○	○

38. What do you think are the important benefits of unit testing?

	Strongly agree	Agree	Somewhat agree	Neutral	Somewhat disagree	Disagree	Strongly disagree	N/A
They force the SDE to think through a design	○	○	○	○	○	○	○	○
They reduce build breaks	○	○	○	○	○	○	○	○
They reduce regressions	○	○	○	○	○	○	○	○
They isolate dependencies between teams	○	○	○	○	○	○	○	○
More bugs get caught before check-in	○	○	○	○	○	○	○	○
They help localize mistakes faster	○	○	○	○	○	○	○	○
They help fix mistakes faster	○	○	○	○	○	○	○	○
They encourage refactoring	○	○	○	○	○	○	○	○
Other...	○	○	○	○	○	○	○	○

39. What are other important benefits of unit tests (if you answered "Other" above)?
(Max Characters: 1000)

Agile Practices

40. Does your team use...

	Strongly agree	Agree	Somewhat agree	Neutral	Somewhat disagree	Disagree	Strongly disagree	N/A
... ''sprints,'' ie. a development cycle that lasts four (or so) weeks?	○	○	○	○	○	○	○	○
... ''scrum meetings,'' ie. a brief daily status meeting including all stake-holders?	○	○	○	○	○	○	○	○
... a ''bullpen'' or other open-floorplan space for the team?	○	○	○	○	○	○	○	○
... ''burndown'' estimate or chart, ie. a measure of the time remaining in the sprint?	○	○	○	○	○	○	○	○
... pair programming, ie. developers working together, shoulder-to-shoulder on a problem?	○	○	○	○	○	○	○	○
... an intentional policy for prefer-ring face-to-face over electronic communication?	○	○	○	○	○	○	○	○
... an intentional policy to involve customers (internal or external) deeply into design & planning?	○	○	○	○	○	○	○	○
... collective code ownership within the team	○	○	○	○	○	○	○	○
... other...	○	○	○	○	○	○	○	○

41. What other agile software development practices does your team use (if you answered "Other" above)? (Max Characters: 1000)

42. Do you think that your team should adopt agile software development methodologies...

	Strongly agree	Agree	Somewhat agree	Neutral	Somewhat disagree	Disagree	Strongly disagree	N/A
... more aggressively	○	○	○	○	○	○	○	○
... less aggressively	○	○	○	○	○	○	○	○

Final Questions

43. Do you want to be notified when survey results are available?

○ Yes
○ No

44. Would you be willing to participate in future user studies to improve Microsoft's software development process?

○ Yes
○ No

45. Other comments or suggestions
(Max Characters: 2000)

APPENDIX 4: MATERIALS FROM EXPLORATION LAB STUDY (CHAPTERS 5 AND 7)

Introduction

Information about software architecture concerns the important elements of a design, the relationships between them, and important designs made about them. While much work has been done to help architects document, analyze, or reverse engineer architecture, less is known about the use of architectural information during coding tasks. While it seems such information should be helpful, which information is most helpful, how it should be presented, and exactly how it would help remain less clear. To understand the requirements for future diagram or visualization systems, specifications, or coding conventions to better present architectural information during coding tasks, we are studying how developers currently interact with architectural information. We hope that a careful analysis of the questions developers ask, the information they seek, and the difficulties they experience will lead to the design of more useful future systems for presenting architectural information.

You will be asked to perform two change tasks. You will have 1.5 hours (beginning after you finish reading the actual task description) to work on each task. The tasks have been designed to be challenging, so you will likely not have time to investigate or design as much as you could with unlimited time constraints. Your goal is to complete each task with a well designed implementation that respects the existing architecture of the system as much as possible. You will be investigating and trying to change two complicated pieces of code in a codebase that you will have little time to explore. So do not feel discouraged if you are not able to accomplish as much as you might hope you could accomplish.

You will be provided with a copy of Eclipse 3.2. You may use any feature of Eclipse – including running the program – and may use Explorer to open any files created by jEdit – such as log files or files edited by jEdit – in Notepad or jEdit. But you may not use any other application (including a web browser). At the end of the first task, you will be given a fresh copy of the source to work on the second task. After each task, the experimenter will ask some questions about your understanding of the system, how you were working, and decisions you made in creating your implementation.

To understand how you are working, you will be asked to "think aloud" – talk about what you are thinking while you work. You should describe what you're trying to accomplish and what you're considering doing to accomplish your goals. If you find information that you've been seeking for a while, you should make sure that you say something about what you found. If you are distracted of forget to think aloud, the experimenter will prompt you to continue to think aloud by asking you about what you're doing. To allow us to analyze how you work, you will be recorded using a laptop audio recorder, a screen capture program, an

Eclipse event logger, and two video cameras. Your identity will not be revealed to anyone other than researchers in the research group. To ensure anonymity, only the researchers in the research group will have access to the audio and video recordings. However, transcripts of the audio recording with any personally identifying information removed and the screen recording may be used in public presentation of this work.

Eclipse Code Navigation Tutorial

Eclipse features a number of sophisticated features for navigating through source code. In this brief tutorial, you will try out several of the most useful features.

1. Press CONTROL-SHIFT-T and type “jedit” to open up the jedit class.
2. Press CONTROL-O to open the method outline and type getProperties to navigate to the method getProperties().
3. Hold down control and click on getProperties() (the method call on propMgr, not the method declaration). This moves you to the method declaration.
4. Go back to the previous method you were looking at by clicking on the left arrow secondmost from the right edge of the toolbar.
5. Place the cursor over the getProperties() method declaration, right click, and select “Open Call Hierarchy”. At the bottom of the screen, a tree view shows either the Callee Hierarchy or the Caller Hierarchy. Switch between them by clicking a button to the right of the Call Hiearchy tab.
6. Place the cursor on propMgr. All references to this field in the current Java file are now highlighted with a grey highlight both in the text editor and next to the scrollbar, allowing you to quickly find all of the references in the file.
7. Find all of the methods that assign the field propMgr by placing the cursor on top of it and selecting from the “Search” menu at the top of the screen “Write Access” and then “Project”.
8. Find all references to the jEdit type by selecting Java from the Search menus, typing in jEdit, and selecting “Type” in Search For.

jEdit Background

For your two tasks, you will be working with jEdit. jEdit is an open source text editor mainly intended for editing code written in a little over 50,000 LOC of Java. It was originally written by mostly one developer (Slava Pestov) but has more recently been edited by several other developers. Start jEdit by clicking the debug button the on the Eclipse toolbar (the fourth icon from the left).

Several important classes in jEdit can be considered components:

jEdit

jEdit is the main class in the application and contains the application entry point. It is mostly responsible for orchestrating interactions between other components such as opening and closing buffers, creating and switching views, managing plugins, and setting and retrieving global application properties.

View

The top level window for a jEdit application toolbars, user input, creating and deleting EditPanes, and visual look and feel. There can be more than one open view for a particular application – to create a new View simply select View.New View on the menu bar.

EditPane

The containing visual element for a JEditTextArea text editing control. There may be more than one when a view is split into multiple panes. EditPanes are responsible for switching buffers and managing markers (bookmarks).

JEditTextArea

jEdit's text control for editing text. Responsibilities include scrolling, selection, painting, buffer access, caret position, text input, and text transformations.

JEditBuffer

jEdit's representation of a file containing functionality for manipulating the text in a buffer.

Buffer (inherits from JEditBuffer)

jEdit's representation of a file containing rules about how the text file in memory is loaded and stored to disk.

EditBus

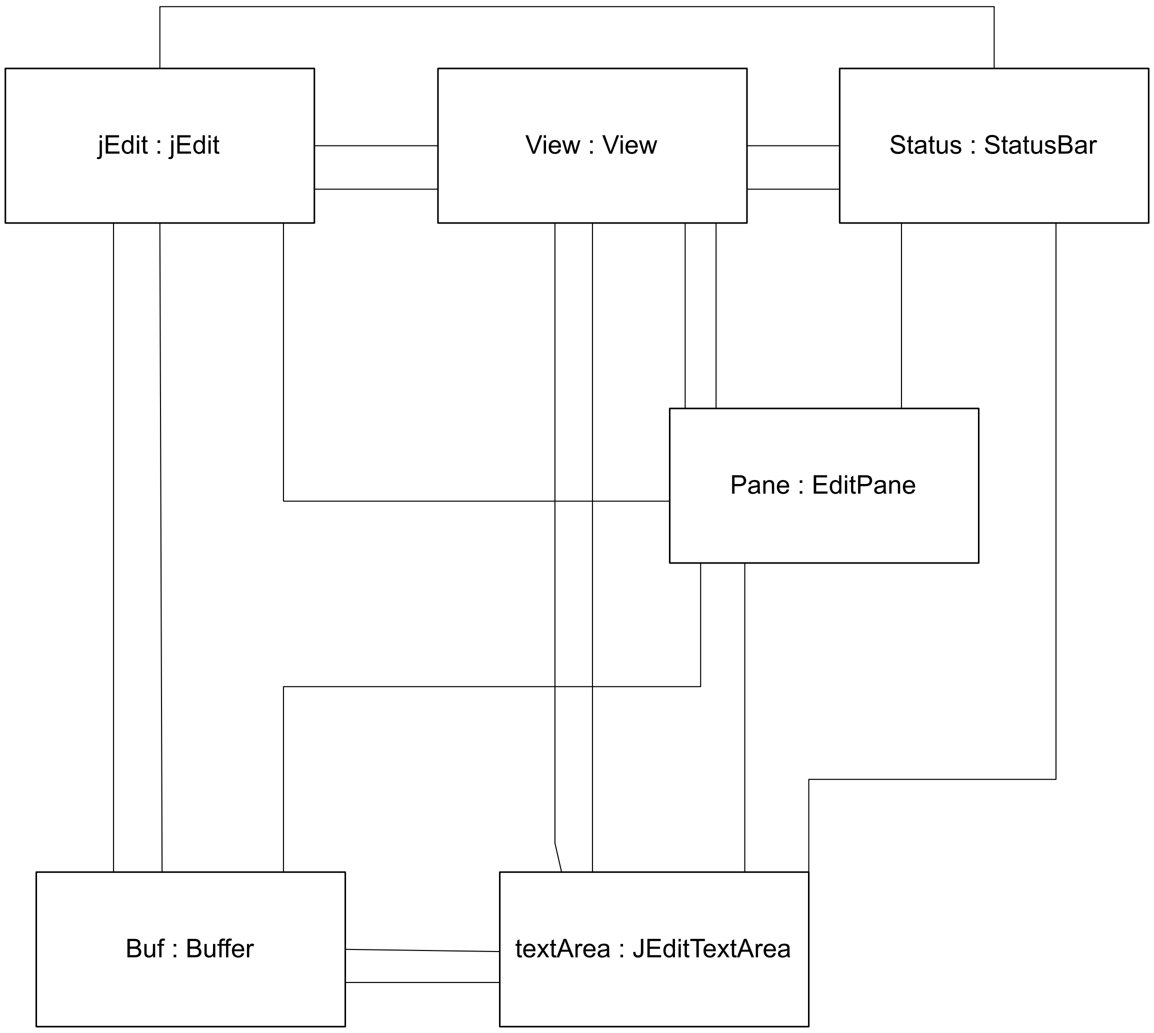
Registered components receive messages reflecting changes in the application's state, including changes in buffers, views and edit panes, changes in the set of properties maintained by the application, and the closing of the application.

StatusBar

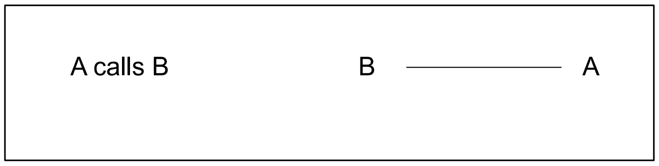
StatusBars are a small message bar at the bottom of a View that convey information about the state of the application such as the scroll position and caret position.

The next two pages give a component and connector diagram of these components split into two diagrams based on the type of communication occurring.

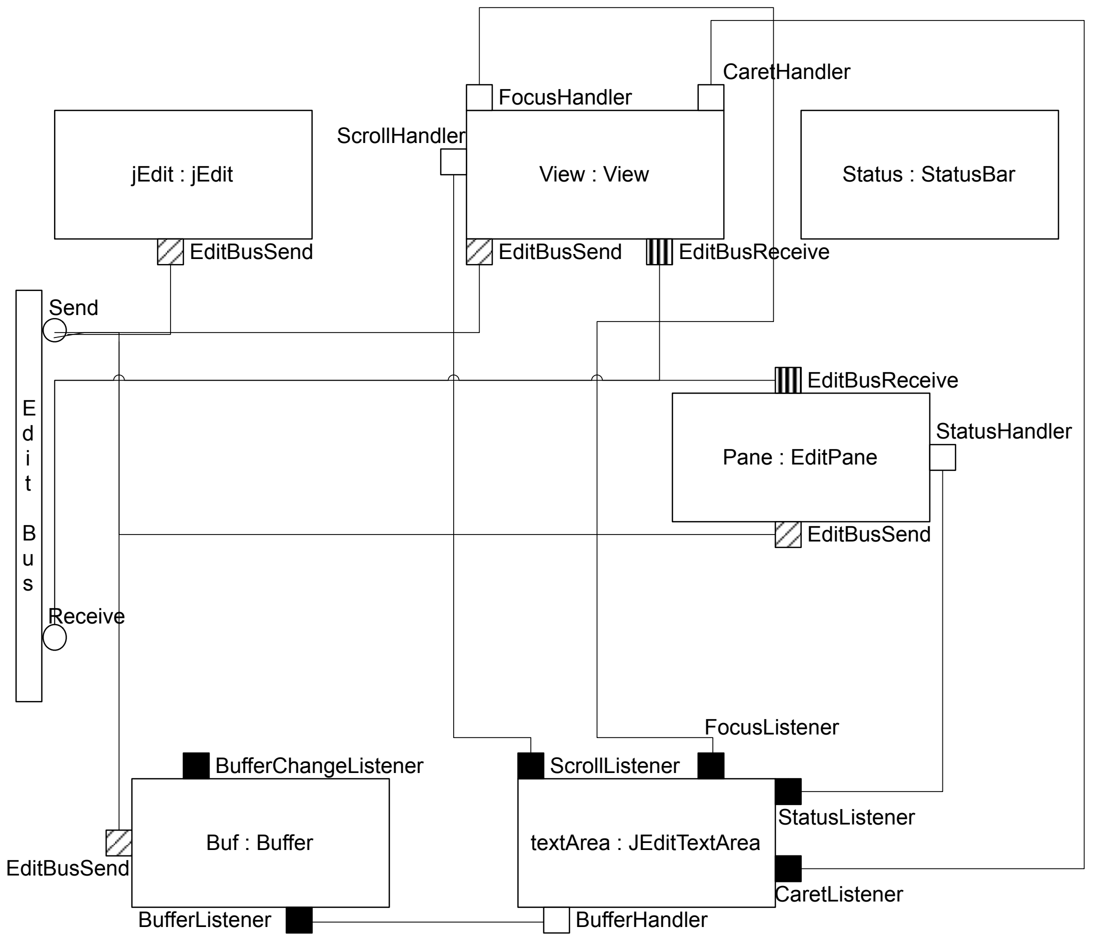
Call / Return Communication



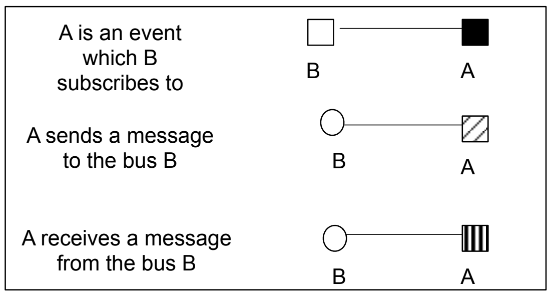
KEY



Implicit Invocation Communication



KEY


A is an event which B subscribes to
B
A
A sends a message to the bus B
B
A
A receives a message from the bus B
B
A

Try out jEdit by performing the following actions:

-Create a new file

-Type some text in the file and note that the file's icon changes.

-Press return enough times so that there is more text than can fit on the screen. Scroll the active window. Note that the left portion of the StatusBar displays two pieces of information. The leftmost piece is the line and column of the caret (cursor's) position. Next to this is a percentage describing how far down the text are is scrolled. Other messages (such as a buffer being autosaved) are displayed in the center portion of the status bar. On the right edge is information on the current character set and mode and current memory use.

-Create a second new file. Use the list box with the file's name above the text area to switch back to the first file.

-JEdit has a feature called "folds" which hierarchically hides lines of text. Type in the following text using tabs for spacing:

```
The
    Quick
        Brown
            Fox
    Jumped
        Over
    The
        Lazy
            Dog.
```

Note that JEdit displays triangles next to each of these lines. Clicking on the triangles causes JEdit to hide or show the lines of lower fold level than the line selected.

Task 1 – StatusBar caret updates

Start jEdit from the debug command, type a couple of characters of text, and create a second buffer. Place a breakpoint at the beginning of StatusBar.updateCaretStatus() at StatusBar.java:368 and switch buffers. updateCaretStatus() should be being called a number of times, resulting in the breakpoint also being hit a number of times. This is bad, at the very least from a performance perspective, because the status bar's caret message should only be changing once – from the value for the old buffer to the value for the new buffer. But it likely reflects deeper problems in the semantics of what the events that trigger these updates mean.

Your task is to investigate why this is the case and implement a better design. You should carefully budget your time to make your improved design as ideal as possible while carefully scoping your changes to what you can implement within your allotted time. As you work, you should produce a design diagram or diagrams illustrating your new design. You may use any notation you wish (e.g. UML class diagrams), including your own, but are encouraged to use a notation that captures as much of your design as possible. You may change as much or as little code as you'd like. Your goal is only to make the design as ideal as possible by the criteria of performance, understandability, and reusability. Once you've completed your task, the experimenter will ask you several questions about your design and design process including what alternatives you considered and what criteria you used to select a design alternative. You may use the file "Task 1 Notes.txt" for any notes you wish to record, and the piece of paper on the desk for any diagrams or notes you do not wish to type. You have 1.5 hours beginning now to work.

Task 2 – Fold Level Updates

Consider the following code fragment:

BufferHandler.doDelayedUpdate() : BufferHandler.java:363

```
            // force the fold levels to be
            // updated.

            // when painting the last line of
            // a buffer, Buffer.isFoldStart()
            // doesn't call getFoldLevel(),
            // hence the foldLevelChanged()
            // event might not be sent for the
            // previous line.

            buffer.getFoldLevel(delayedUpdateEnd);
```

BufferHandler is a subcomponent of JEditTextArea responsible for responding to events on the BufferListener event bus provided by Buffers. It does this by implementing the BufferListener interface and being subscribed to the bus. Unfortunately, getFoldLevel() is both architecturally questionable and clearly bad design. Architecturally, it is intended to change the buffer's state from within a different component (JEditTextArea), which may not be architecturally ideal. At a design level, it is changing the state by calling a getter method, which most developers might reasonably assume to have no effects (does not change any of the object's fields). Indeed, it is using a getter method solely to change the state of the buffer and ignoring the information the getter method is supposed to be used to obtain. This is clearly poor design.

Your task is to investigate why this is the case and implement a better design. You should carefully budget your time to make your improved design as ideal as possible while carefully scoping your changes to what you can implement within your allotted time. As you work, you should produce a design diagram or diagrams illustrating your new design. You may use any notation you wish (e.g. UML class diagrams), including your own, but are encouraged to use a notation that captures as much of your design as possible. You may change as much or as little code as you'd like. Your goal is only to make the design as ideal as possible by the criteria of performance, understandability, and reusability. Once you've completed your task, the experimenter will ask you several questions about your design and design process including what alternatives you considered and what criteria you used to select a design alternative. You may use the file "Task 2 Notes.txt" for any notes you wish to record, and the piece of paper on the desk for any diagrams or notes you do not wish to type. You have 1.5 hours beginning now to work.

APPENDIX 5: MATERIALS FROM PAPER PROTOTYPE STUDY (CHAPTER 10)

Introduction

We are currently designing a new tool for helping developers more effectively explore and understand large, complex codebases. In this study, you will help us better determine the ways in which our proposed design might or might not make these tasks easier.

You will be asked to perform one change task and will have 1 hour (beginning after your finish reading all of the instructions) to work. Your goal is to complete the task with a well-designed implementation that respects the existing architecture of the system as much as possible. The task has been designed to be challenging, so you will likely not have time to investigate or design as much as you might have with unlimited time. So do not feel discouraged if you are not able to accomplish as much as you might hope you could accomplish.

In this study, you will work with two applications – a complete, working application (ECLIPSE) and a paper prototype of a second application (REACHER). In ECLIPSE, you will be able to navigate to and edit files. But when you have questions about code (e.g., what does this do, or when does this happen?), you should first try to ask them using REACHER. But as this application does not yet exist, you will be using a paper version. To use REACHER, simply tell the experimenter what you wish to do (e.g., right click on this element), and the experimenter will attempt to simulate REACHER using paper mockups. Sometimes mockups may not exist for the question you asked – in these cases, the experimenter will tell you how to proceed.

We hope that REACHER will make coding tasks less time consuming and error prone by helping developers answer reachability questions more effectively. A reachability question is a search for target statements along possible paths through code. Current development environments make answering reachability questions difficult because developers must traverse across method calls to search for relevant statements. For example, to debug a deadlock, a developer might use a debugger or call graph tool to search from an origin statement for calls acquiring or releasing resources. Developers use their intuition or limited knowledge to decide which methods are most likely to contain relevant statements. But when targets are widely distributed, hidden behind misleading method names, or simply unexpected, this process can be time consuming or error prone. In contrast, REACHER supports directly asking and answering reachability questions. To debug a deadlock, developers can simply select an origin, search for calls that acquire or release resources, and inspect the resulting paths.

To understand how you are working, you will be asked to "think aloud" – talk about what you are thinking while you work. You should describe what you're trying to accomplish and what you're considering doing to accomplish your goals. If you find information that you've been seeking for a while, you should make sure that you say something about what you found. If you are distracted of forget to think aloud, the experimenter will prompt you to continue to think aloud by asking you about what you're doing. To allow us to analyze how you work, you will be recorded using a laptop audio recorder, a screen capture program, and a video camera. Your identity will not be revealed to anyone other than researchers in the research group. To ensure anonymity, only the researchers in the research group will have access to the audio and video recordings. However, transcripts of the audio recording with any personally identifying information removed and the screen recording may be used in public presentation of this work.

At the conclusion of the study, the experimenter will ask you some questions about your understanding of the system, how you were working, and decisions you made while working.

jEdit Background

In this study, you will be working with jEdit. jEdit is an open source text editor mainly intended for editing code written in a little over 50,000 LOC of Java. It was originally written by mostly one developer (Slava Pestov) but has more recently been edited by several other developers. Start jEdit by clicking the debug button on the ECLIPSE toolbar (the fourth icon from the left).

Several important classes in jEdit can be considered components:

jEdit

jEdit is the main class in the application and contains the application entry point. It is mostly responsible for orchestrating interactions between other components such as opening and closing buffers, creating and switching views, managing plugins, and setting and retrieving global application properties.

View

The top level window for a jEdit application toolbars, user input, creating and deleting EditPanes, and visual look and feel. There can be more than one open view for a particular application – to create a new View simply select View.New View on the menu bar.

EditPane

The containing visual element for a JEditTextArea text editing control. There may be more than one when a view is split into multiple panes. EditPanes are responsible for switching buffers and managing markers (bookmarks).

JEditTextArea

jEdit's text control for editing text. Responsibilities include scrolling, selection, painting, buffer access, caret position, text input, and text transformations.

JEditBuffer

jEdit's representation of a file containing functionality for manipulating the text in a buffer.

Buffer (inherits from JEditBuffer)

jEdit's representation of a file containing rules about how the text file in memory is loaded and stored to disk.

EditBus

Registered components receive messages reflecting changes in the application's state, including changes in buffers, views and edit panes, changes in the set of properties maintained by the application, and the closing of the application.

StatusBar

StatusBars are a small message bar at the bottom of a View that convey information about the state of the application such as the scroll position and caret position.

The next two pages give a component and connector diagram of these components split into two diagrams based on the type of communication occurring.

Try out jEdit by performing the following actions:

-Create a new file

-Type some text in the file and note that the file's icon changes.

-Press return enough times so that there is more text than can fit on the screen. Scroll the active window. Note that the left portion of the StatusBar displays two pieces of information. The leftmost piece is the line and column of the caret (cursor's) position. Next to this is a percentage describing how far down the text are is scrolled. Other messages (such as a buffer being autosaved) are displayed in the center portion of the status bar. On the right edge is information on the current character set and mode and current memory use.

-Create a second new file. Use the list box with the file's name above the text area to switch back to the first file.

-JEdit has a feature called "folds" which hierarchically hides lines of text. Type in the following text using tabs for spacing:

```
The
        Quick
                Brown
                        Fox
        Jumped
                Over
        The
                Lazy
                        Dog.
```

Note that JEdit displays triangles next to each of these lines. Clicking on the triangles causes JEdit to hide or show the lines of lower fold level than the line selected.

ECLIPSE Code Navigation Tutorial

ECLIPSE features a number of sophisticated features for navigating through source code. In this brief tutorial, you will try out several of the most useful features.

1. Press CONTROL-SHIFT-T and type "jedit" to open up the jedit class.
2. Press CONTROL-O to open the method outline and type getProperties to navigate to the method getProperties().
3. Hold down control and click on getProperties() (the method call on propMgr, not the method declaration). This moves you to the method declaration.
4. Go back to the previous method you were looking at by clicking on the left arrow secondmost from the right edge of the toolbar.
5. Place the cursor over the getProperties() method declaration, right click, and select "Open Call Hierarchy". At the bottom of the screen, a tree view shows either the Callee Hierarchy or the Caller Hierarchy. Switch between them by clicking a button to the right of the Call Hiearchy tab.
6. Place the cursor on propMgr. All references to this field in the current Java file are now highlighted with a grey highlight both in the text editor and next to the scrollbar, allowing you to quickly find all of the references in the file.
7. Find all of the methods that assign the field propMgr by placing the cursor on top of it and selecting from the "Search" menu at the top of the screen "Write Access" and then "Project".
8. Find all references to the jEdit type by selecting Java from the Search menus, typing in jEdit, and selecting "Type" in Search For.

REACHER Tutorial

In this tutorial, you'll learn how to use REACHER by following along with another developer (Brad) as he attempts to complete a similar task in jEdit to the one you will shortly work on.

Brad has received a bug report that there is a potential performance problem with redrawing a portion of the jEdit status bar. Whenever the file currently open in jEdit is changed, the display of the caret's position in the status bar is redrawn 7 times when it only needs to be redrawn once. While somewhat trivial by itself, the bug asks if this is a symptom of a larger problem. Thus, Brad decides to investigate how this code works to see if a better design is possible.

He first navigates to the method StatusBar.updateCaretStatus(), which he knows from past experience is the method that does the redraw in question. Next, he right clicks the method, and selects "When does this happen?" from the context menu, launching REACHER with a new upstream reachability question. REACHER supports two types of reachability questions – upstream and downstream. Upstream reachability questions are a search across paths that reach a destination statement. A downstream reachability question is a search across paths beginning at an origin statement.

[T0]

A brief tour of the main interface elements of REACHER:

TraceMaps – visual depictions of paths through code

Navigation controls – works like a browser back and forwards button to quickly return to previous view states

Search box – search for statements, comments, and methods by name along upstream or downstream paths

Search results list – a list of statements matching the active search

Search cursor – all searches are relative to a start method. Dragging the start cursor changes where the search starts from.

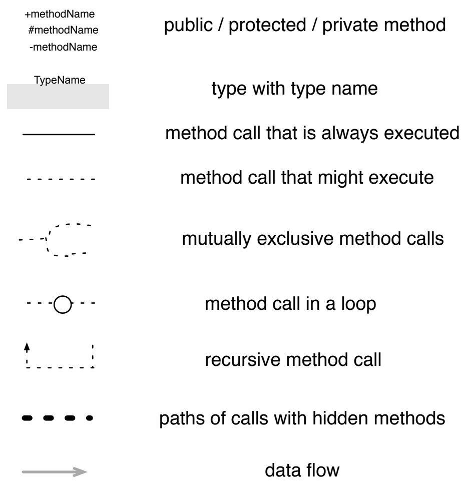
Question list – REACHER supports investigating multiple reachability questions simultaneously. Each numbered reachability question corresponds to the number in the TraceMap. Reachability questions may be closed (x button) or hidden (dot button).

TraceSource – clicking on a method in the TraceMap highlights the method with a yellow outline and navigates the TraceSource window. Eclipse must currently be navigated manually by pressing CONTROL-SHIFT-T to open the type and then CONTROL-SHIFT-O to open the method.

Brad wonders – what are these callers of updateCaretStatus. To answer this question, he slides the show depth slider to 1. For the upstream question he is currently exploring, this shows methods one call up from updateCaretStatus.

[T1]

A TraceMap depicts paths through code as a tree of methods. Time flows from left to right and top to bottom – the in-order traversal of the tree corresponds to execution order. Visual attributes of the diagram depict information about these paths:


+methodName
#methodName
-methodName
public / protected / private method
TypeName
type with type name
method call that is always executed
method call that might execute
mutually exclusive method calls
method call in a loop
recursive method call
paths of calls with hidden methods
data flow

Brad next wonders – are all these callers really on paths from a buffer switch? How many of these paths occur on each buffer switch? To answer this question, Brad enters “buffer switch” in the search box, sees one result – BufferSwitch.ActionHandler.actionPerformed() – and clicks on it. This denotes the method as a target. Now, only those paths containing it are shown. And a separate copy of updateCaretStatus is shown each time it is invoked.

[T2]

Brad notices that 4 of the paths from the buffer switch to updateCaretStatus come through a scrolledVertically() method. What could scrolling possibly have to do with updating the caret status information on the Status bar?

To investigate this question, Brad better wants to understand what updateCaretStatus is doing. But to answer such questions, the active question must be switched from an upstream question to a downstream question. Brad right clicks actionPerformed and selects set as origin. Now, REACHER is configured to follow paths downstream from actionPerformed rather than upstream from updateCaretStatus. Next, Brad clicks and drags the search cursor from actionPerformed to updateCaretStatus. This focuses the region to be searched to downstream from updateCaretStatus. Finally, Brad enters “scroll” in the search text box.

[T4]

Surprisingly, two lines in updateCaretStatus match – a comment and a statement getting scroll information. And, further downstream, is a method accessing the scrollLine. Curious, Brad clicks on line 418 in the results list to select it the trace source (and add it the to trace map).

[T4.1]

Inspecting the trace source, Brad sees that the scroll position is being used to influence the text being written to the status bar.

Abandoning this line of questioning, Brad hits back twice to clear the “scroll” search. He wonders, are there other interesting patterns in the paths by which this update is happening. To answer this question, Brad drags a box around the dashed edges to the right of actionPerfomed, selecting all of them. Invoking a context menu on the set of items, Brad selects the “Show paths” command, expanding the methods along the paths currently hidden.

[T5]

Many of the paths go through finishCaretUpdate and #_finishCaretUpdate. These methods sound like something that might be determining what should happen when the caret position changes – maybe these might have some role in controlling when updateCaretStatus is called? Brad clicks on the first one to investigate.

[T6]

There is already a guard on the _finishCaretUpdate() call – something about transactions. What is this related to? Brad double clicks isTransactionComplete() to navigate to the method.

[T6.2]

There seem to be three different mechanisms already in place to stop these updates. What are they about? Undo doesn’t seem relevant, and a buffer switch isn’t an edit, so that doesn’t seem relevant. What about transactions? What's a transaction, and how's it being used? Brad right clicks on the field, and selects "Where might its value come from?"

[T6.3]

This creates a followup question – a new upstream question (2) with it's own (set of) destination statements (writes to JEDitBuffer.transaction). It looks like transactions are being used to denote some sort of complex edit occurring when content is added or removed from the buffer. So this still isn't relevant.

But maybe this is a good place to add a new flag? Brad hits back twice, opens up JEditBuffer.finishCaretUpdate() in Eclipse, and creates a new flag that should prevent all but the last updateCaretStatus call through caretUpdate. He switches back to REACHER, and REACHER asks – "Do not update, update, or update and compare". Brad wants to see what changed, so he selects update and compare, creating new reachability questions for the updated code and a comparison against the original code.

[T7]

In comparisons, REACHER shows call edges in separate colors corresponding to each condition. In this case, asparagus corresponds to the original version and magenta to the changed version. It seems that the change did indeed prevent calls to updateCaretStatus through finishCaretUpdate. Looking at the TraceSource, Brad sees that his new flag is false, causing _finishCaretUpdate() not to be executed.

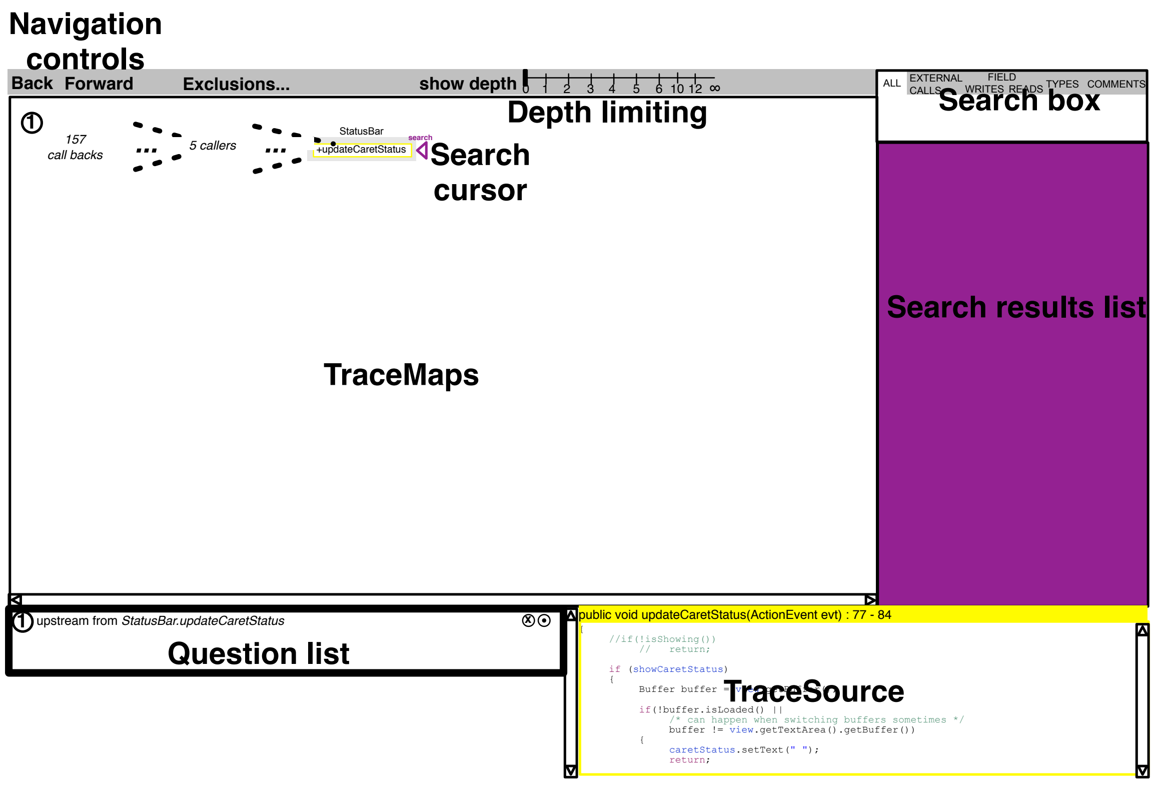
However, all of the calls, including the last one, are gone. Why isn't the last call still there? Brad opens the last copy of finishCaretUpdate to see why the call path is not being taken.

END OF TUTORIAL

If you have any more questions about REACHER, feel free to ask the experimenter at any point during the study!

Screenshots of Omnigraffle diagrams used in the tutorial

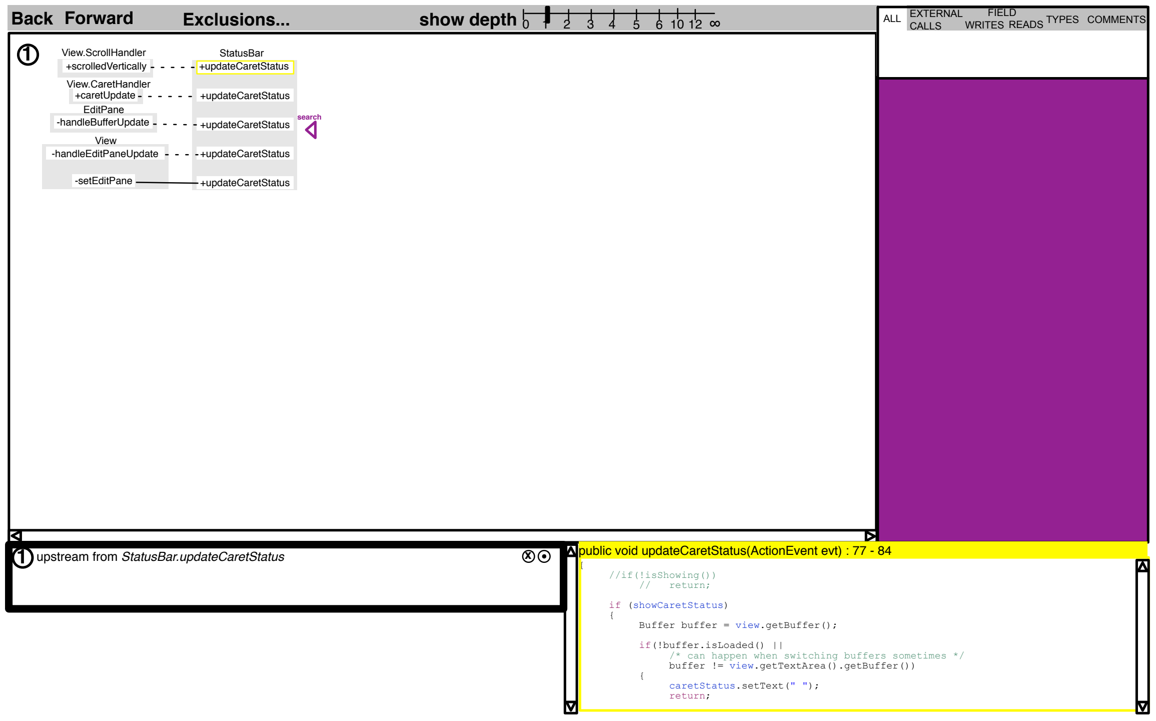
[T0]



public void updateCaretStatus(ActionEvent evt) : 77 - 84

```
{
    //if(!isShowing())
        //   return;

    if (showCaretStatus)
    {
        Buffer buffer =

        if(!buffer.isLoaded() ||
            /* can happen when switching buffers sometimes */
            buffer != view.getTextArea().getBuffer())
        {
            caretStatus.setText(" ");
            return;
```

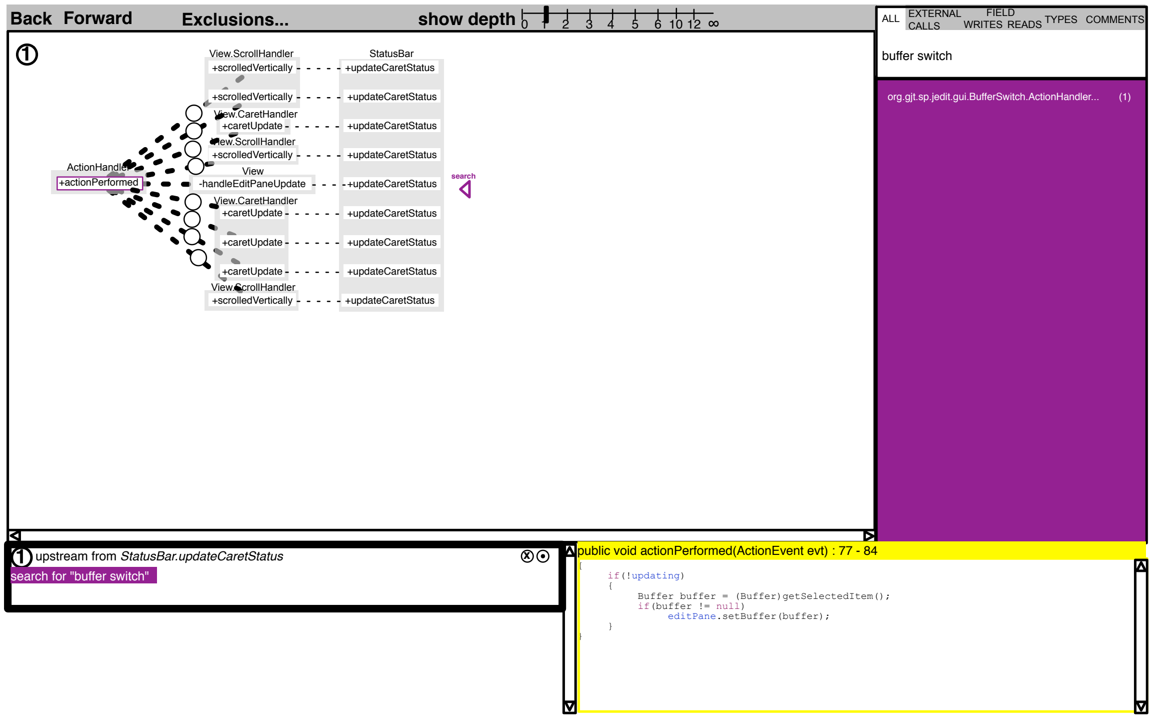
[T1]



public void updateCaretStatus(ActionEvent evt) : 77 - 84

```
{
    //if(!isShowing())
        //   return;

    if (showCaretStatus)
    {
        Buffer buffer = view.getBuffer();

        if(!buffer.isLoaded() ||
            /* can happen when switching buffers sometimes */
            buffer != view.getTextArea().getBuffer())
        {
            caretStatus.setText(" ");
            return;
```

[T2]



buffer switch

org.gjt.sp.jedit.gui.BufferSwitch.ActionHandler... (1)

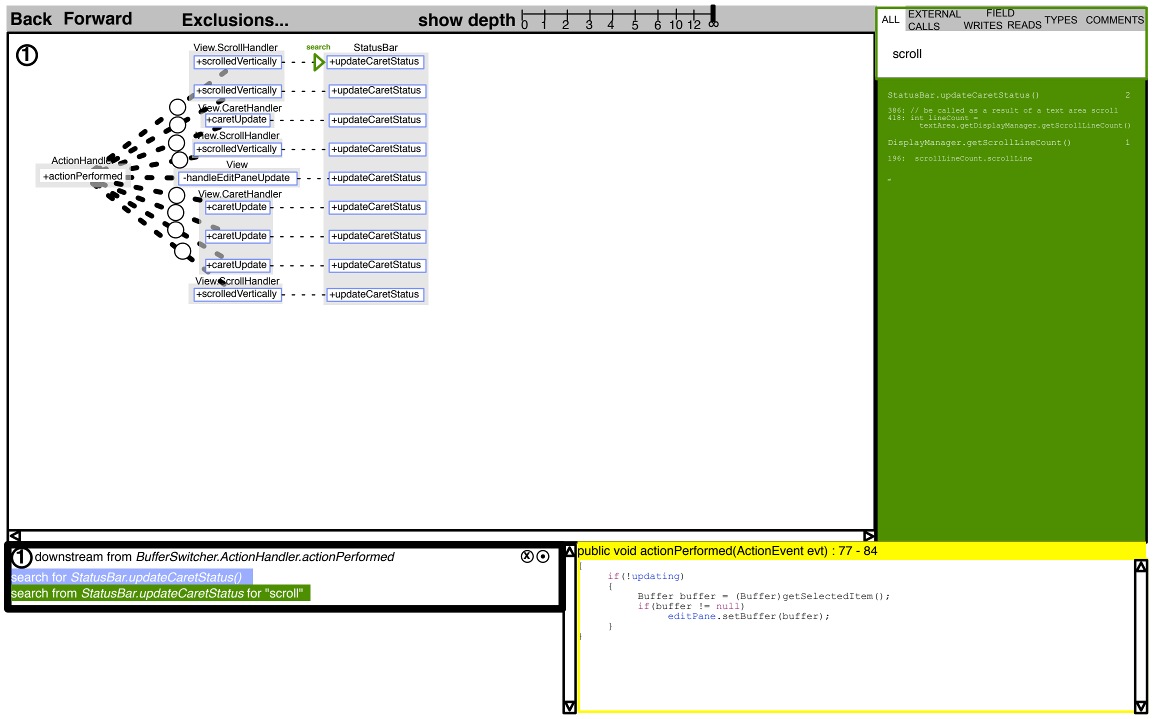
① upstream from *StatusBar.updateCaretStatus*

search for "buffer switch"

public void actionPerformed(ActionEvent evt) : 77 - 84

```
{
     if(!updating)
     {
          Buffer buffer = (Buffer)getSelectedItem();
          if(buffer != null)
               editPane.setBuffer(buffer);
     }
}
```

[T4]



scroll

```
StatusBar.updateCaretStatus()                2

386: // be called as a result of a text area scroll
418: int lineCount =
        textArea.getDisplayManager.getScrollLineCount()

DisplayManager.getScrollLineCount()          1

196:  scrollLineCount.scrollLine
```

① downstream from *BufferSwitcher.ActionHandler.actionPerformed*

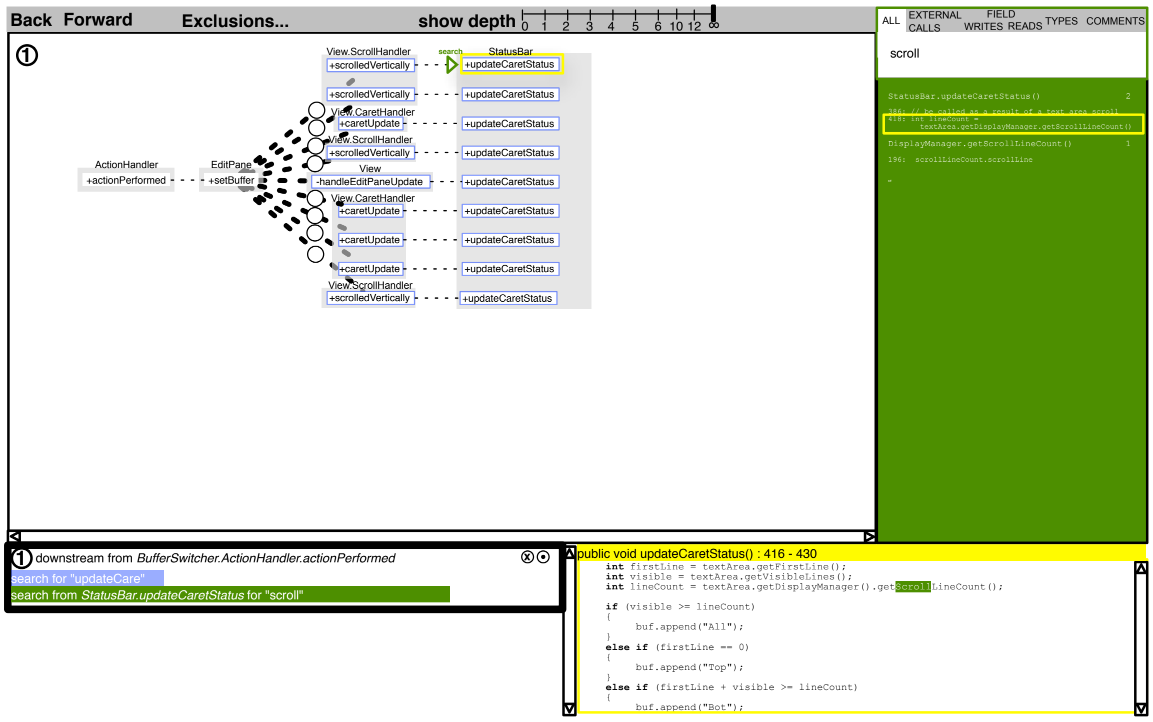
search for *StatusBar.updateCaretStatus()*

search from *StatusBar.updateCaretStatus* for "scroll"

public void actionPerformed(ActionEvent evt) : 77 - 84

```
{
     if(!updating)
     {
          Buffer buffer = (Buffer)getSelectedItem();
          if(buffer != null)
               editPane.setBuffer(buffer);
     }
}
```

[T4.1]



scroll

```
StatusBar.updateCaretStatus()                        2
386: // be called as a result of a text area scroll
418: int lineCount =
       textArea.getDisplayManager.getScrollLineCount()
DisplayManager.getScrollLineCount()                  1
196:  scrollLineCount.scrollLine
```

① downstream from *BufferSwitcher.ActionHandler.actionPerformed*

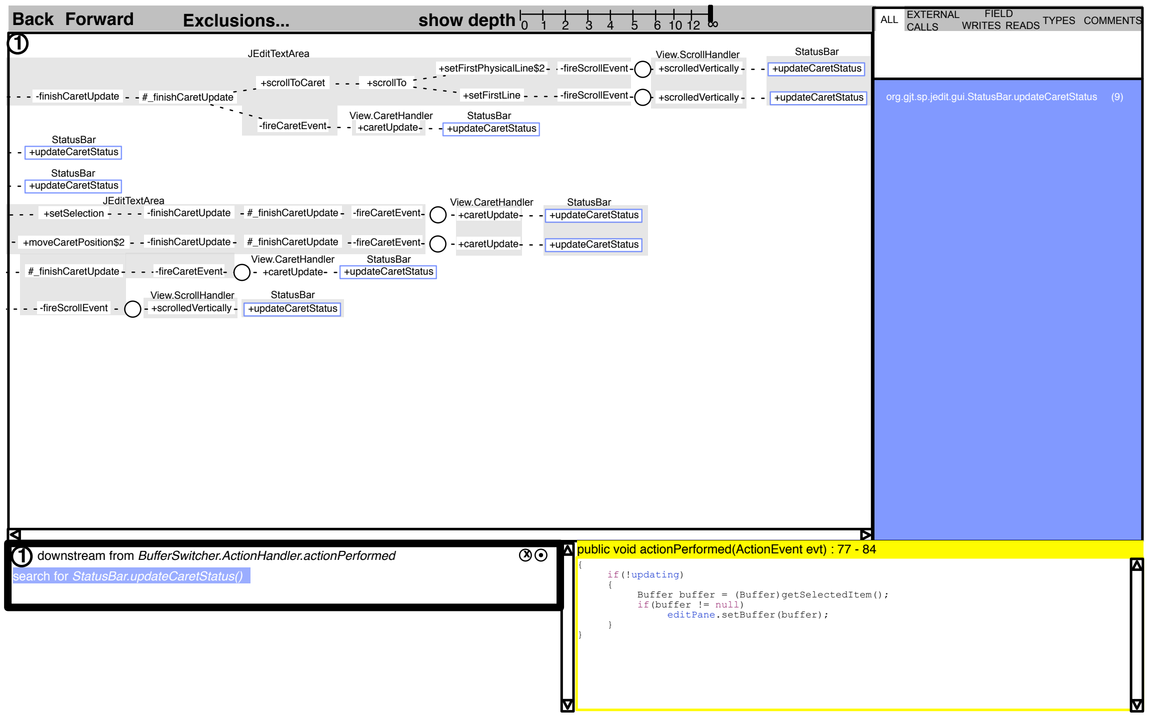
search for "updateCare"

search from *StatusBar.updateCaretStatus* for "scroll"

public void updateCaretStatus() : 416 - 430

```
    int firstLine = textArea.getFirstLine();
    int visible = textArea.getVisibleLines();
    int lineCount = textArea.getDisplayManager().getScrollLineCount();

    if (visible >= lineCount)
    {
         buf.append("All");
    }
    else if (firstLine == 0)
    {
         buf.append("Top");
    }
    else if (firstLine + visible >= lineCount)
    {
         buf.append("Bot");
```

[T5]



org.gjt.sp.jedit.gui.StatusBar.updateCaretStatus (9)

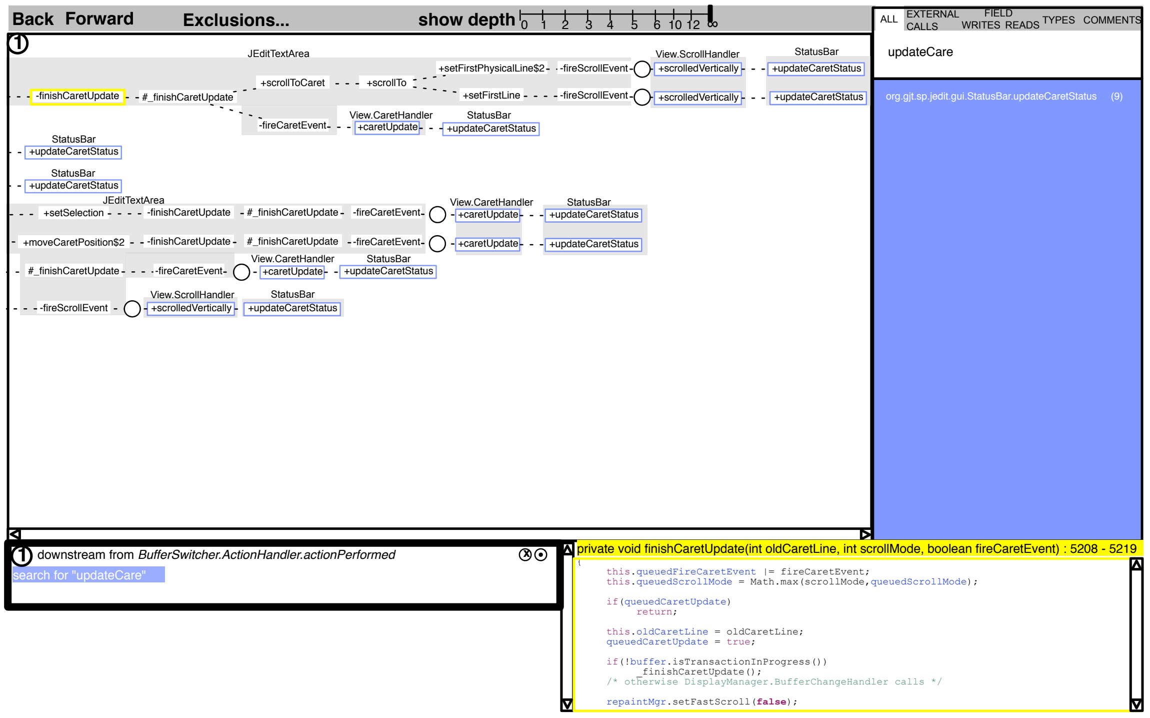
① downstream from *BufferSwitcher.ActionHandler.actionPerformed*

search for *StatusBar.updateCaretStatus()*

public void actionPerformed(ActionEvent evt) : 77 - 84

```
{
     if(!updating)
     {
          Buffer buffer = (Buffer)getSelectedItem();
          if(buffer != null)
               editPane.setBuffer(buffer);
     }
}
```

[T6]



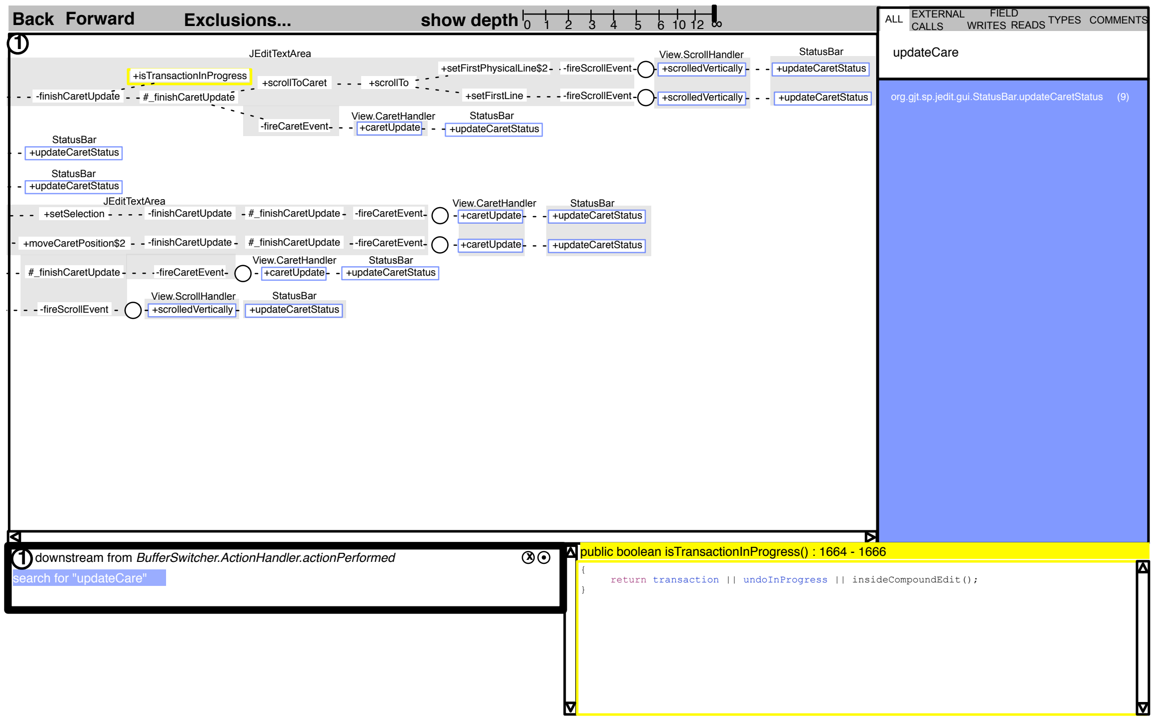
search for "updateCare"

org.gjt.sp.jedit.gui.StatusBar.updateCaretStatus (9)

private void finishCaretUpdate(int oldCaretLine, int scrollMode, boolean fireCaretEvent) : 5208 - 5219

```
{
	this.queuedFireCaretEvent |= fireCaretEvent;
	this.queuedScrollMode = Math.max(scrollMode,queuedScrollMode);

	if(queuedCaretUpdate)
		return;

	this.oldCaretLine = oldCaretLine;
	queuedCaretUpdate = true;

	if(!buffer.isTransactionInProgress())
		_finishCaretUpdate();
	/* otherwise DisplayManager.BufferChangeHandler calls */

	repaintMgr.setFastScroll(false);
```

[T6.2]



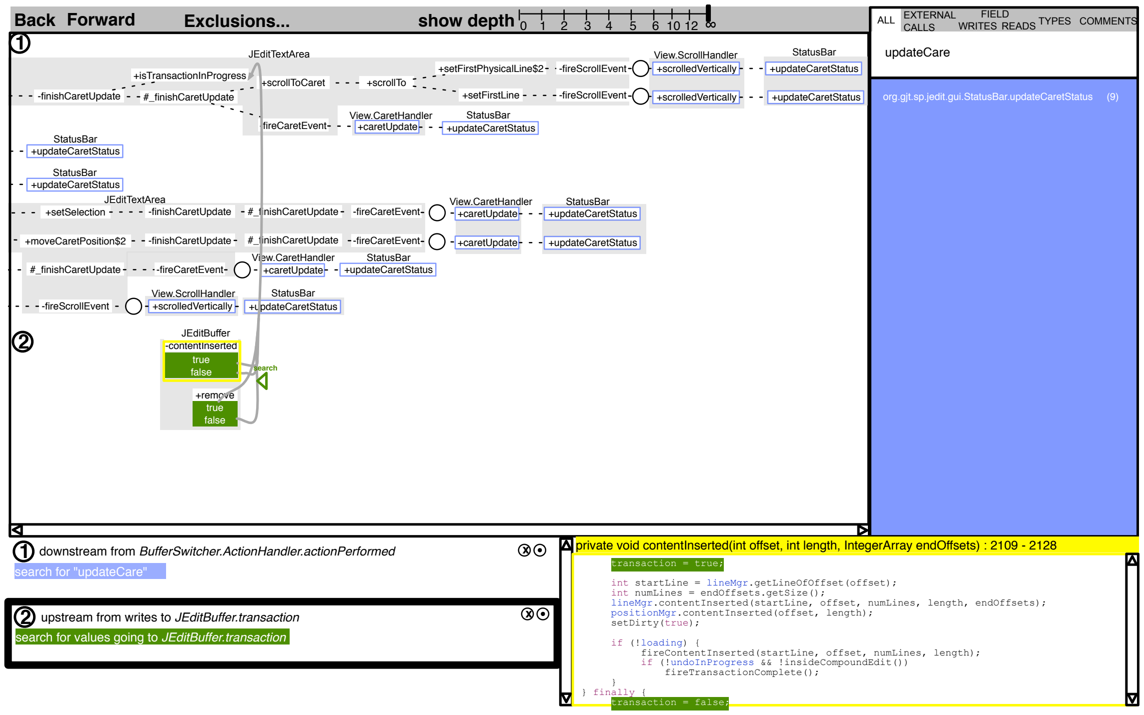
search for "updateCare"

org.gjt.sp.jedit.gui.StatusBar.updateCaretStatus (9)

public boolean isTransactionInProgress() : 1664 - 1666

```
{
	return transaction || undoInProgress || insideCompoundEdit();
}
```

[T6.3]



1 downstream from *BufferSwitcher.ActionHandler.actionPerformed*

search for "updateCare"

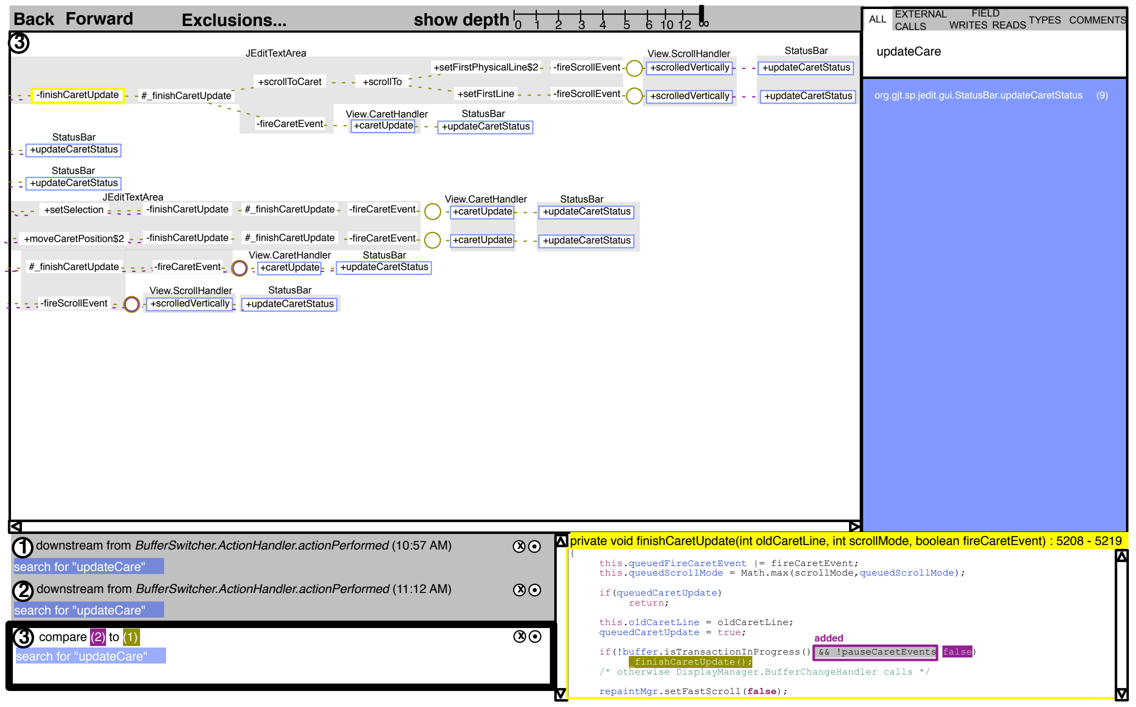
2 upstream from writes to *JEditBuffer.transaction*

search for values going to *JEditBuffer.transaction*

private void contentInserted(int offset, int length, IntegerArray endOffsets) : 2109 - 2128

```
        transaction = true;

        int startLine = lineMgr.getLineOfOffset(offset);
        int numLines = endOffsets.getSize();
        lineMgr.contentInserted(startLine, offset, numLines, length, endOffsets);
        positionMgr.contentInserted(offset, length);
        setDirty(true);

        if (!loading) {
            fireContentInserted(startLine, offset, numLines, length);
            if (!undoInProgress && !insideCompoundEdit())
                fireTransactionComplete();
        }
    } finally {
        transaction = false;
```

[T7]



1 downstream from *BufferSwitcher.ActionHandler.actionPerformed* (10:57 AM)

search for "updateCare"

2 downstream from *BufferSwitcher.ActionHandler.actionPerformed* (11:12 AM)

search for "updateCare"

3 compare (2) to (1)

search for "updateCare"

private void finishCaretUpdate(int oldCaretLine, int scrollMode, boolean fireCaretEvent) : 5208 - 5219

```
{
    this.queuedFireCaretEvent |= fireCaretEvent;
    this.queuedScrollMode = Math.max(scrollMode,queuedScrollMode);

    if(queuedCaretUpdate)
        return;

    this.oldCaretLine = oldCaretLine;
    queuedCaretUpdate = true;
                                              added
    if(!buffer.isTransactionInProgress() && !pauseCaretEvents false)
        _finishCaretUpdate();
    /* otherwise DisplayManager.BufferChangeHandler calls */

    repaintMgr.setFastScroll(false);
```

Task – Fold Level Updates

Consider the following code fragment:

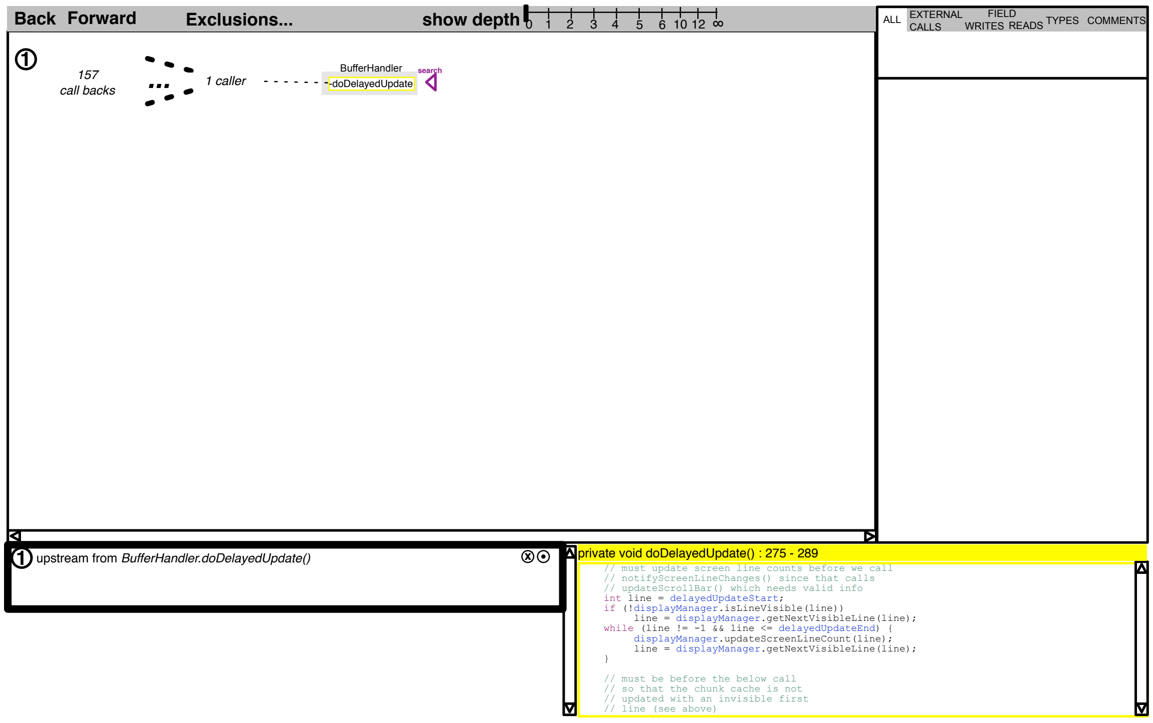
BufferHandler.doDelayedUpdate() : BufferHandler.java:363

```
// force the fold levels to be
// updated.

// when painting the last line of
// a buffer, Buffer.isFoldStart()
// doesn't call getFoldLevel(),
// hence the foldLevelChanged()
// event might not be sent for the
// previous line.

buffer.getFoldLevel(delayedUpdateEnd);
```

BufferHandler is a subcomponent of JEditTextArea responsible for responding to events on the BufferListener event bus provided by Buffers. It does this by implementing the BufferListener interface and being subscribed to the bus. Unfortunately, getFoldLevel() is both architecturally questionable and clearly bad design. Architecturally, it is intended to change the buffer's state from within a different component (JEditTextArea), which may not be architecturally ideal. At a design level, it is changing the state by calling a getter method, which most developers might reasonably assume to have no effects (does not change any of the object's fields). Indeed, it is using a getter method solely to change the state of the buffer and ignoring the information the getter method is supposed to be used to obtain. This is clearly poor design.

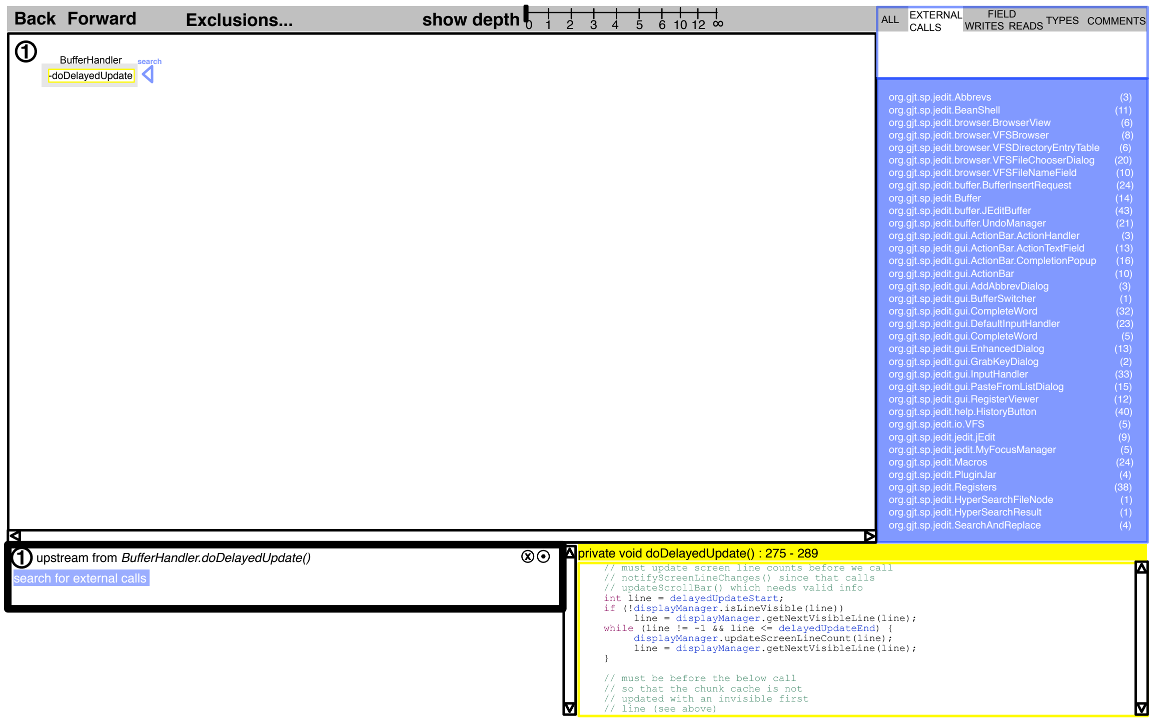
Your task is to investigate why this is the case and implement a better design. You should carefully budget your time to make your improved design as ideal as possible while carefully scoping your changes to what you can implement within your allotted time. As you work, you should produce a design diagram or diagrams illustrating your new design. You may use any notation you wish (e.g. UML class diagrams), including your own, but are encouraged to use a notation that captures as much of your design as possible. You may change as much or as little code as you'd like. Your goal is only to make the design as ideal as possible by the criteria of performance, understandability, and reusability. Once you've completed your task, the experimenter will ask you several questions about your design and design process including what alternatives you considered and what criteria you used to select a design alternative. You may use the file "Task 2 Notes.txt" for any notes you wish to record, and the piece of paper on the desk for any diagrams or notes you do not wish to type. You have 1 hour beginning now to work.

Screenshots of Omnigraffle diagrams used in the Fold Level Updates task



private void doDelayedUpdate() : 275 - 289

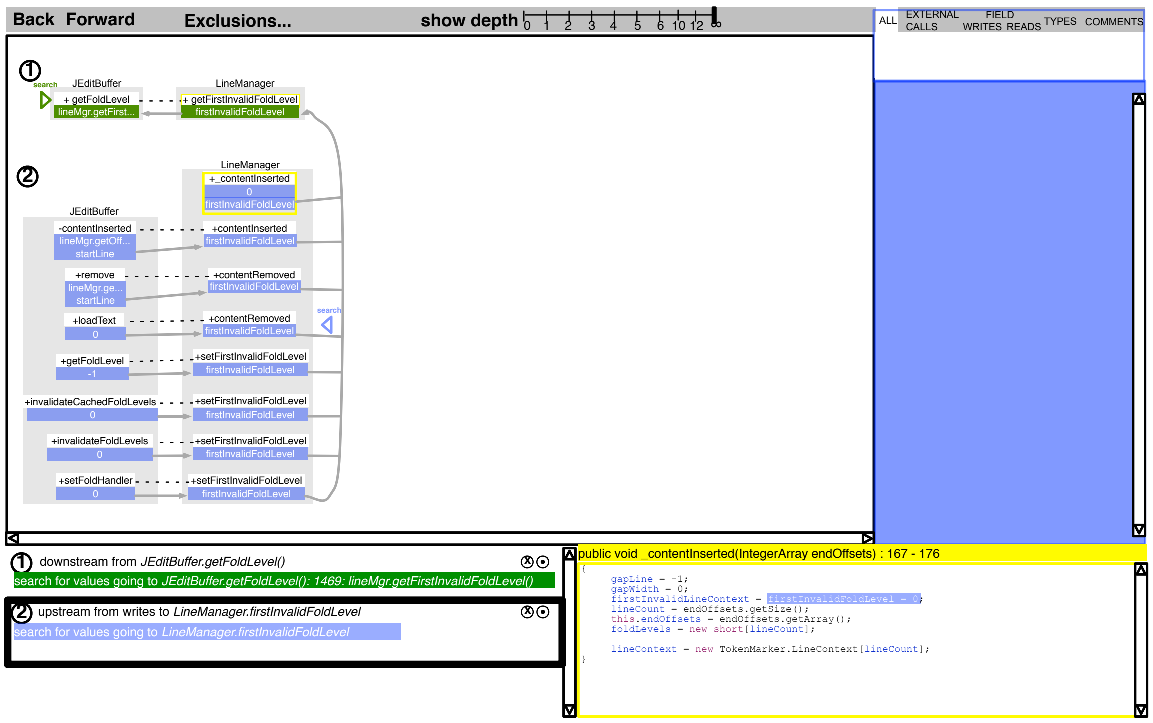
```
    // must update screen line counts before we call
    // notifyScreenLineChanges() since that calls
    // updateScrollBar() which needs valid info
    int line = delayedUpdateStart;
    if (!displayManager.isLineVisible(line))
         line = displayManager.getNextVisibleLine(line);
    while (line != -1 && line <= delayedUpdateEnd) {
         displayManager.updateScreenLineCount(line);
         line = displayManager.getNextVisibleLine(line);
    }

    // must be before the below call
    // so that the chunk cache is not
    // updated with an invisible first
    // line (see above)
```



org.gjt.sp.jedit.Abbrevs	(3)
org.gjt.sp.jedit.BeanShell	(11)
org.gjt.sp.jedit.browser.BrowserView	(6)
org.gjt.sp.jedit.browser.VFSBrowser	(8)
org.gjt.sp.jedit.browser.VFSDirectoryEntryTable	(6)
org.gjt.sp.jedit.browser.VFSFileChooserDialog	(20)
org.gjt.sp.jedit.browser.VFSFileNameField	(10)
org.gjt.sp.jedit.buffer.BufferInsertRequest	(24)
org.gjt.sp.jedit.Buffer	(14)
org.gjt.sp.jedit.buffer.JEditBuffer	(43)
org.gjt.sp.jedit.buffer.UndoManager	(21)
org.gjt.sp.jedit.gui.ActionBar.ActionHandler	(3)
org.gjt.sp.jedit.gui.ActionBar.ActionTextField	(13)
org.gjt.sp.jedit.gui.ActionBar.CompletionPopup	(16)
org.gjt.sp.jedit.gui.ActionBar	(10)
org.gjt.sp.jedit.gui.AddAbbrevDialog	(3)
org.gjt.sp.jedit.gui.BufferSwitcher	(1)
org.gjt.sp.jedit.gui.CompleteWord	(32)
org.gjt.sp.jedit.gui.DefaultInputHandler	(23)
org.gjt.sp.jedit.gui.CompleteWord	(5)
org.gjt.sp.jedit.gui.EnhancedDialog	(13)
org.gjt.sp.jedit.gui.GrabKeyDialog	(2)
org.gjt.sp.jedit.gui.InputHandler	(33)
org.gjt.sp.jedit.gui.PasteFromListDialog	(15)
org.gjt.sp.jedit.gui.RegisterViewer	(12)
org.gjt.sp.jedit.help.HistoryButton	(40)
org.gjt.sp.jedit.io.VFS	(5)
org.gjt.sp.jedit.jedit.jEdit	(9)
org.gjt.sp.jedit.jedit.MyFocusManager	(5)
org.gjt.sp.jedit.Macros	(24)
org.gjt.sp.jedit.PluginJar	(4)
org.gjt.sp.jedit.Registers	(38)
org.gjt.sp.jedit.HyperSearchFileNode	(1)
org.gjt.sp.jedit.HyperSearchResult	(1)
org.gjt.sp.jedit.SearchAndReplace	(4)

private void doDelayedUpdate() : 275 - 289

```
    // must update screen line counts before we call
    // notifyScreenLineChanges() since that calls
    // updateScrollBar() which needs valid info
    int line = delayedUpdateStart;
    if (!displayManager.isLineVisible(line))
         line = displayManager.getNextVisibleLine(line);
    while (line != -1 && line <= delayedUpdateEnd) {
         displayManager.updateScreenLineCount(line);
         line = displayManager.getNextVisibleLine(line);
    }

    // must be before the below call
    // so that the chunk cache is not
    // updated with an invisible first
    // line (see above)
```



① downstream from *JEditBuffer.getFoldLevel()*

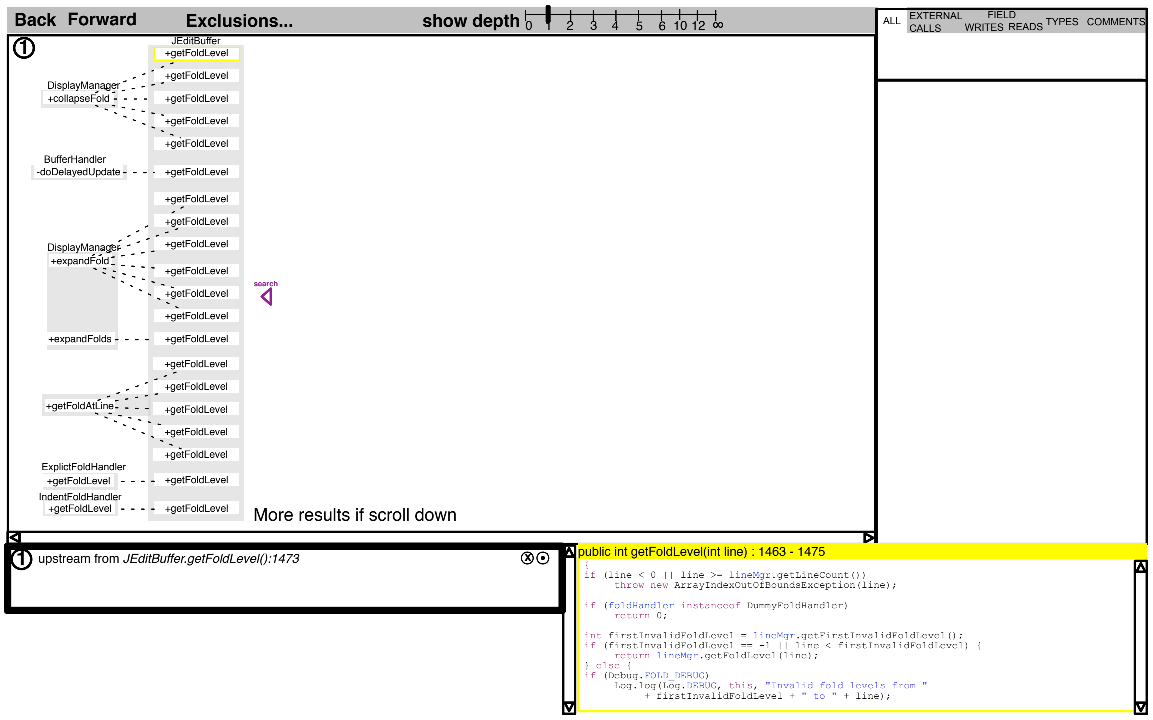
search for values going to *JEditBuffer.getFoldLevel(): 1469: lineMgr.getFirstInvalidFoldLevel()*

② upstream from writes to *LineManager.firstInvalidFoldLevel*

search for values going to *LineManager.firstInvalidFoldLevel*

public void _contentInserted(IntegerArray endOffsets) : 167 - 176

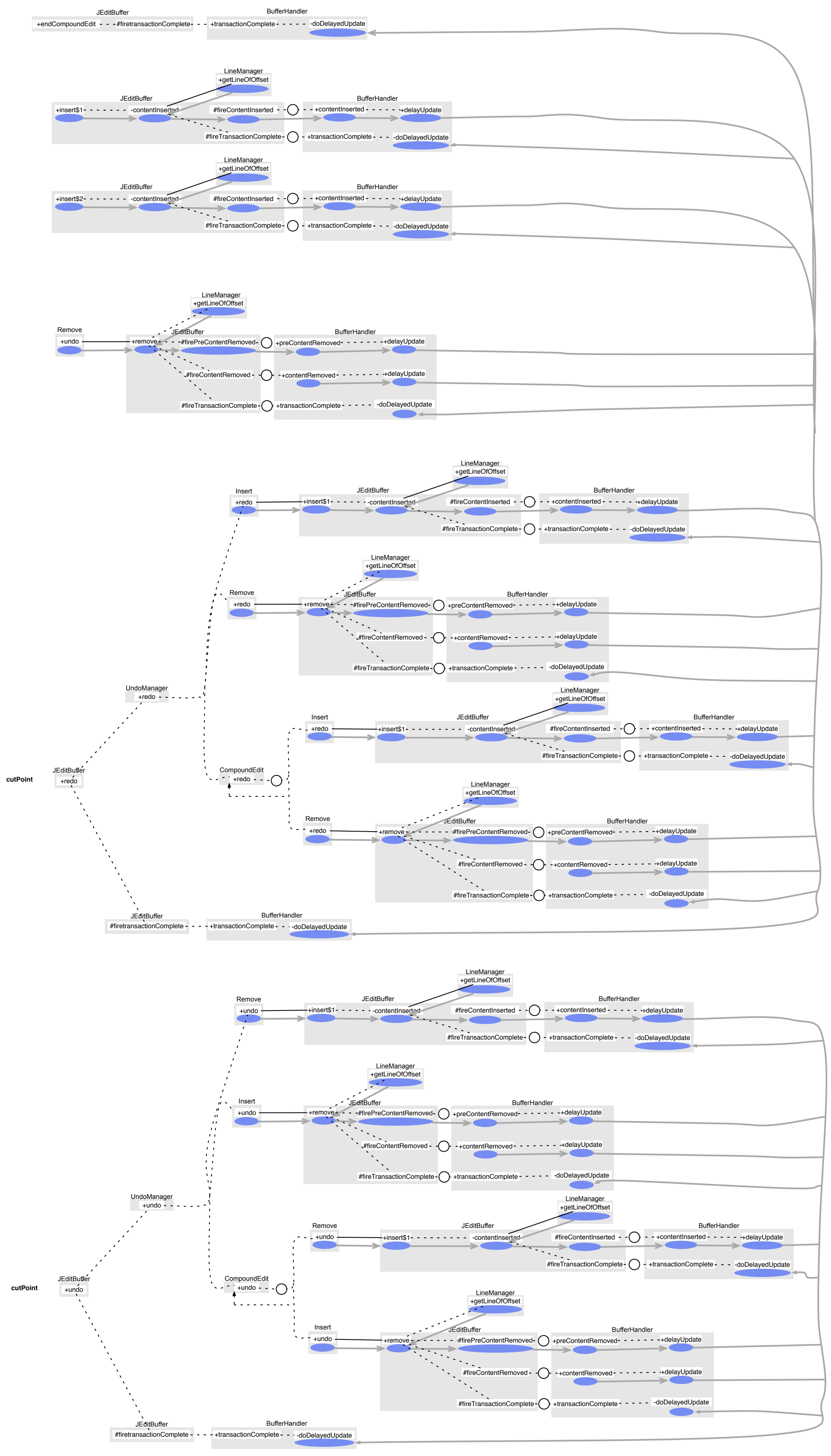
```
{
     gapLine = -1;
     gapWidth = 0;
     firstInvalidLineContext = firstInvalidFoldLevel = 0;
     lineCount = endOffsets.getSize();
     this.endOffsets = endOffsets.getArray();
     foldLevels = new short[lineCount];

     lineContext = new TokenMarker.LineContext[lineCount];
}
```

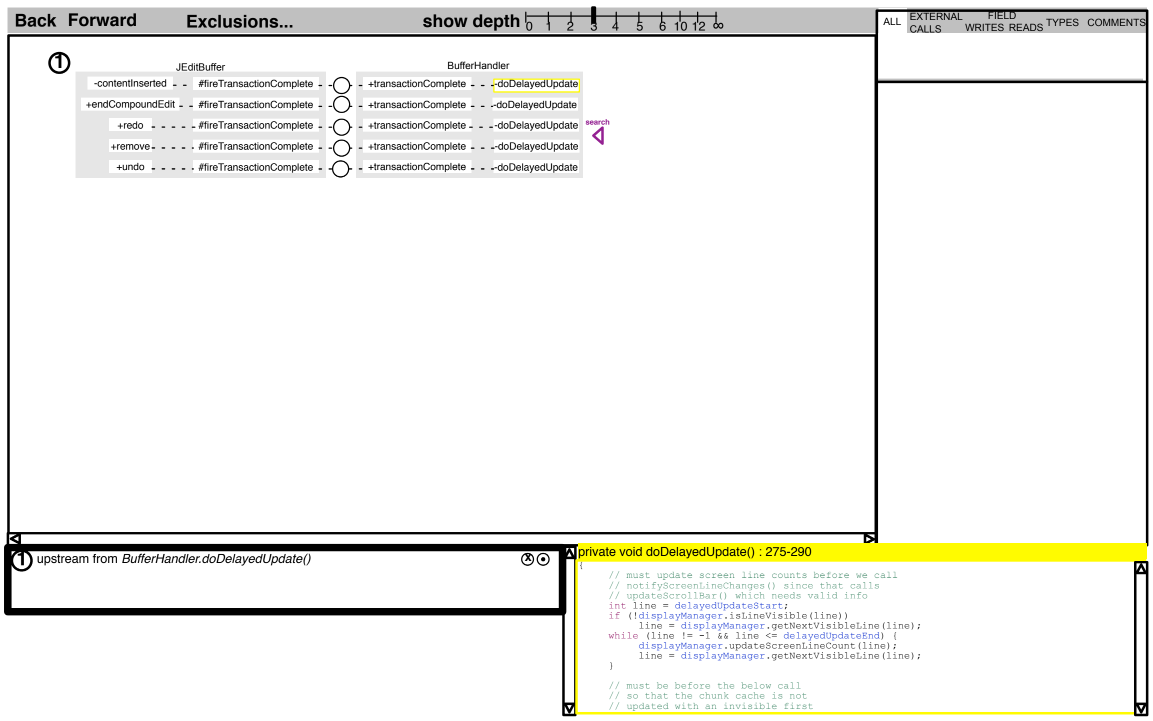


① upstream from *JEditBuffer.getFoldLevel():1473*

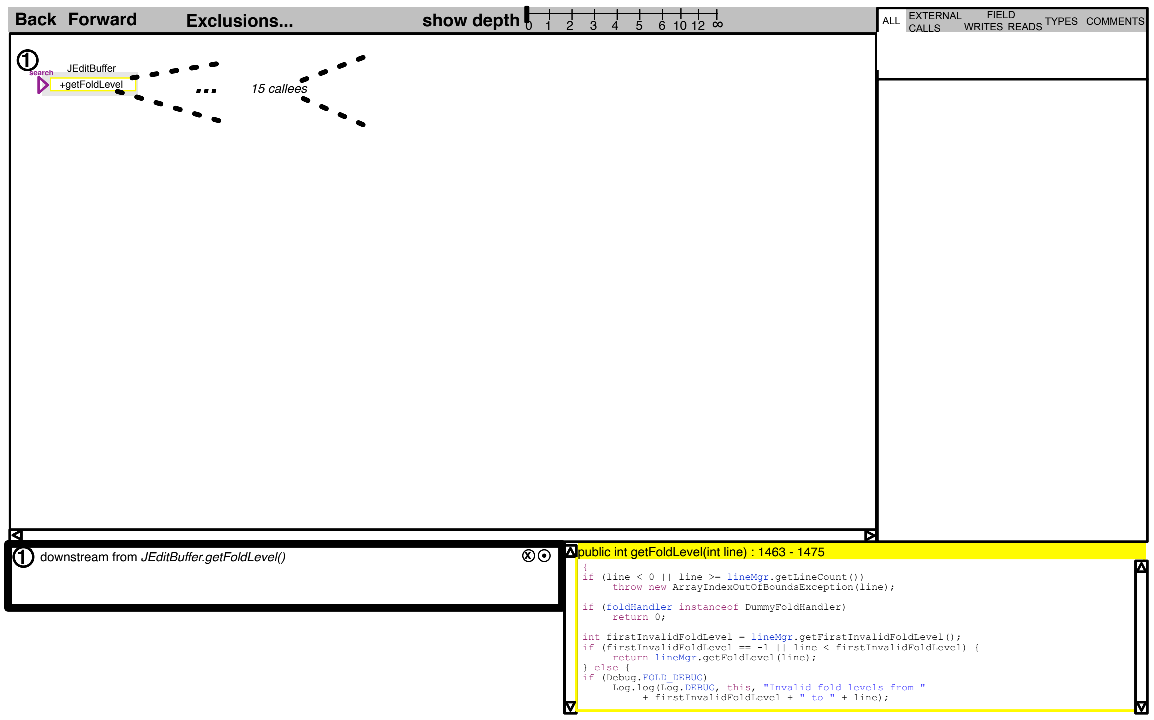
public int getFoldLevel(int line) : 1463 - 1475

```
{
if (line < 0 || line >= lineMgr.getLineCount())
     throw new ArrayIndexOutOfBoundsException(line);

if (foldHandler instanceof DummyFoldHandler)
     return 0;

int firstInvalidFoldLevel = lineMgr.getFirstInvalidFoldLevel();
if (firstInvalidFoldLevel == -1 || line < firstInvalidFoldLevel) {
     return lineMgr.getFoldLevel(line);
} else {
if (Debug.FOLD_DEBUG)
     Log.log(Log.DEBUG, this, "Invalid fold levels from "
          + firstInvalidFoldLevel + " to " + line);
```

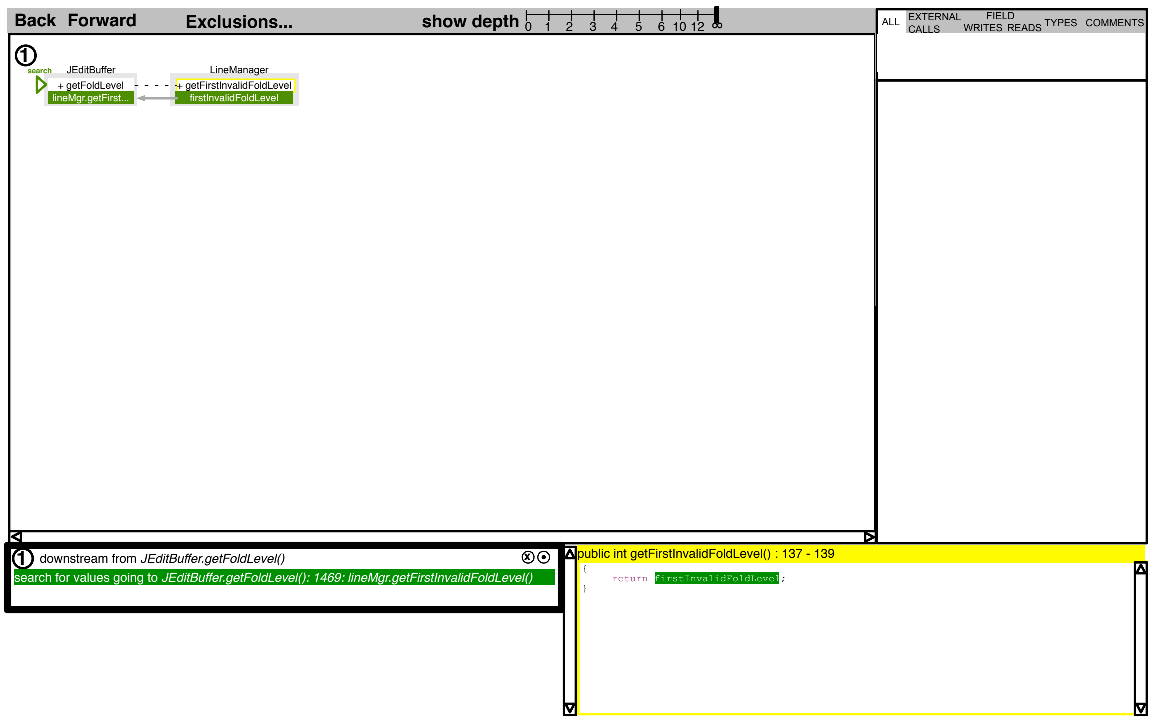

JEditBuffer
+endCompoundEdit
#firetransactionComplete
BufferHandler
+transactionComplete
-doDelayedUpdate
LineManager
+getLineOfOffset
+insert$1
-contentInserted
#fireContentInserted
#fireTransactionComplete
+contentInserted
+delayUpdate
+insert$2
Remove
+undo
+remove
#firePreContentRemoved
#fireContentRemoved
+preContentRemoved
+contentRemoved
Insert
+redo
UndoManager
CompoundEdit
cutPoint



```
{
     // must update screen line counts before we call
     // notifyScreenLineChanges() since that calls
     // updateScrollBar() which needs valid info
     int line = delayedUpdateStart;
     if (!displayManager.isLineVisible(line))
          line = displayManager.getNextVisibleLine(line);
     while (line != -1 && line <= delayedUpdateEnd) {
          displayManager.updateScreenLineCount(line);
          line = displayManager.getNextVisibleLine(line);
     }

     // must be before the below call
     // so that the chunk cache is not
     // updated with an invisible first
```



```
{
if (line < 0 || line >= lineMgr.getLineCount())
     throw new ArrayIndexOutOfBoundsException(line);

if (foldHandler instanceof DummyFoldHandler)
     return 0;

int firstInvalidFoldLevel = lineMgr.getFirstInvalidFoldLevel();
if (firstInvalidFoldLevel == -1 || line < firstInvalidFoldLevel) {
     return lineMgr.getFoldLevel(line);
} else {
if (Debug.FOLD_DEBUG)
     Log.log(Log.DEBUG, this, "Invalid fold levels from "
          + firstInvalidFoldLevel + " to " + line);
```

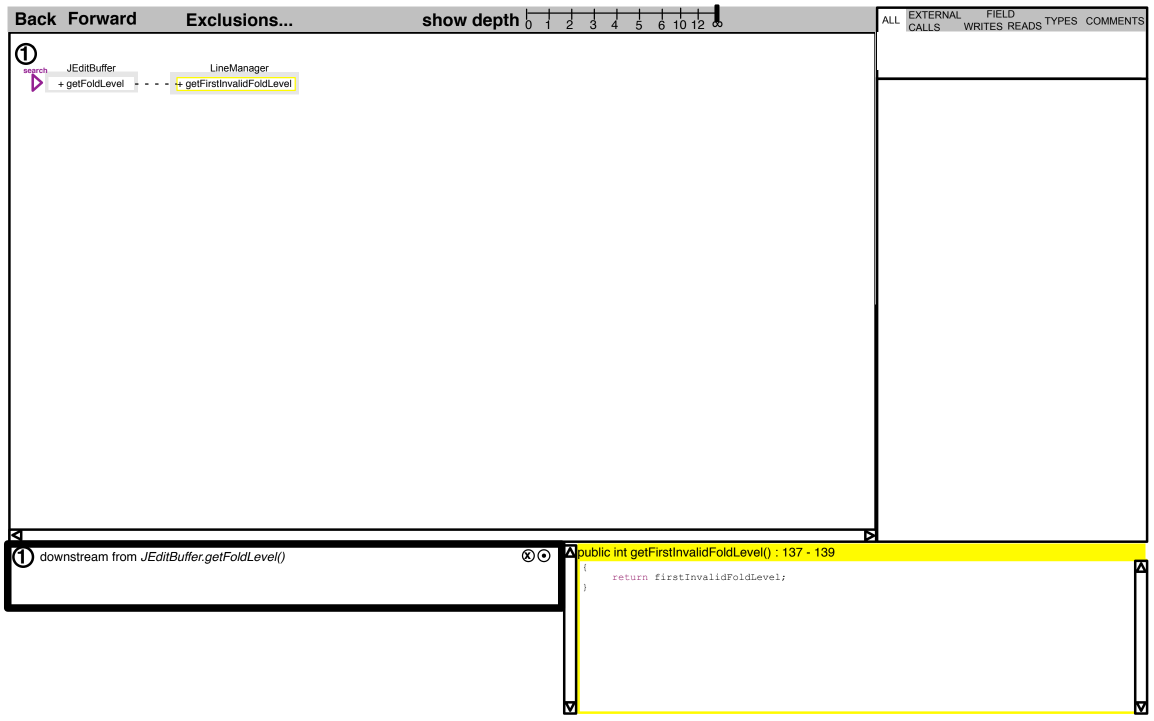


① downstream from *JEditBuffer.getFoldLevel()*

search for values going to *JEditBuffer.getFoldLevel(): 1469: lineMgr.getFirstInvalidFoldLevel()*

public int getFirstInvalidFoldLevel() : 137 - 139

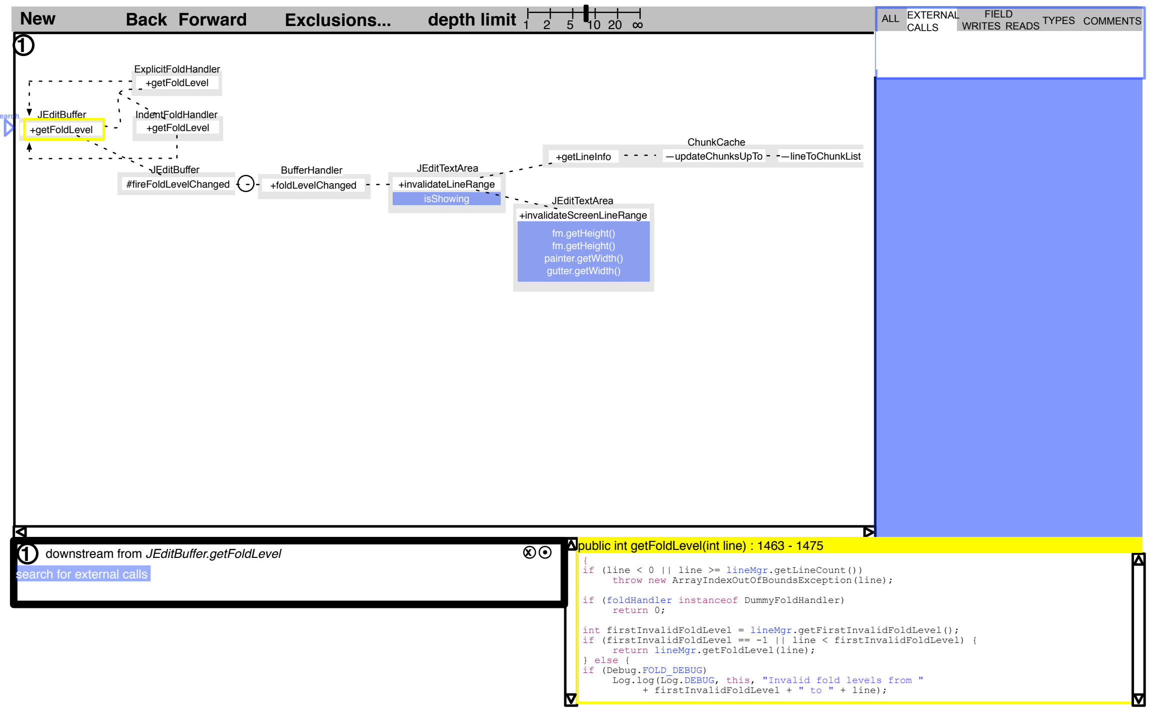
```
{
    return firstInvalidFoldLevel;
}
```



① downstream from *JEditBuffer.getFoldLevel()*

public int getFirstInvalidFoldLevel() : 137 - 139

```
{
    return firstInvalidFoldLevel;
}
```

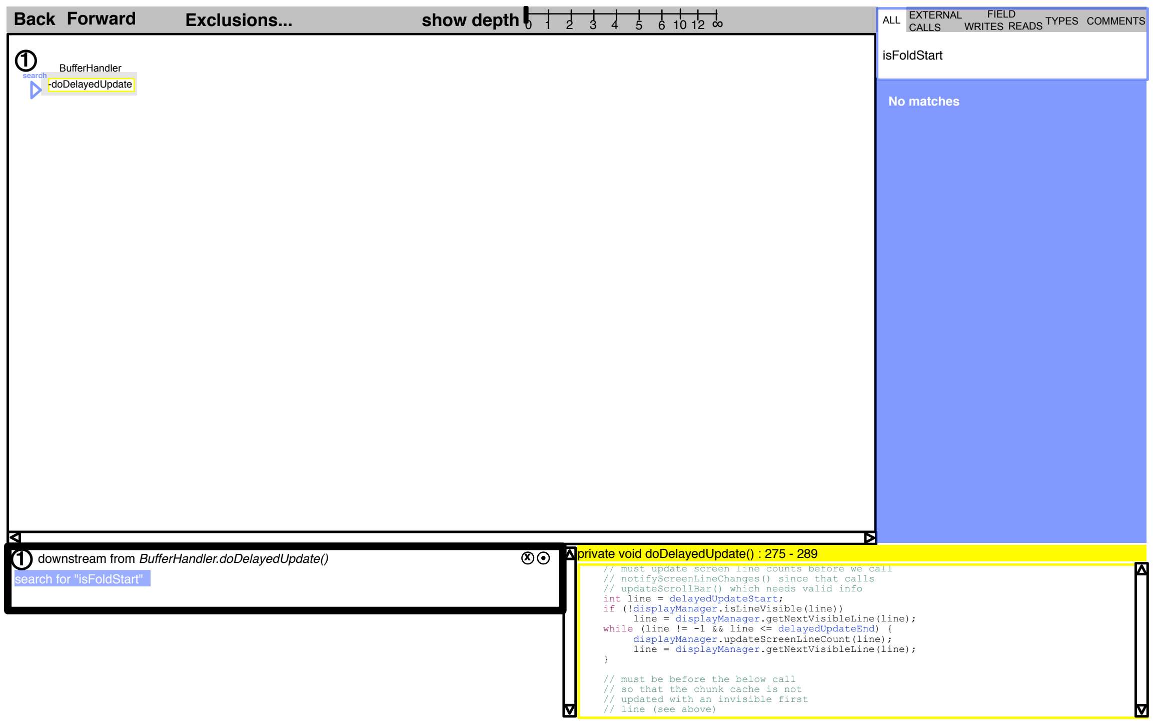


1 downstream from *JEditBuffer.getFoldLevel*

search for external calls

public int getFoldLevel(int line) : 1463 - 1475

```
{
if (line < 0 || line >= lineMgr.getLineCount())
     throw new ArrayIndexOutOfBoundsException(line);

if (foldHandler instanceof DummyFoldHandler)
     return 0;

int firstInvalidFoldLevel = lineMgr.getFirstInvalidFoldLevel();
if (firstInvalidFoldLevel == -1 || line < firstInvalidFoldLevel) {
     return lineMgr.getFoldLevel(line);
} else {
if (Debug.FOLD_DEBUG)
     Log.log(Log.DEBUG, this, "Invalid fold levels from "
          + firstInvalidFoldLevel + " to " + line);
```

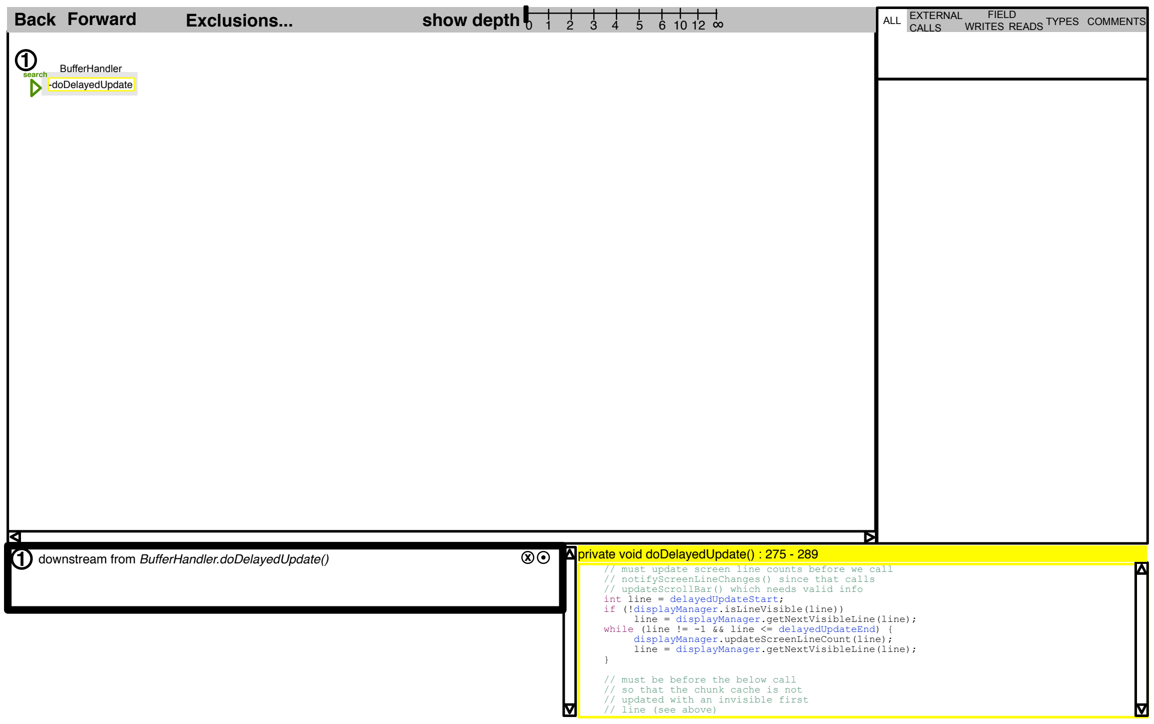


1 downstream from *BufferHandler.doDelayedUpdate()*

search for "isFoldStart"

private void doDelayedUpdate() : 275 - 289

```
    // must update screen line counts before we call
    // notifyScreenLineChanges() since that calls
    // updateScrollBar() which needs valid info
    int line = delayedUpdateStart;
    if (!displayManager.isLineVisible(line))
         line = displayManager.getNextVisibleLine(line);
    while (line != -1 && line <= delayedUpdateEnd) {
         displayManager.updateScreenLineCount(line);
         line = displayManager.getNextVisibleLine(line);
    }

    // must be before the below call
    // so that the chunk cache is not
    // updated with an invisible first
    // line (see above)
```



1 downstream from *BufferHandler.doDelayedUpdate()*

private void doDelayedUpdate() : 275 - 289

```
    // must update screen line counts before we call
    // notifyScreenLineChanges() since that calls
    // updateScrollBar() which needs valid info
    int line = delayedUpdateStart;
    if (!displayManager.isLineVisible(line))
         line = displayManager.getNextVisibleLine(line);
    while (line != -1 && line <= delayedUpdateEnd) {
         displayManager.updateScreenLineCount(line);
         line = displayManager.getNextVisibleLine(line);
    }

    // must be before the below call
    // so that the chunk cache is not
    // updated with an invisible first
    // line (see above)
```

Appendix 6: Materials from Reacher Lab Study 1 (Chapter 10)

Overview

In order to better understand how developers understand and explore code, we are conducting this study. In this study, you will perform two tasks. In each task, you will be given a question about a codebase and 30 minutes to work to answer the question. You will be working in two different Java codebases and using the Eclipse IDE. You may use any feature of the Eclipse IDE and are free to take notes if you wish. Both of the codebases you will work with are not small, so you should focus your understanding of the code simply on the portion of the code relevant to answering the questions you are given. As soon as you think you've found the answer to the question, you should let the experimenter know.

To understand how you are working, you will be asked to "think aloud" – talk about what you are thinking while you work. You should describe what you're trying to accomplish and what you're considering doing to accomplish your goals. If you find information that you've been seeking for a while, you should make sure that you say something about what you found. If you are distracted of forget to think aloud, the experimenter will prompt you to continue to think aloud by asking you about what you're doing. To allow us to analyze how you work, you will be recorded using a laptop audio recorder and a screen capture program. Your identity will not be revealed to anyone other than researchers in the research group. To ensure anonymity, only the researchers in the research group will have access to the audio and video recordings. However, transcripts of the audio recording with any personally identifying information removed and the screen recording may be used in public presentation of this work.

Eclipse Code Navigation Tutorial

Eclipse features a number of sophisticated features for navigating through source code. In this brief tutorial, you will try out several of the most useful features. Note that, as this is a Mac, all Eclipse commands are WINDOWS – [KEY], not CONTROL –KEY as on a Windows PC.

1. Press WINDOWS-SHIFT-T and type "jedit" to open up the jedit class.
2. Press WINDOWS -O to open the method outline and type getProperties to navigate to the method getProperties().
3. Hold down windows and click on getProperties() (the method call on propMgr, not the method declaration). This moves you to the method declaration.
4. Go back to the previous method you were looking at by clicking on the left arrow secondmost from the right edge of the toolbar.
5. Place the cursor over the getProperties() method declaration, right click, and select "Open Call Hierarchy". At the bottom of the screen, a tree view shows either the Callee Hierarchy or the Caller Hierarchy. Switch between them by clicking a button to the right of the Call Hiearchy tab.
6. Place the cursor on propMgr. All references to this field in the current Java file are now highlighted with a grey highlight both in the text editor and next to the scrollbar, allowing you to quickly find all of the references in the file.
7. Find all of the methods that assign the field propMgr by placing the cursor on top of it and selecting from the "Search" menu at the top of the screen "Write Access" and then "Project".
8. Find all references to the jEdit type by selecting Java from the Search menus, typing in jEdit, and selecting "Type" in Search For.

REACHER Tutorial

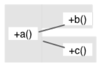
In this tutorial, you'll learn how to use REACHER, an Eclipse plugin for answering reachability questions. A reachability question is a search across control flow paths through code for target statements matching search criteria. Current development environments require developers to answer reachability questions by manually traversing across these paths. For example, to debug a deadlock using Eclipse, you might start at a root method before the deadlock and then use the call hierarchy or go to declaration to follow control flow paths forward to search. As you went, you might then look for calls that were acquiring or releasing resources. REACHER supports directly answering such questions. In REACHER, you simply start a new search in Eclipse and then enter search criteria to find matching statements. For example, to debug a deadlock you might have used REACHER to search downstream (i.e., methods called by) a root method and then entered searches for acquire and release calls. REACHER then generates a list of these statements. Clicking on each statement adds it to a visualization of control flow.

Let's use REACHER to search for statements.

- open the method StatusBar.updateCaretStatus() in Eclipse.
- Then select the method in the outline on the right,
- right click the method and select "REACHER",
- and then "Search downstream from this method". This starts REACHER.

Let's figure out what this method has to do with scrolling. The right panel has two dropdowns. The top dropdown lets you select what type of statement to search for. The second lets you select what part of its text you search for. Change REACHER to search for "any call or field access". REACHER now shows all calls and field accesses downstream. Next, enter "scro", to search for only those containing this text. Clicking on one of these statements adds it to REACHER's TraceMap view.

A TraceMap depicts paths through code as a tree of methods. Time flows from left to right and top to bottom – the in-order traversal of the tree corresponds to execution order. Visual attributes of the diagram depict information about these paths:

TypeName	type with type name
+methodName()	method name
+methodName #methodName -methodName	public / protected / private method
+methodName()	origin or destination method
expression	expression
⊕	expand hidden callees / callers
⊖	hide visible callees / callers
	a() calls b(), then c()
———	direct method call
- - - - - - -	single path with hidden methods
▬ ▬ ▬ ▬	multiple paths with hidden methods
—⟲—	method call in a loop

- Click on the last statement to add it to REACHER,
- right click on DisplayImage.getscrollLineCount() in the visualization,
- and select "Open callsite in Eclipse".

This navigates Eclipse. You can always navigate either to a method's callsite or a method's declaration.

- Open EditPane.setBuffer() in Eclipse and search downstream from this method in REACHER.
- Search for "finish". Click on one of the calls to finishCaretUpdate.
- Click the circle with a plus next to EditPane.setBuffer() to expand all of its callees.
- Click the left arrow at the top to go back.
- Then go forward again with the right arrow and back again.
- Click on a dashed line to expand a hidden path.
- Left click, hold and drag white space to pan the view.
- Scroll to zoom the view. Then click "reset zoom".

REACHER lets you save a method or statement and then search for other statements. This lets you use REACHER to see many statements and understand how these paths are related.

- Double click the selected call to finishCaretUpdate in the listview.
- Delete the text in the textbox
- Enter "setBuf" in the text box and click on one of the calls.

REACHER now shows that JEditTextArea.setBuffer() is called once before two different calls to finishCaretUpdate.

- Right click JEdtiTextArea.setBuffer() in the visualization and select "Open callsite in Eclipse"

This lets you see the call.

Task 1

Find the method BufferHandler.doDelayedUpdate(). Note that the end of this method calls JEditBuffer.getFoldLevel(). But why is a method calling a getter method but ignoring its return value? One reason why might be that the method is mutating state by assigning to a class's fields (e.g., this.field = x). In this task, your question to answer is

What are all of the statements where jEditBuffer.getFoldLevel() and any method it directly or indirectly calls (i.e., methods it calls that call other methods that call other methods ...) assigns to a field?

As you find each statement, let the experimenter know that you have found a statement.

Task 2

In this task, you will debug a NullPointerException in Fusion, a program analysis built using the Crystal Analysis Framework. In this task, you will be working with two Eclipse instances. The first Eclipse instance contains the code for Fusion. The second runs this code in a separate Eclipse instance and contains several small test classes.

First, start the child Eclipse instance by clicking run in the parent window and open *edu.cmu.cs.fusion.test.aspnet.api.Control*. Right click and select "Crystal" and then "Run analyses". This causes Fusion to run. A few seconds later it finishes. Now run Crystal on a second class – open ListControl and again invoke crystal. Note that Crystal now stops – there's a NullPointerException. *This null pointer exception occurs only happens when you run Crystal a second time.*

To see the stack trace, open the Error Log and double click on the top error message, which should be "An internal error occurred during: "Crystal"". You can also run Fusion in the debugger. To do so, close the child Eclipse window. Next, add a breakpoint to XMLRetriever.getStartingContext(). Finally, start Fusion using the debugger and run it again. You should now hit the breakpoint.

Your task will be to determine why this NullPointerException is happening. This NullPointerException is caused by methods not being called in the correct order. After Crystal begins its analysis in the method Crystal.runAnalyses(), Crystal eventually creates an XMLRetriever instance. After this, it will then eventually call the getStartContext() method on this instance in a number of places. But, first Crystal should call retrieveRelationships() to initialize several fields used by getStartContext. The NullPointerException is occurring because retrieveRelationships() is not being called before getStartContext(). In this task, your question to answer is

What are the conditions in which getStartContext() might be called without a call to retrieveRelationships() happening first?

Note, that while this is related to the null pointer exception, you do not necessarily have to get a definitive explanation for why it occurred. And the task is ***not** to implement a fix* – you could fix it by simply moving the statement initializing the null field. The task is, *given the current design*, to figure out how these calls can occur in an order which causes this exception to occur.

Appendix 7: Materials from Reacher Lab Study 2 (Chapter 10)

Overview

We are conducting this study to better understand the effects of code exploration tools on the ways in which developers explore code. In particular, we are studying how developers answer *reachability questions* – questions about what may happen in response to an event. In this study, you will perform 6 tasks, 3 with Eclipse along and 3 with a version of Eclipse augmented with a new tool. In each task, you will be given a question to answer and 15 minutes to explore the code and answer the question. You may use any feature of the Eclipse IDE and are free to take notes if you wish. The codebase in which you will work is not small, so you should focus on understanding only the code relevant to the question. As soon as you think you've answered the question, you should let the experimenter know.

To understand how you are working, you will be asked to "think aloud" – talk about what you are thinking while you work. You should describe what you're trying to accomplish and what you're considering doing to accomplish your goals. If you find information that you've been seeking for a while, you should make sure that you say something about what you found. If you are distracted of forget to think aloud, the experimenter will prompt you to continue to think aloud by asking you about what you're doing. To allow us to analyze how you work, you will be recorded using a laptop audio recorder and a screen capture program. Your identity will not be revealed to anyone other than researchers in the research group. To ensure anonymity, only the researchers in the research group will have access to the audio and video recordings. However, transcripts of the audio recording with any personally identifying information removed and the screen recording may be used in public presentation of this work.

Answering reachability questions can be hard, and the tasks have been designed to be challenging. Thus, you may not have enough time to answer some of the questions. You should work to come up with the best answer in the time you have. But, as soon as you are confident you have a complete answer to the question, please let the experimenter know.

Eclipse Code Navigation Tutorial

Eclipse features a number of sophisticated features for navigating through source code. In this brief tutorial, you will try out several of the most useful features. Note that, as this is a Mac, all Eclipse commands are WINDOWS – [KEY], not CONTROL –KEY as on a Windows PC.

1. Press WINDOWS-SHIFT-T and type "jedit" to open up the jedit class.
2. Press WINDOWS -O to open the method outline and type getProperties to navigate to the method getProperties().
3. Hold down windows and click on getProperties() (the method call on propMgr, not the method declaration). This moves you to the method declaration.
4. Go back to the previous method you were looking at by clicking on the left arrow secondmost from the right edge of the toolbar.
5. Place the cursor over the getProperties() method declaration, right click, and select "Open Call Hierarchy". At the bottom of the screen, a tree view shows either the Callee Hierarchy or the Caller Hierarchy. Switch between them by clicking a button to the right of the Call Hiearchy tab.
6. Place the cursor on propMgr. All references to this field in the current Java file are now highlighted with a grey highlight both in the text editor and next to the scrollbar, allowing you to quickly find all of the references in the file.
7. Find all of the methods that assign the field propMgr by placing the cursor on top of it and selecting from the "Search" menu at the top of the screen "Write Access" and then "Project".
8. Find all references to the jEdit type by selecting Java from the Search menus, typing in jEdit, and selecting "Type" in Search For.

REACHER Tutorial

In this tutorial, you'll learn how to use REACHER, an Eclipse plugin for exploring code. Let's start with an example question:

> *As a file is opened (jEdit.openFile(View, String)), what messages may be set on the StatusBar (StatusBar.setMessage())?*

Try for a few minutes to answer this question using only Eclipse.

Now let's look at how REACHER helps answer this question.

- Open jEdit.openFile(View, String)) in Eclipse (remember that control-shift-T can open a type).
- Right click the method in the outline or package explorer and select "REACHER" and "Search downstream from this method".

This invokes REACHER and displays the starting method in the REACHER visualization window.

- Click the plus icon next to the method to expand its callees (methods it calls).
- Click the plus icon on the new method to expand this method's callees.

The visualization now depicts several methods (boxes) and the classes in which they are located (gray background and bold label). Lines represent calls between methods and are decorated with icons describing the call. Outgoing edges from a method are sorted in the order in which they execute from top to bottom – the edge at the top happens first.

- Hover over a line to highlight it and see the tooltip.
- Click the line to navigate to the call.
- Click the method it calls to navigate to the method declaration.
- Click the "back" label at the top of the REACHER visualization window to go back.

Back to the example question. While you could expand callers to look for paths to StatusBar.setMessage(), REACHER provides a window for searching along control flow paths.

- Type "setMess" in the search textbox.
- Click on the first result to add it to the visualization

REACHER has searched for this method and found connections from your starting point. The dashed line indicates that there is a path or paths of methods which is currently hidden.

- Double click on the path to expand it. If you can't click on it, you can zoom using the scroll wheel. Click "ResetZoom" to restore the default zoom and panning.

REACHER now displays all of the paths by which setMessage() may be reached. All of the methods you see now were previously on paths hidden inside the dashed line. Now let's answer the question!

- Pan the display by clicking and dragging on the background to make sure that setMessage() is visible.
- Click on the edge from loadCaretInfo() to setMessage().

The message it sets is a null message.

- Click on the edge from dispose() to setMessage().

Also a null message. So the only message set on the StatusBar as a file is opened is a null message.

Notice that the edges are decorated with various notations (?, a loop icon, numbers, dashed lines). Each of these icons indicates something is true about the edge. Hovering over each of the icons gives an explanation of what each icon means.

- Read the description of an icon of each type in the popup window to find out what each icon means.

To review, the icons mean:

? – there are one or more conditionals affecting if the call happens or not

loop icon – the callsite in the method is enclosed by one or more loops

[n] – there are [n] paths from the source to destination method (each call inside a method is considered a separate path)

In some cases, it can be hard to tell if an edge is entering or leaving a method. In this case, simply hover over the edge to see its popup. The text will explain which method calls which and also the line of code containing the call. Note also that outgoing edges always leave from the center of the right edge while incoming edges enter on either the left or bottom edges.

Task A

When a new view is created in jEdit.newView(View), what messages, in what order, may be sent on the EditBus (EditBus.send())?

Task B

When text is deleted (JEditTextArea.delete()), what is the first message that may be sent on the EditBus (EditBus.send())?

Task C

Does setting the buffer in EditPane.setBuffer() cause the caret status on the status bar to be updated at least once (StatusBar.updateCaretStatus())?

Task D

Other than the check that the firstLine has changed from the oldFirstLine in setFirstLine(), are there other conditionals that might cause JEditTextArea.setFirstLine() not to update the scroll bar (JEditTextArea.updateScrollBar())?

Task E

How many messages may jEdit.commitTemporary() send to the EditBus? (i.e.,, how many times might it invoke EditBus.send()?) Count calls in a loop as a single call and don't count messages that themselves were caused by an EditBus message. But count multiple calls from the same method as multiple paths.

Task F

How many messages may jEdit.reloadModes() send to the EditBus? (i.e.,, how many times might it invoke EditBus.send()?) Count calls in a loop as a single call and don't count messages that themselves were caused by an EditBus message. But count multiple calls from the same method as multiple paths.

BIBLIOGRAPHY

[A95] Agesen, O. 1995. The cartesian product algorithm: simple and precise inference of parametric polymorphism. *European Conference on Object Oriented Programming (ECOOP)*.

[AAL10] Abi-Antoun, M., Ammar, N. and LaToza, T. 2010. Questions about object structure during coding activities. *ICSE Workshop on Cooperative and Human Aspects of Software Engineering* (2010).

[ABG02] Atkins, D.L., Ball, T., Graves, T.L. and Mockus, A. 2002. Using Version Control Data to Evaluate the Impact of Software Tools: A Case Study of the Version Editor. *Transactions on Software Engineering*. 28, 7 (Jul. 2002), 625–637.

[AFF06] Abadi, M., Flanagan, C. and Freund, S.N. 2006. Types for safe locking: static race detection for Java. *Transactions on Programming Languages and Systems*. 28, 2 (Mar. 2006), 207–255.

[AHM06] Anvik, J., Hiew, L. and Murphy, G.C. 2006. Who should fix this bug? *International Conference on Software Engineering (ICSE)* (2006), 361–370.

[AM06] de Alwis, B. and Murphy, G.C. 2006. Using Visual Momentum to Explain Disorientation in the Eclipse IDE. *Visual Languages and Human-Centric Computing* (2006), 51–54.

[AM08] de Alwis, B. and Murphy, G.C. 2008. Answering Conceptual Queries with Ferret. *International Conference on Software Engineering* (2008), 21–30.

[AMR07] de Alwis, B., Murphy, G.C. and Robillard, M.P. 2007. A Comparative Study of Three Program Exploration Tools. *International Conference on Program Comprehension* (2007), 103–112.

[ART03] Anderson, P., Reps, T. and Teitelbaum, T. 2003. Design and Implementation of a Fine-Grained Software Inspection Tool. *Transactions on Software Engineering*. 29, 8 (Aug. 2003), 721–733.

[AT01] Anderson, P. and Teitelbaum, T. 2001. Software Inspection Using CodeSurfer. *Workshop on Inspection in Software Engineering (CAV 2001)* (2001).

[B02] Blackwell, A.F. 2002. First steps in programming: a rationale for attention investment models. *Symposia on Human Centric Computing Languages and Environments* (2002), 2–10.

[B07] Buxton, B. 2007. *Sketching user experiences*. Morgan Kaufmann.

[B10] Bracha, G. 2010. The fitness function for programming languages: a matter of taste? *Keynote at the Evaluation and Usability of Languages and Tools (PLATEAU) at SPLASH.* (2010).

[B95] Baker, B.S. 1995. On finding duplication and near-duplication in large software systems. *Working Conference on Reverse Engineering* (1995), 86–95.

[BB08] Burge, J.E. and Brown, D.C. 2008. Software engineering using RATionale. *Journal of Systems and Software*. 81, 3 (Mar. 2008), 395–413.

[BBC06] Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S.K. and Ustuner, A. 2006. Thorough Static Analysis of Device Drivers. *Conference on Computer Systems* (2006), 73–85.

[BDW10] Brandt, J., Dontcheva, M., Weskamp, M. and Klemmer, S.R. 2010. Example-centric programming: integrating web search into the development environment. *CHI*. (Jan. 2010).

[BH97] Beyer, H. and Holtzblatt, K. 1997. *Contextual Design: Defining Customer-Centered Systems*. Morgan Kaufmann.

[BKA11] Beckman, N.E., Kim, D. and Aldrich, J. 2011. An empirical study of object protocols in the wild. *European Conference on Object-Oriented Programming* (2011).

[BMS08] Bennett, C., Myers, D., Storey, M.-A., German, D.M., Ouellet, D., Salois, M. and Charland, P. 2008. A Survey and Evaluation of Tool Features for Understanding Reverse-Engineered Sequence Diagrams. *J. Softw. Maint. Evol.* 20, (Jul. 2008), 291–315.

[BNR03] Ball, T., Naik, M. and Rajamani, S.K. 2003. From symptom to cause: localizing errors in counterexample traces. *Symposium on Principles of Programming Languages (POPL)* (2003), 97–105.

[BPZ10] Begel, A., Phang, K.Y. and Zimmermann, T. 2010. Codebook: discovering and exploiting relationships in software repositories. *International Conference on Software Engineering (ICSE)*. (2010), 125–134.

[BR01] Ball, T. and Rajamani, S.K. 2001. Automatically Validating Temporal Safety Properties of Interfaces. *International SPIN Workshop on Model Checking of Software* (2001), 103–122.

[BRZ10] Bragdon, A., Reiss, S.P., Zeleznik, R., Karumuri, S., Cheung, W., Kaplan, J., Coleman, C., Adeputra, F. and LaViola, J.J., Jr 2010. Code Bubbles: Rethinking the User Interface Paradigm of Integrated Development Environments. *International Conference on Software Engineering* (2010), 455–464.

[CBB02] Clements, P., Bachman, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R. and Stafford, J. 2002. *Documenting software architectures: views and beyond*. Addison-Wesley.

[CCD98] Canfora, G., Cimitile, A. and De Lucia, A. 1998. Conditioned program slicing. *Information and Software Technology*. 40, 11-12 (Dec. 1998), 595–607.

[CGP07] Cook, B., Gotsman, A., Podelski, A., Rybalchenko, A. and Vardi, M.Y. 2007. Proving that programs eventually do something good. *Symposium on Principles of Programming Languages (POPL)* (2007), 265–276.

[CS73] Chase, W.G. and Simon, H.A. 1973. Perception in Chess. *Cognitive Psychology*. 4, 1 (1973), 55–81.

[CSL04] Cantrill, B.M., Shapiro, M.W. and Leventhal, A.H. 2004. Dynamic instrumentation of production systems. *USENIX Annual Technical Conference* (2004).

[CVD07] Cherubini, M., Venolia, G., Deline, R. and Ko, A.J. 2007. Let's go to the whiteboard: how and why software developers use drawings. *CHI*. (Jan. 2007).

[CZ00] Cleve, H. and Zeller, A. 2000. Finding failure causes through automated testing. *International Workshop on Automated Debugging (AADEBUG)* (2000).

[D02] Detienne, F. 2002. *Software design---cognitive aspects*. Springer-Verlag.

[D68] Dijkstra, E.W. 1968. Letters to the Editor: Go To Statement Considered Harmful. *Commun. ACM*. 11, (Mar. 1968), 147–148.

[D90] Detienne, F. 1990. Program Understanding and Knowledge Organization: the Influence of Acquired Schemata. *Cognitive Ergonomics: Understanding, Learning, and Designing Human-Computer Interaction*. (1990), 245–256.

[D93] Davies, S.P. 1993. Externalizing Information during Coding Activities: Effects of Expertise, Environment, and Task. *Empirical Studies of Programmers: Fifth Workshop* (1993), 42–61.

[DAB08] Dzidek, W.J., Arisholm, E. and Briand, L.C. 2008. A Realistic Empirical Evaluation of the Costs and Benefits of UML in Software Maintenance. *IEEE Trans. Softw. Eng.* 34, (May. 2008), 407–432.

[DDH72] Dahl, O.J., Dijkstra, E.W. and Hoare, C.A.R. 1972. *Structured Programming*. Academic Press.

[DKC05] DeLine, R., Khella, A., Czerwinski, M. and Robertson, G. 2005. Towards Understanding Programs through Wear-Based Filtering. *Symposium on Software visualization* (2005), 183–192.

[DLS02] Das, M., Lerner, S. and Seigle, M. 2002. ESP: Path-Sensitive Program Verification in Polynomial Time. *Conference on Programming Language Design and Implementation* (2002), 57–68.

[DM05] Dietl, W.M. and Muller, P. 2005. Universes: lightweight ownership for JML. *Journal of Object Technology*. 4, 8 (Oct. 2005), 5–32.

[DVR10] DeLine, R., Venolia, G. and Rowan, K. 2010. Software Development with Code Maps. *Commun. ACM*. 53, (Aug. 2010), 48–54.

[E03] Erickson, C. 2003. Memory leak detection in C. *Linux Jounral*.

[E09] Edwards, J. 2009. Coherent Reaction. *Conference Companion on Object Oriented Programming Systems Languages and Applications* (2009), 925–932.

[FBL10] Fahndrich, M., Barnett, M. and Logozzo, F. 2010. Embedded contract languages. *Symposium on Applied Computing (SAC)* (2010), 2103–2110.

[FKS08] Fleming, S.D., Kraemer, E., Stirewalt, R.E.K., Xie, S. and Dillon, L.K. 2008. A study of student strategies for the corrective maintenance of concurrent software. *International Conference on Software Engineering* (2008).

[FLL02] Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B. and Stata, R. 2002. Extended Static Checking for Java. *Conference on Programming Language Design and Implementation* (2002), 234–245.

[FM10] Fritz, T. and Murphy, G.C. 2010. Using Information Fragments to Answer the Questions Developers Ask. *International Conference on Software Engineering* (2010), 175–184.

[FMH07] Fritz, T., Murphy, G.C. and Hill, E. 2007. Does a Programmer's Activity Indicate Knowledge of Code? *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)* (2007), 341–350.

[G07] Godefroid, P. 2007. Compositional Dynamic Test Generation. *Symposium on Principles of Programming Languages* (2007), 47–54.

[G75] Goodenough, J.B. 1975. Exception handling: issues and a proposed notation. *Communications of the ACM (CACM)*. 18, 12 (Dec. 1975), 683–696.

[G89] Green, T.R.G. 1989. Cognitive dimensions of notations. *People and Computers V*. A. Sutcliffe and L. Macaulay, eds. Cambridge University Press. 443–460.

[GC01] Grove, D. and Chambers, C. 2001. A Framework for Call Graph Construction Algorithms. *ACM Trans. Program. Lang. Syst.* 23, (Nov. 2001), 685–746.

[GDD97] Grove, D., DeFouw, G., Dean, J., Chambers, C., Grove, D., DeFouw, G., Dean, J. and Chambers, C. 1997. Call graph construction in object-oriented languages. *Conference on Object Oriented Programming Systems Languages and Applications* (OOPSLA), 108–124.

[GHJ95] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing.

[GKL04] Groce, A., Kroening, D. and Lerda, F. 2004. Understanding counterexamples with explain. *International Conference on Computer-Aided Verification (CAV)* (2004), 453–456.

[GO86] Gugerty, L. and Olson, G.M. 1986. Comprehension Differences in Debugging by Skilled and Novice Programmers. *Empirical Studies of Programmers*. (1986), 13–27.

[H09] Hanenberg, S. 2009. What is the impact of static type systems on programming time? *PLATEAU Workshop at OOPSLA '09*. (Oct. 2009).

[H11] How do practitioners perceive Software Engineering Research?: 2011. *http://catenary.wordpress.com/2011/05/19/how-do-practitioners-perceive-software-engineering-research/*. Accessed: 2011-08-01.

[H69] Hoare, C.A.R. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM*. 12, (Oct. 1969), 576–580.

[HB08] Holmes, R. and Begel, A. 2008. Deep intellisense: a tool for rehydrating evaporated information. *Working Conference on Mining Software Repositories* (2008), 23–26.

[HCL05] Heer, J., Card, S.K. and Landay, J.A. 2005. Prefuse: a Toolkit for Interactive Information Visualization. *Conference on Human Factors in Computing Systems* (2005), 421–430.

[HH01] Harman, M. and Hierons, R. 2001. An overview of program slicing. *Software Focus*.

[HH11] Howison, J. and Herbsleb, J.D. 2011. Scientific software production: incentives and collaboration. *Computer Supported Cooperative Work* (2011).

[HM06] Hajiyev, M.V.E. and de Moor, O. 2006. CodeQuest: Scalable Source Code Queries with Datalog. *European Conference on Object-Oriented Programming* (2006), 2–27.

[HPV07] Hill, E., Pollock, L. and Vijay-Shanker, K. 2007. Exploring the Neighborhood with Dora to Expedite Software Maintenance. *International Conference on Automated Software Engineering* (2007), 14–23.

[J11] JUnit: A Regression Testing Framework: Accessed: 2011-07-11.

[JC07] Jaspan, C., Chen, I.-C. and Sharma, A. 2007. Understanding the Value of Program Analysis Tools. *Companion to the Conference on Object-Oriented Programming Systems and Applications* (2007).

[JHS02] Jones, J.A., Harrold, M.J. and Stasko, J. 2002. Visualization of test information to assist fault localization. *International Conference on Software Engineering (ICSE)* (2002), 467–477.

[JR94] Jackson, D. and Rollins, E.J. 1994. A new model of program dependencies for reverse engineering. *Symposium on Foundations of Software Engineering (FSE)* (1994), 2–10.

[JSB97] Jerding, D.F., Stasko, J.T. and Ball, T. 1997. Visualizing Interactions in Program Executions. *International Conference on Software Engineering* (1997), 360–370.

[JV03] Janzen, D. and de Volder, K. 2003. Navigating and Querying Code without Getting Lost. *International Conference on Aspect-Oriented Software Development* (2003), 178–187.

[K08] Ko, A.J. 2008. *Asking and answering questions about the causes of software behavior*. Dissertation, Carnegie Mellon University.

[K76] King, J.C. 1976. Symbolic Execution and Program Testing. *Commun. ACM*. 19, (Jul. 1976), 385–394.

[KAM05] Ko, A.J., Aung, H. and Myers, B.A. 2005. Eliciting Design Requirements for Maintenance-Oriented IDEs: A Detailed Study of Corrective and Perfective Maintenance Tasks. *International Conference on Software Engineering* (2005), 126–135.

[KBL04] Kim, M., Bergman, L., Lau, T. and Notkin, D. 2004. An ethnographic study of copy and paste programming practices in OOPL. *International Symposium on Empirical Software Engineering (ISESE)*, 83–92.

[KDV07] Ko, A.J., DeLine, R. and Venolia, G. 2007. Information Needs in Collocated Software Development Teams. *International Conference on Software Engineering* (2007), 344–353.

[KEL10] Kuhn, A., Erni, D., Loretan, P. and Nierstrasz, O. 2010. Software Cartography: Thematic Software Visualization with Consistent Layout. *J. Softw. Maint. Evol.* 22, (Apr. 2010), 191–210.

[KEN10] Kuhn, A., Erni, D. and Nierstrasz, O. 2010. Embedding Spatial Software Visualization in the IDE: An Exploratory Study. *International Symposium on Software Visualization* (2010), 113–122.

[KFH08] Khoo, Y.P., Foster, J.S., Hicks, M. and Sazawal, V. 2008. Path Projection for User-Centered Static Analysis Tools. *Workshop on Program Analysis for Software Tools and Engineering* (2008), 57–63.

[KHH01] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W.G. 2001. An Overview of AspectJ. *European Conference on Object-Oriented Programming* (2001), 327–353.

[KKI02] Kamiya, T., Kusumoto, S. and Inoue, K. 2002. CCFinder: A Multilinguistic Token-Based Code Clone Detection System. *Transactions on Software Engineering*. 28, 7 (Jul. 2002).

[KM05] Ko, A.J. and Myers, B. 2005. A framework and methodology for studying the causes of software errors in programming systems. *Journal of Visual Languages and Computing*. (Jan. 2005).

[KM08] Ko, A.J. and Myers, B.A. 2008. Debugging Reinvented: Asking and Answering Why and Why Not Questions about Program Behavior. *International Conference on Software Engineering* (2008), 301–310.

[KM09] Ko, A.J. and Myers, B.A. 2009. Finding Causes of Program Output with the Java Whyline. *Conference on Human Factors in Computing Systems* (2009), 1569–1578.

[KMC06] Ko, A.J., Myers, B.A. and Chau, D.H. 2006. A linguistic analysis of how people describe software problems. *Visual Languages and Human-Centric Computing*. (2006).

[KMCA06] Ko, A.J., Myers, B.A., Coblenz, M.J. and Aung, H.H. 2006. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. *IEEE Trans. Softw. Eng.* 32, (Dec. 2006), 971–987.

[KSN05] Kim, M., Sazawal, V., Notkin, D., and Murphy, G. 2005. An empirical study of code clone genealogies. *European Software Engineering Conference and the Foundations of Software Engineering (ESEC/FSE),* 187-196.

[KSS02] Kollman, R., Selonen, P., Stroulia, E., Systa, T. and Zundorf, A. 2002. A study on the current state of the art in tool-supported UML-based static reverse engineering. *Working Conference on Reverse Engineering (WCRE)* (2002).

[L11] Lehnert, S. 2011. A taxonomy for software change impact analysis. *International Workshop on Principles of Software Evolution and the Workshop on Software Evolution (IWPSE-EVOL)* (2011), 41–50.

[L87] Letovsky, S. 1987. Cognitive Processes in Program Comprehension. *Journal of Systems and Software*. 7, 4 (Dec. 1987), 325–339.

[LAZ03] Liblit, B., Aiken, A., Zheng, A.X. and Jordan, M.I. 2003. Bug isolation vis remote program sampling. *Programming Language Design and Implementation (PLDI)* (2003), 141–154.

[LBB08] Lawrance, J., Bellamy, R., Burnett, M. and Rector, K. 2008. Using Information Scent to Model the Dynamic Foraging Behavior of Programmers in Maintenance Tasks. *Conference on Human Factors in Computing Systems* (2008), 1323–1332.

[LBB10] Lawrance, J., Bogart, C., Burnett, M., Bellamy, R., Rector, K. and Fleming, S. 2010. How Programmers Debug, Revisited: An Information Foraging Theory Perspective. *IEEE Transactions on Software Engineering*. 99 (2010).

[LC06] Leavens, G.T. and Cheon, Y. 2006. Design by contract with JML. (2006), 1–13.

[LGH07] LaToza, T.D., Garlan, D., Herbsleb, J.D. and Myers, B.A. 2007. Program Comprehension as Fact Finding. *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering* (2007), 361–370.

[LM10-1] LaToza, T.D. and Myers, B.A. 2010. Developers Ask Reachability Questions. *International Conference on Software Engineering* (2010), 185–194.

[LM10-2] LaToza, T.D. and Myers, B.A. 2010. Hard-to-Answer Questions about Code. *Second Workshop on the Evaluation and Usability of Programming Languages and Tools at SPLASH '10* (2010).

[LM10-3] LaToza, T.D. and Myers, B.A. 2010. On the importance of understanding the strategies the developers use. *ICSE Workshop on Cooperative and Human Aspects of Software Engineering* (2010).

[LM11] LaToza, T.D. and Myers, B.A. 2011. Designing useful tools for developers. *Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU)* (2011).

[LM11] LaToza, T.D. and Myers, B.A. 2011. Visualizing Call Graphs. *Visual Languages and Human-Centric Computing* (2011).

[LSY03] Linden, G., Smith, B. and York, J. 2003. Amazon.com Recommendations: Item-to-Item Collaborative Filtering. *IEEE Internet Computing*. 7, (Jan. 2003), 76–80.

[LVD06] LaToza, T.D., Venolia, G. and DeLine, R. 2006. Maintaining Mental Models: A Study of Developer Work Habits. *International Conference on Software Engineering* (2006), 492–501.

[M92] Meyer, B. 1992. Applying "Design by contract." *IEEE Computer*. 25, 10 (Oct. 1992), 40–51.

[MB08] Murphy-Hill, E. and Black, A.P. 2008. Seven habits of a highly effective smell detector. *International Workshop on Recommendation Systems for Software Engineering* (2008), 36–40.

[MC96] Moran, T.P. and Carroll, J.M. eds. 1996. *Design Rationale: Concepts, Techniques, and Use*. Lawrence Erlbaum Associates, Inc.

[MGM93] Maulsby, D., Greenberg, S. and Mander, R. 1993. Prototyping an intelligent agent through Wizard of Oz. *Conference on Human Factors in Computing Systems* (1993), 277–284.

[MH02] Mockus, A. and Herbleb, J.D. 2002. Expertise browser: a quantitative approach to identifying. *International Conference on Software Engineering* (2002), 503–512.

[MPB09] Murphy-Hill, E., Parnin, C. and Black, A.P. 2009. How we refactor, and how we know it. *International Conference on Software Engineering* (2009), 287–297.

[MS10] Myers, D. and Storey, M.-A. 2010. Using Dynamic Analysis to Create Trace-Focused User Interfaces for IDEs. *International Symposium on Foundations of Software Engineering* (2010), 367–368.

[NNH04] Nielson, F., Nielson, H.R. and Hankin, C. 2004. *Principles of program analysis*. Springer.

[P02] Patton, M.Q. 2002. *Qualitative research and evaluation methods*. Sage Publications.

[P06] Parent, S. 2006. A possible future for software development. *Keynote talk at the Workshop on Library-Centric Software Design at OOPSLA* (2006).

[P11] Pattern Insight: *http://patterninsight.com*. Accessed: 2011-10-05.

[P72] Parnas, D.L. 1972. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*. 15, 12 (Decemeber. 1972), 1053–1058.

[P87] Pennington, N. 1987. Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs. *Cognitive Psychology*. 19, 3 (1987), 295–341.

[PC99] Pirolli, P. and Card, S. 1999. Information Foraging. *Psychology Review*. 106, 4 (1999), 643–675.

[PO11] Parnin, C. and Orso, A. 2011. Are Automated Debugging Techniques Actually Helping Developers? *International Symposium on Software Testing and Analyisis* (2011), 199–209.

[PR03] Renieris, M. and Reiss, S. 2003. Fault localization with nearest neighbor queries. *Automated Software Engineering (ASE)* (2003), 30–39.

[PSV94] Perry, D., Staudenmayer, N. and Votta, L.G. 1994. People, Organizations, and Process Improvement. *IEEE Software*.

[PW92] Perry, D.E. and Wolf, A.L. 1992. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*. 17, 4 (Oct. 1992), 40–52.

[R08] Robillard, M.P. 2008. Topology Analysis of Software Dependencies. *ACM Transactions on Software Engineering and Methodology*. 17, (Aug. 2008), 18:1–18:36.

[RC96] Rosson, M.B. and Carroll, J.M. 1996. The reuse of uses in Smalltalk programming. *ACM Transactions on Computer-Human Interaction*. 3, 3 (Sep. 1996), 219–253.

[RCM04] Robillard, M.P., Coelho, W. and Murphy, G.C. 2004. How Effective Developers Investigate Source Code: An Exploratory Study. *IEEE Trans. Softw. Eng.* 30, (Dec. 2004), 889–903.

[RJB99] Rumbaugh, J., Jacobson, I. and Booch, G. eds. 1999. *The Unified Modeling Language Reference Manual*. Addison-Wesley Longman Ltd.

[S08] Sutherland, D.F. 2008. *The code of many colors: semi-automated reasoning about multi-thread policy for Java*. Dissertation, Carnegie Mellon University.

[S96] Steensgaard, B. 1996. Points-to analysis in almost linear time. Symposium on Principles of Programming Languages (POPL), 32–41.

[SB01] Schwaber, K. and Beedle, M. 2001. *Agile software development with Scrum*. Prentice Hall.

[SC07] Stylos, J. and Clarke, S. 2007. Usability implications of requiring parameters in objects' constructors. *International Conference on Software Engineering*. (2007), 529–539.

[SFB07] Sridharan, M., Fink, S.J. and Bodik, R. 2007. Thin Slicing. *Conference on Programming Language Design and Implementation* (2007), 112–122.

[SFY09] Stylos, J., Faulring, A., Yang, Z. and Myers, B.A. 2009. Improving API documentation using API usage information. *Visual Languages and Human-Centric Computing (VL/HCC)* (2009), 119–126.

[SG96] Shaw, M. and Garlan, D. 1996. *Software architecture: perspectives on an emerging discipline*. Prentice-Hall.

[SGW01] Sullivan, K.J., Griswold, W.G., Cai, Y. and Ben Hallen 2001. The structure and value of modularity in software design. *Foundations of Software Engineering (ESEC/FSE)* (2001), 99–108.

[SJS05] Sangal, N., Jordan, E., Sinha, V. and Jackson, D. 2005. Using dependency models to manage complex software architecture. *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (2005), 167–176.

[SLV97] Singer, J., Lethbridge, T., Vinson, N. and Anquetil, N. 1997. An Examination of Software Engineering Work Practices. *CASCON* (1997), 209–223.

[SM95] Storey, M.-A.D. and Muller, H.A. 1995. Manipulating and Documenting Software Structures using SHriMP Views. *International Conference on Software Maintenance* (1995).

[SMK06] Sinha, V., Miller, R. and Karger, D. 2006. Relo: Helping Users Manage Context during Interactive Exploratory Visualization of Large Codebases. *Visual Languages and Human-Centric Computting (VL/ HCC)* (2006), 4–8.

[SMV08] Sillito, J., Murphy, G.C. and de Volder, K. 2008. Asking and Answering Questions during a Programming Change Task. *IEEE Trans. Softw. Eng.* 34, (Jul. 2008), 434–451.

[SP81] Sharir, M. and Pnueli, A. 1981. Two approaches to interprocedural data flow analysis. *Program Flow Analysis: Theory and Applications.* S.S. Muchnick and N.D. Jones, eds. Prentice-Hall. 189–233.

[SV09] Sutherland, A. and Venolia, G. 2009. Can peer code reviews be exploited for later information needs? (2009), 259–262.

[SY96] Strom, R.E. and Yemini, S. 1986. Typestate: a programming language concept for enhancing software reliability. *Transactions on Software Engineering (TSE)*. 12, 1 (Jan. 1986), 157–171.

[T95] Tip, F. 1995. A survey of program slicing techniques. *Journal of Programming Languages*. (1995).

[TBG04] Toomim, M., Begel, A. and Graham, S.L. 2004. Managing Duplicated Code with Linked-Editing. *Symposium on Visual Languages and Human-Centric Computing* (2004).

[TH08] Tillmann, N. and de Halleux, J. 2008. Pex- white box text generation for .NET. *International Conference on Tests and Proofs* (2008), 134–153.

[V79] Valiant, L.G. 1979. The complexity of enumeration and reliability problems. *SIAM J. Computing*. 8, 3 (Aug. 1979), 410–421.

[W82] Weiser, M. 1982. Programmers Use Slices When Debugging. *Commun. ACM*. 25, (Jul. 1982), 446–452.

[W84] Weiser, M. 1984. Program Slicing. *IEEE Trans. Software Eng.* 10, 4 (1984), 352–357.

[WA87] Wright, G. and Ayton, P. 1987. Eliciting and modeling expert knowledge. *Decision Support Systems*. 3, 1 (Mar. 1987), 13–26.

[WGR10] Wursch, M., Ghezzi, G., Reif, G. and Gall, H.C. 2010. Supporting Developers with Natural Language Queries. *International Conference on Software Engineering* (2010), 165–174.

[WS95] Wilde, N. and Scully, M.C. 1995. Software reconnaissance: mapping program features to code. *Journal of Software Maintenance: Research and Practice*. 7, 1 (Jan. 1995), 49–62.

[YC79] Yourdon, E. and Constantine, L.L. 1979. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice-Hall.

[ZZW05] Zimmermann, T., Zeller, A., Weissgerber, P. and Diehl, S. 2005. Mining version histories to guide software changes. *Transactions on Software Engineering (TSE)*. 31, 6 (Jun. 2005), 429–445.