Hard-to-Answer Questions about Code

Thomas D. LaToza
Institute for Software Research
School of Computer Science
Carnegie Mellon University
tlatoza@cs.cmu.edu

Brad A. Myers
Human Computer Interaction Institute
School of Computer Science
Carnegie Mellon University
bam@cs.cmu.edu

Abstract

To build new tools and programming languages that make it easier for professional software developers to create, debug, and understand code, it is helpful to better understand the questions that developers ask during coding activities. We surveyed professional software developers and asked them to list hard-to-answer questions that they had recently asked about code. 179 respondents reported 371 questions. We then clustered these questions into 21 categories and 94 distinct questions. The most frequently reported categories dealt with intent and rationale – what does this code do, what is it intended to do, and why was it done this way? Many questions described very specific situations – e.g., what does the code do when an error occurs, how to refactor without breaking callers, or the implications of a specific change on security. These questions revealed opportunities for both existing research tools to help developers and for developing new languages and tools that make answering these questions easier.

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

General Terms Experimentation, Human Factors

Keywords program comprehension, developer questions

1. Introduction

A central part of developers' work is answering questions about code [19][15][24]. Better understanding what questions developers ask and the challenges developers face answering these questions has several benefits. When these questions match challenges addressed by existing tools or language features, they provide evidence that the challenges addressed are important and scenarios for conducting evaluations: do they help developers more effectively answer these questions compared to their existing approaches? For questions that are not directly supported by existing approaches, these questions reveal opportunities for new tools and programming languages. These questions capture a problem developers experience: approaches that support these questions can help address this problem.

We conducted a survey of professional software developers to discover questions about code that developers perceive to be hard-to-answer. Respondents reported 371 questions. We then grouped these questions into 94 distinct questions spanning 21 categories. Figure 1 lists the five most frequently reported categories. 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 they provide to developers.

Rationale (42): *Why wasn't it done this other way? (15)*
Intent and implementation (32): *What does this do (6) in this case (10)? (16)*
Debugging (26): *How did this runtime state occur? (12)*
Refactoring (25): *How can I refactor this (2) without breaking existing users(7)? (9)*
History (23): *Who, when, how, and why was this code changed or inserted? (13)*

Figure 1. The 5 most frequently reported question categories, each category's most frequent question, and the number of reports in parenthesis.

In some cases, we found research tools that might help answer the questions developers ask. For example, the concurrency question "What threads reach this code?" is answered directly by thread coloring [28]. In other cases, we only found research tools for related questions or which do not work in the situations that developers identified. For example, thin slicing helps traverse data flow, but only backwards, not for the forward questions developers also reported. Other questions revealed open, unexplored research areas – e.g., what is the policy for doing this, or how can this code be generalized? Each of these questions might serve as inspiration for new tools or language features.

In this paper, we report 94 hard-to-answer questions about code, discuss challenges developers face when answering them, and speculate on how new tools or languages might make these questions easier to answer.

2. Related work

Several studies have examined developers' information needs and reported them using lists of 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) [21].

One study found 21 questions about interactions with code, other artifacts, and teammates [15]. When writing code, developers seek functionality to reuse and information on 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 looked like and when it occurs before asking about its cause. Developers ask design questions about code's rationale and implications of a change. Our respondents reported 11 of these 21 questions, which are denoted below with a citation (i.e., [15]).

Anther study identified 44 questions about code [24]. Developers begin coding tasks by finding focus points corresponding to domain concepts or application functionality and work outward following relationships between methods and classes. Developers ask higher-level questions about relationships between multiple methods and classes, including questions about control and data flow. Our respondents reported 25 of these 44 questions and are cited below or were elsewhere in our survey. Another study identified 78 questions that involve integrating information in code, work items, change sets, teams, comments, and the web [7]. Our respondents reported 10 of these questions.

Our results build on these studies by using survey data, rather than observations, to sample more questions, focusing on questions that developers perceive to be hard-to-answer rather than all questions, and identifying 67 questions that have not been previously reported.

3. Method

We conducted a survey of software developers at Microsoft as part of a larger study investigating reachability questions [19]. We invited approximately 2000 developers to participate by randomly sampling all developers at Microsoft. 469 developers responded. The complete survey included several demographic items, ratings of 14 control flow related questions, and a free response item about other hard-to-answer questions. This paper focuses on answers to the free response item, which was completed by 179 developers. 149 were individual contributor developers, 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%).

After being primed by rating specific control flow questions in the first section, developers answered the free response question "What other hard to answer questions about code have you recently asked?" Responses included both questions and whole stories illustrating the factors that developers perceived made these questions challenging. To analyze responses, we broke up each answer into individual questions, yielding 371 questions and clustered them 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 a question about runtime behavior, not rationale. Finally, within each category, we clustered each into distinct questions.

4. Results

Developers reported hard-to-answer questions about 21 different categories across three topics: changes, properties of elements, and relationships between elements. For each category, we list each distinct question reported and the number of reports in parenthesis, discuss examples of hard-to-answer questions, and speculate on how tools might help answer these questions.

4.1 Questions about Changes

4.1.1 Debugging (26)

How did this runtime state occur? (12) [15]
What runtime state changed when this executed? (2)
Where was this variable last changed? (1)
How is this object different from that object? (1)
Why didn't this happen? (3)
How do I debug this bug in this environment? (3)
In what circumstances does this bug occur? (3) [15]
Which team's component caused this bug? (1)

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. Developers wondered about differences between executions – why functionality did not work on some browsers – or the circumstances necessary for the bug to occur. Environments that could not be recreated locally and crash dumps were particularly challenging to debug. When debugging, developers did not always seek to devise a fix – some simply wanted to trace the flaw far enough to understand which team should be assigned the bug.

Several of these questions are supported by research tools. Omniscient debuggers, such as the WhyLine [16], record and play back traces or even support directly asking why a behavior did or did not occur. However, these tools are not applicable in situations such as debugging crash dumps. Mining change histories sometimes helps determine the developer who should fix a bug [2], but not when the developer must first debug to localize the failure.

4.1.2 Implementing (19)

How do I implement this (8), given this constraint (2)? (10)
Which function or object should I pick? (2)
What's the best design for implementing this? (7)

Developers with partially formed ideas for a change asked hard-to-answer questions about how to implement it. Changes included 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 sought only implementations subject to fixed constraints such as API backwards compatibility. When reusing functionality, developers asked about differences between similar methods or objects to decide which to pick. Finally, developers weighed design quality tradeoffs – where should functionality be located between callers or callees or in classes or layers.

Pattern catalogs (e.g., [8]) aim to help developers generate implementations for challenging problems and better understand relationships between alternatives. But they likely capture only a few of the patterns used in practice – most challenging changes developers reported are not covered by any catalog. Moreover, they provide only general solutions – to the extent that challenges lie in understanding the characteristics of a situation, they may help less.

4.1.3 Policies (15)

What is the policy for doing this? (10) [24]
Is this the correct policy for doing this? (2) [15]
How is the allocation lifetime of this object maintained? (3)

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. Helping developers find policies is an open, unexplored problem.

4.1.4 Rationale (42)

Why was it done this way? (14) [15][7]
Why wasn't it done this other way? (15)
Was this intentional, accidental, or a hack? (9)[15]
How did this ever work? (4)

Rationale questions were the most frequently reported category of questions, similar to a previous study [20]. Developers attempted to understand the rationale for surprising design decisions by contemplating what hidden criteria motivated the choice [18]. Surprising decisions included naming, code structure, inheritance relationships, where resources are freed, code duplication, lack of instrumentation, lack of refactoring, reimplimenting instead of reusing, algorithm choice, optimizations, where behavior is implemented, parameter validation, visibility, and exception policies. Some questions explicitly referenced alternatives the developer expected to see. Others speculated whether the decision reflected a hack made in haste, was erroneously overlooked, reflected a lack of knowledge, was the right decision to make at the time, or was a considered judgment reflecting a deeper understanding of the problem [15]. In other cases, developers were extremely surprised (How did this work?) and nearly convinced that the decision was a bug. While some rationale questions could be answered by implementing a change and testing, most of the reported decisions concerned non-functional properties where testing or verification is not possible.

Despite their prevalence, effective support for answering rationale questions remains an open problem. A popular strategy – ask an expert teammate [20] – interrupts the teammate, interrupts the question asker when the teammate is unavailable, and does not work when the teammate has left the company. Many questions concerned decisions about design, algorithm choice, and other issues that are important but difficult to test. Systems for explicitly representing rationale have been devised, but have mostly focused on higher-level decisions earlier in the life cycle [23]. Systems for browsing code history (see 4.1.5) may help. Comments might also help, but require future questions to be anticipated, the comments to be correctly updated, and the author to make the time investment. Moreover, some questions reflected questions about decisions that are infrequently commented.

4.1.5 History (23)

When, how, by whom, and why was this code changed or inserted? (13)[7]
What else changed when this code was changed or inserted? (2)
How has it changed over time? (4)[7]
Has this code always been this way? (2)
What recent changes have been made? (1)[15][7]
Have changes in another branch been integrated into this branch? (1)

One strategy for answering rationale questions is to find the code's creation in history to understand its context and motivation. Developers ask questions about who made the change, the date, content, and rationale for code's existence and to what else it was related. In other situations, developers wanted to know the entire history of a block of code, rather than its most recent change. Sometimes, developers were interested in the design or intent behind how abstractions had evolved. Developers also tracked history between version control systems and determined if changes had been migrated between branches. Confirming previous findings, developers were usually interested in history at the level of a code snippet [12], while many existing tools require searching through all changes at the file level to find changes to a snippet.

Research systems have been built to connect code snippets to related historical artifacts including commit comments, bug descriptions, and emails. For example, Deep Intellisense [12] mines linkages between artifacts and recommends artifacts based on the currently selected code.

4.1.6 Implications (21)

What are the implications of this change for (5) API clients (5), security (3), concurrency (3), performance (2), platforms (1), tests (1), or obfuscation (1)? (21) [15][24]

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 [18]. In contrast to how questions with constraints but no proposals, developers had proposals and wanted to find constraints. When changing functionality exposed to other components or other teams, developers wondered if changes would cause bugs elsewhere or require versioning to keep existing behavior. Developers wondered if changes could introduce security concerns or timing issues such as deadlocks or how a change would affect performance characteristics such as execution time or network or disk usage.

While implementing and testing a change can help answer some implication questions, many of the questions require better tool or language support.

4.1.7 Refactoring (25)

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

Developers ask questions when simplifying or generalizing code by refactoring. Developers look for refactoring candidates – obsolete code, duplicated functionality, or redundant data between equally accessible structures. Developers consider the design qualities of the existing design, ask if it

is possible to refactor, and if so, how? Most how questions involved generating proposals subject to constraints imposed by maintaining compatibility with callers. Finally, developers consider whether the benefits of the refactoring outweighed the costs, especially with respect to the time investment to make the change. Challenging refactorings developers reported included changing a method's scope, moving functionality between layers, changing the implementation of configuration values, making operations more data driven, or generalizing code to be more reusable.

Modern development environments can automatically perform certain refactorings by checking for necessary preconditions and updating the code. However, support is limited to low-level, focused changes – e.g., moving code between methods or renaming methods – and does not directly support the higher-level changes developers reported. Code duplication detectors (e.g., [14]) can find syntactically similar code snippets, but are not capable of finding redundant functionality or data.

4.1.8 Testing (20)

Is this code correct? (6) [15]
How can I test this code or functionality? (9)
Is this tested? (3)
Is the test or code responsible for this test failure? (1)
Is the documentation wrong, or is the code wrong? (1)

Developers checking their changes for mistakes ask testing questions. Developers wondered if code was correct – if it did what the comments implied or the callers expected, worked in situations with multiple users or servers, and if it had security vulnerabilities. Developers asked how to test code which depended on an external API or with error paths and how to check for memory leaks, race conditions, or hangs. Developers wondered if existing unit tests already exercised functionality or codepaths. When discovering problems, developers wondered if there was a problem with the test, the documentation, or the code.

Research and industrial tools exist to automatically check for memory leaks (e.g., [6]), race conditions (e.g., [1]), and hangs [4] without any developer input. Research tools also test robustness by deriving program inputs to run a program through its paths (e.g., [9]). However, tools testing for correct behavior require a specification of correct behavior to check. More work remains to help developers more effectively and easily capture expected behavior.

4.1.9 Building and branching (11)

Should I branch or code against the main branch? (1)
How can I move this code to this branch? (1)
What do I need to include to build this? (3)
What includes are unnecessary? (2)
How do I build this without doing a full build? (1)
Why did the build break? (2)[59]
Which preprocessor definitions were active when this was built? (1)

Developers asked a variety of questions about how to build or use a version control system. Developers contemplating a change wondered if they should start a new branch or keep coding on the main branch. When a bug fix was made for an old version, developers asked how to migrate it to the substantially changed current version. When making a change, developers asked what includes or dependencies to add; when inspecting code, they asked which ones were no longer necessary. Developers sought to learn the minimal number of packages necessary to rebuild, rather than trigger a time-consuming full build. Faced with intermittently breaking builds, developers sought to understand their circumstances. After building, developers wondered what the preprocessor had done – which definitions were active.

IDEs such as Eclipse automatically add and remove include statements in many situations. Helping developers more easily understand build problems and manage changes between branches is an open research area.

4.1.10 Teammates (16)

Who is the owner or expert for this code? (3)[7]
How do I convince my teammates to do this the "right way"? (12)
Did my teammates do this? (1)

When trying to understand unfamiliar or complicated code, developers often try to find an owner or expert. Research tools more directly answer these questions [22]. Other questions expressed developers' 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 outdated style, or writing hacks. No research has yet investigated how consensus on such conventions forms.

4.2 Questions about Elements

4.2.1 Intent and Implementation (32)

What is the intent of this code? (12) [15]
What does this do (6) in this case (10)? (16) [24]
How does it implement this behavior? (4) [24]

Developers ask questions about both what code is supposed to do – intention – and what it actually does. They ask about the intention of code, SQL queries, structures, objects, files, and components. Most "What does this do?" questions were not about everything code does but about a specific situation – an exception or error, a slow or timed-out operation, multiple threads, boot up, or execution on a server farm. Developers ask about how code's behavior maps to specific code – how optimized code implements an algorithm, how an exception could be thrown, how binding work, or how a class implements application functionality. Comments and design documents seek to answer these questions, but again require the questions to be anticipated and an investment in writing and updating them.

4.2.1 Method properties (2)

How big is this code? (1)
How overloaded are the parameters to this function? (1)

4.2.2 Location (13)

Where is this functionality implemented? (5) [24]
Is this functionality already implemented? (5) [15]
Where is this defined? (3)

When considering reuse, developers asked if the functionality they desired was already implemented. Developers faced challenges locating the definition of types referenced in code – picking the definition referenced by include files, understanding type renames done at compilation, and navigating between multiple files defining the same class.

The WhyLine [16] lets users select a user interface element at runtime to find the code implementing its recent behavior. Better search tools for reusing functionality have been designed for reusing APIs (e.g., [11]) but not for reusing code within a codebase.

4.2.3 Performance (16)

What is the performance of this code (5) on a large, real dataset (3)? (8)
Which part of this code takes the most time? (4)
Can this method have high stack consumption from recursion? (1)
How big is this in memory? (2)
How many of these objects get created? (1)

Developers ask questions about performance. 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 what missing functionality could be added.

4.2.4 Concurrency (9)

What threads reach this code (4) or data structure (2)? (6)
Is this class or method thread-safe? (2)
What members of this class does this lock protect? (1)

Developers ask both if multi-threaded behavior is possible – if multiple threads could reach code or a data structure – and if code has already been designed for multithreading. Thread coloring documents and checks expectations about the thread safety of code and threads that may reach it [28].

4.3 Questions about Element Relationships

4.3.1 Contracts (17)

What assumptions about preconditions does this code make? (5)
What assumptions about pre(3)/post(2)conditions can be made?
What exceptions or errors can this method generate? (2)
What are the constraints on or normal values of this variable? (2)
What is the correct order for calling these methods or initializing these objects? (2)
What is responsible for updating this field? (1)

When understanding relationships between methods and callers, developers ask questions about both the assumptions currently made by an implementation and all assumptions that could be made. Assumptions included constraints on parameters such as ordering or size and the possible states a program might be in. Developers asked about what errors or exceptions could be returned and the conditions under which they are generated. Developers also asked about constraints on variables: if it was a 0-based or 1-based index or what constituted typical values.

Assertions and contracts specify and check constraints on both pre- and postconditions (e.g., [3]) and on ordering relationships between methods [26]. Checked Exceptions [10] specify some of the exceptions that can be thrown but not the conditions under which they are thrown. However, contracts only describe assumptions that the original developer decided to express, not all possible assumptions the code makes or that could be made. Moreover, little is known about the assumptions developers make and the ability of existing notations to express them.

4.3.2 Control flow (19)

In what situations or user scenarios is this called? (3) [15][24]
What parameter values does each situation pass to this method? (1)
What parameter values could lead to this case? (1)
What are the possible actual methods called by dynamic dispatch here? (6)
How do calls flow across process boundaries? (1)
How many recursive calls happen during this operation? (1)
Is this method or code path called frequently, or is it dead? (4)
What throws this exception? (1)
What is catching this exception? (1)

When investigating relationships between methods, developers ask questions about control flow such as the situations or user scenarios in which methods are called, how they differ with parameter values, and what parameter values were necessary to reach specific code within a method. Determining calls from a method was challenging in the presence of dynamic dispatch – calls to interfaces, signaling events, or changing bound properties. Developers ask both about how frequently methods are called and if they are never called and dead. Modern development environments provide views for traversing through callers and callees but do not directly answer any of these questions.

4.3.3 Dependencies (5)

What depends on this code or design decision? (4)[7]
What does this code depend on? (1)

In contrast to questions about the implications of a specific change, dependency questions ask about code affected by any possible change to code or a design decision. Design structure matrices let developers express dependencies between decisions [27], but tools to reverse engineer them from code (e.g., [13]) are limited to use relationships between packages or call relationships between methods.

4.3.4 Data flow (14)

What is the original source of this data? (2) [15]
What code directly or indirectly uses this data? (5)
Where is the data referenced by this variable modified? (2)
Where can this global variable be changed? (1)
Where is this data structure used (1) for this purpose (1)? (2) [24]
What parts of this data structure are modified by this code? (1) [24]
What resources is this code using? (1)

Developers ask questions about how data flows through code – where it originates, where it goes, and how it is aggregated, translated, or transformed. Developers ask both about where global variables or data flowing through a variable could be modified and how a piece of code modifies a data structure or uses resources.

Research tools exist to track data back to its original source [25], but do not currently support tracking it forward. We are not aware of any tools that let developers see where data referenced by a variable may be modified or how code interacts with data structures or resources.

4.3.5 Type relationships (15)

What are the composition, ownership, or usage relationships of this type? (5) [24]
What is this type's type hierarchy? (4) [24]
What implements this interface? (4) [24]
Where is this method overridden? (2)

When investigating classes or interfaces, developers ask hard-to-answer questions about relationships between

types. Questions were more challenging for classes spanning modules or projects that were not open in the development environment. Modern development environments depict type hierarchies and in-heritance relationships. UML reverse engineering tools depict composition and usage relationships between types [17]. Ownership systems (e.g., [5]) specify and check ownership relationships.

4.3.6 Architecture (11)

How does this code interact with libraries? (4)
What is the architecture of the code base? (3)
How is this functionality organized into layers? (1)
Is our API understandable and flexible? (3)

When understanding architecture, developers ask questions about interactions between code and libraries or how functionality is organized into layers. Research tools exist to reverse engineer UML sequence diagrams describing interactions between code and other components [17]. Diagrams can help document architecture, but tool support for synchronizing them with evolving code is limited.

5. Limitations

Our study was limited to a single organization and its processes, practices, and conventions. While there is a huge variability amongst teams within Microsoft, other organizations with radically different processes or tools may have different questions. By using a survey to gather a large corpus of questions, rather than direct observations, our data is based on self-reports that are subject to biases in perception of difficulty and what developers are able to recall.

6. Conclusions

A better understanding of developers' information needs may lead to new tools, programming languages, and processes that make hard-to-answer questions less time consuming or error prone to answer. By focusing on real questions that developers ask and experience problems answering, new approaches are more likely to be useful and help developers be more effective.

Acknowledgments

This study was conducted during a visit to the Human Interactions in Programming Group at Microsoft Research. This research was funded in part by NSF grant CCF-0811610 and through the EUSES consortium under NSF grant ITR CCR-0324770.

References

[1] Abadi, M., Flanagan, C., and Freund, S. N. (2006.) Types for safe locking: Static race detection for Java. In *TOPLAS,* 28(2), 207-255.

[2] Anvik, J, Hiew, L, and Murphy, G.C. (2006). Who should fix this bug? In *Proc. of the Int'l Conf. on Soft. Eng. (ICSE).*

[3] Burdy, L., Cheon, Y, Cok, D., Ernst, M., Kiniry, J., Leavens, G.T., Leino, K.R.M., and Poll, E. (2003). An overview of JML tools and applications. In *International Journal on Software Tools for Technology Transfer*, 7(3), 212-232.

[4] Cook, B., Gotsman, A., Podelski, A., Rybalchenko, A., and Vardi, M. (2007). Proving that programs eventually do something good. In *Principles of Programming Languages (POPL).*

[5] Dietl, W. and Muller, P. (2005). Universes: lightweight ownership for JML. In *Journal of Object Technology*, 4(8).

[6] Erickson, C. (2003). Memory leak detection in C++. In *Linux J.*, 110 (Jun. 2003), 8.

[7] Fritz, T., and Murphy, G.C. (2010). Using information fragments to answer the questions developers ask. In *Proc. of the Int'l Conf. on Soft. Eng. (ICSE), 175-184.*

[8] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (2003) *Design patterns: elements of reusable object-oriented software.* Addison – Wesley.

[9] Godefroid, P., Klarlund, N, and Sen, K. (2005). DART: directed automated random testing. In *PLDI.*

[10] Goodenough, J.B. (1975). Exception handling: issues and a proposed notation. In *Communications of the ACM (CACM),* 18(12), 683-693.

[11] Hoffmann, R., Fogarty, J., and Weld, D.S. (2007). Assieme: finding and leveraging implicit references in a web search interface for programmers. In *UIST*, 13-22.

[12] Holmes, R., and Begel, A. (2008). Deep intellisense: a tool for rehydrating information. In *Proc. Mining Software Repositories (MSR).*

[13] Jordan, E., Sangal, N., Sinha, V., and Jackson, D. (2005). Using dependency models to manage complex software architecture. In *Proc. OOPSLA.*

[14] Kamiya, T., Kusumoto, S., and Inoue, K. (2002). CCFinder: a multilinguistic token-based code clone detection system for large scale source code. In *TSE*, 28(7).

[15] Ko, A. J., DeLine, R., and Venolia, G. (2007). Information needs in collocated software development teams. In *ICSE*, 344-353.

[16] Ko, A.J., and Myers, B.A. (2008). Debugging reinvented: asking and answering why and why not questions about program behavior. In *Proc. of the Int'l Conf. on Soft. Eng. (ICSE).*

[17] Kollmann, R., Selonen, P., Stroulia, E., Systä, T., and Zündorf, A. (2002). A study on the current state of the art in tool-supported UML-based static reverse engineering. In *WCRE.*

[18] LaToza, T.D., Garlan, D., Herbsleb, J.D., and Myers, B.A. (2007). Program comprehension as fact finding. In *FSE.*

[19] LaToza, T.D., Myers, B.A. (2010). Developers ask reachability questions. In *Proc. of the Int'l Conf. on Soft. Eng. (ICSE).*

[20] LaToza, T.D., Venolia, G., and DeLine, R. (2006). Maintaining mental models: a study of developer work habits. In *Proc. ICSE.*

[21] Letovsky, S. (1986). Cognitive processes in program comprehension. In *Empirical Studies of Programmers.*

[22] Mockus, A., and Herblseb, J. (2002). Expertise browser: a quantitative approach to identifying expertise. In *Proc ICSE.*

[23] Moran, T. P. and Carroll, J. M., Eds. (1996). *Design rationale: concepts, techniques, and use*. Lawrence Erlbaum Associates, Inc.

[24] Sillito, J., Murphy, G.C., and De Volder, K. (2008). Asking and answering questions during a programming change task. In *Transactions on Software Engineering (TSE)*, 34(4).

[25] Sridharan, M., Fink, S.J., and Bodik, R. (2007). Thin slicing. In *Programming Language Design & Implementation (PLDI).*

[26] Strom, R.E., Yemini, S. (1986). Typestate: a programming language concept for enhancing software reliability. In *Transactions on Software Engineering* (*TSE)*, 12(1), 157-171.

[27] Sullivan, K.J., Griswold, W.G., Cai, Y., and Hallen, B. (2001). The structure and value of modularity in design. In *Proc. FSE.*

[28] Sutherland, D. (2008). The code of many colors: semi-automated reasoning about multi-thread policy for Java. *Dissertation, Carnegie Mellon University.*