On the Importance of Understanding the Strategies that Developers Use

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

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

ABSTRACT
Understanding the strategies that developers use during coding activities is an important way to identify challenges developers face and the corresponding opportunities for tools, languages, or processes to better address the challenges and more effectively support the strategies. After creating a design, evaluation studies often measure task success, time, and bugs to argue that the design improves programmer productivity. Considering the strategies that developers use while conducting these studies increases the likelihood of a successful test and makes the results easier to generalize. Therefore, we believe that identifying strategies developers use is an important goal. Beyond identifying strategies, there are also research opportunities in better understanding how developers choose strategies.

Categories and Subject Descriptors
D.2.6 [**Programming Languages**]: Programming Environments; D2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms
Experimentation, human factors

Keywords
Program comprehension, developer questions, strategies

1. INTRODUCTION
Researchers studying the activities of software developers make use of two types of study: exploratory studies generate ideas for what might make developers more productive, and evaluation studies determine if a particular design succeeds in improving productivity. Compared to the intuition or personal experience of the designer, systematic and detailed exploratory studies can reveal challenges that were unexpected, find frequent but unmemorable problems, and lead to alternative perspectives on developers' work. Evaluation studies build confidence and evidence that a design is usable, help to weed out those that are not, and help to iteratively create designs that overcome discovered shortcomings.

Recently, exploratory studies of coding activities have begun to identify goals, information needs, questions, and strategies used by developers. By identifying strategies developers use and the challenges developers face applying these strategies, key insights for new designs can be generated [17][11]. For example, we found that developers often debug or investigate the implications of changes by searching for target statements across control flow paths through a program [10]. This strategy was challenging when developers had to guess which paths led to targets or when some of the paths were infeasible and could never execute. To address these challenges and better support this strategy, we are designing a tool for searching across paths [11]. Identifying strategies also helps in designing studies to evaluate a design's effectiveness. We plan to design an evaluation study that measures not only task time and success but also how well the tool supports the kinds of searches that developers attempt.

Of course developers do not always debug or investigate code by searching for statements across paths. Sometimes developers may implement a change and test if it works. Or developers may use their knowledge and intuition to guess the effects of a change. Developers may wonder why the original developer did not use a particular design and explore code history for rationale about why the current design was chosen. Or, if the original developer is still on their team and available to be interrupted, they may walk into his or her office and ask. In some situations, developers may choose one of these strategies instead of searching across paths for statements. If we conducted an evaluation study in which one of these other strategies is possible and more effective, we might fail to see any benefits from our tool, as it only supports the search strategy. Thus, understanding not only the strategies that developers use but also the factors that influence when developers choose to use them is an important part of an argument that a tool is useful. Furthermore, understanding these factors makes it easier to design evaluation studies that are most informative.

Recently, there has been growing recognition of the lack of theory in software engineering and the benefits more theory might provide [7]. A theory of coding activity describing factors influencing developers' strategy choices could help fill this gap. Such a theory would describe how developers start from high-level tasks (e.g., fixing a bug, implementing a feature), ask questions to try to determine how to perform those tasks, and choose strategies to try to answer these questions. Like theories in traditional scientific fields, such a theory could have many benefits. First, it would allow sharing knowledge about the space of strategies and factors between similar designs. Second, studies designed specifically to test the theory could be employed. Third, the theory could predict and explain why developers are likely to use the strategy supported by a tool in a specific situation without ever having to conduct an empirical study. Fourth, the theory could help designers identify assumptions made by tools (e.g., information developers need before choosing to employ a strategy). Fifth, strategies and factors could be taught to undergraduates to provide more effective strategies and help them make better strategy choices. Sixth, when conducting studies, the theory focuses attention on the data that is most important to collect: strategies and factors. Finally, the theory makes possible cheaper lab studies with intentional external invalidity. Rather than attempt to recreate every aspect of a professional software development project in the lab, theories help to predict which aspects are necessary to replicate and which are not.

In this paper, we first illustrate the importance of strategies by considering how identifying strategies might make evaluation studies more successful. We then discuss several types of strategies and speculate on some of the factors that might influence the perceived and actual utility of a strategy.

2. DESIGNING EVALUATION STUDIES

One benefit of identifying strategies is to make designing valid and informative evaluation studies easier. Designing these studies is challenging. Participants, tasks, materials, data collection methods, conditions, and measures must all be chosen. Expected differences in the measures may not occur, and the interpretation may not be clear: is the design just not useful, did usability problems hinder its success, or were the tasks or participants poorly chosen to demonstrate its usefulness? Even when differences do occur, skeptics may argue that the findings do not generalize to more realistic conditions. To help overcome all of these problems, it can be beneficial to collect data not only on task time and success, but also on the strategies developers use.

For example, one study tested if alternative concurrency paradigms (transactional memory, actors) helped undergraduate students complete two-hour programming tasks more quickly or in fewer lines of code compared to a traditional lock-based concurrency paradigm [15]. No significant differences in either measure were found, although participants significantly preferred transactional memory over locks. The paper provides a long list of potential reasons for this result. Hoping to avoid large expected differences between subjects, a within-subjects design had been used. But this may have caused learning effects between tasks. Or, maybe the participants did not yet know how to effectively use the concurrency paradigm, or maybe the tasks were too short and trivial for the benefits to be manifested. Maybe the tasks were inappropriate, or the benefits only occur in maintenance rather than writing new code as was tested, maybe the concurrency implementations lacked relevant features of real implementations that affected the outcome, or maybe the code from which developers started was poorly written.

When presenting his paper at the workshop, the author stated that an important lesson he had learned was that collecting more qualitative data was important. While initially hoping to design a study that objectively demonstrated the benefits of a design, he came to believe that the complexity of developer activity necessitates collecting more qualitative data to understand what is happening.

Collecting data on strategies might have helped provide interesting results even without a main effect of the manipulation. What strategies were developers using to reason about concurrency? Did the concurrency paradigm influence which strategies were chosen, or how difficult these strategies were to use? Did individual developers differ in the strategies they used? Why did they choose the strategies they did? What made the strategies difficult to use? How might these difficulties be greater (or smaller) in more realistic situations?

Another study tested if typed or untyped languages make developers more productive [6]. There is a long-standing debate between proponents of untyped ("dynamic") languages (e.g., Perl, Ruby) and typed languages (e.g., Java). The study tested if undergraduate students writing a parser over the course of 27 hours in a new OO language were faster in the typed or untyped variant. Participants took anywhere from 4% to 42% less time in the untyped variant than the typed variant, providing evidence that untyped languages are superior to typed languages. However, due to the strongly held beliefs by proponents, the author reported at the workshop that he had received intense skepticism of the result.

Understanding what developers did and how they used strategies might help understand and generalize such a finding. What did developers do differently when using the typed variant that caused them to take more time? Did it affect the strategies they used or introduce additional work? Did developers not benefit from type checking because runtime errors were just as effective or only because developers inserted few bugs? Were developers always quickly able to run their programs, negating any benefits of using a type checker before a program was complete enough to run?

3. CHOOSING STRATEGIES

We define a "strategy" as a sequence of actions developers use to accomplish a goal. Actions include both physical actions (e.g., opening a method) and mental actions (e.g., remembering the intent of a method). In coding activities, developers select among various strategies to answer the questions necessary to complete their tasks (e.g., fix a bug, implement a feature). These questions and hypotheses about answers form a hierarchy, as developers decompose questions into lower-level questions that are easier to answer with the available methods and tools [18]. A number of studies have investigated the questions developers ask and high-level characterizations of the types of strategies they use to answer them. Developers engage in activities such as reproducing bugs, debugging, proposing changes, investigating the implications of changes, reusing code, implementing changes, compiling, and testing [12][10]. Developers answer questions through these activities, and also by consulting artifacts such as bug reports, email, code histories, specifications, design documents, asking their teammates, or simply remembering the answer [9]. When exploring code, developers seek information, make decisions about which structural relationship (e.g., which method call) to traverse to find information, and collect and organize the answers [8].

Unfortunately, identifying a strategy by which developers answer a question may still fail to explain developers' behavior in this situation. Developers may be able to choose between strategies. In one of our lab studies, developers tried to determine if they could safely remove a call to a method [10]. Most tried to answer this question by examining what the code did or the situations in which it was called. However, one developer simply removed the call and tested to try to identify a behavior change. But, because the behavior change was subtle, he incorrectly believed there was no behavior change. Another developer wished to rewrite the whole section of code, but did not have time to do so. Developers might alternatively have looked through the code history (which was not provided in this case) to determine what bug or feature the line was related to. During the study, developers also often switched strategies when one did not appear to be working.

In the following sections, we explore several types of strategies developers use and factors that may influence their perceived and actual utility.

3.1 Implement & test

Before implementing a change, developers often wonder about its implications or what it might cause to break [16][9][10]. To answer these questions, developers may employ one of several strategies (e.g., explore the code, check the code history, or guess the answer). But once developers know enough to consider a change, they could instead *implement and test* to see if it works and does not break any of the existing functionality.

A number of factors might influence whether developers choose to understand the implications first or implement and test. Clarke [3][17] considers this choice to be influenced by characteristics of an individual developer (his or her *work style)*, which is described using *personas*. In studying how developers use APIs, Clarke found that developers can be categorized into one of three different personas capturing the strategies they tend to use [3]. *Systematic* developers program defensively, make few assumptions, and wish to understand why something works rather than simply make it work. *Pragmatic* developers try first to implement functionality and resort to more systematic and thorough understanding only when just implementing does not work. *Opportunistic* developers eschew a thorough understanding and try to get their code working as quickly as possible. Systematic developers seem likely to understand rather than implement and test, pragmatic developers to implement and test rather than understand (or even not test at all), and pragmatic developers seem likely to understand when it seems necessary.

A second factor likely influencing strategy choice is developers' development process. For example, developers using *Test-Driven Development* (TDD)[1] write unit tests before implementing changes and use these tests to ensure that their changes work. In contrast, many development projects have few tests or tests that are only sufficient to ensure that nothing important broke rather than that everything necessarily works. Proponents of TDD believe that developers using TDD are more likely to use their unit tests to implement and test rather than try to understand implications: "Comparing [TDD] to the non-test-driven development approach, you're replacing all the mental checking and debugger stepping with code that verifies that your program does exactly what you intended it to do" [18].

A third possible factor is how worried developers are about possible, but infrequent or difficult to discover bugs. In domains such as safety critical systems, developers go to great lengths to ensure there are no bugs. In contrast, developers prototyping or working on short-lived code they expect to be thrown away may care little about the potential for bugs. Given that testing only indicates the presence, not the absence of bugs, developers deeply concerned about potential bugs are likely to spend significant time investigating the potential for bugs or even, in the case of safety critical systems, specifically design their systems to make this easier. In contrast, developers who are most interested in getting something running quickly seem likely to prefer to implement and test.

A fourth factor that might influence this choice is the specific situation of the current coding activity: is understanding or implementing and testing easier right now? Is the change quick and easy to implement, or will it require understanding how to reuse some functionality or implementing large or complex functionality? Is anything that might break easily identified and testable, or is it potentially obscure and hidden? Will the tests execute quickly, or will it take days for a regression test suite to finish? Are there properties potentially affected by the change that cannot be tested (e.g., whether the code follows design conventions)?

3.2 Guess the answer

In some cases, developers use their knowledge and intuition to *guess the answer* to a question [9]. Developers have several sources of knowledge they may use to make these guesses: knowledge of the code itself, knowledge of code's intended behavior, and knowledge of idioms, patterns, or architectural styles used in the code. A variety of factors may influence when developers use this knowledge to guess the answer to a question.

In some cases, knowing code conforms to a standard idiom likely helps developers answer questions. For example, for a system using a model / view / controller architecture, developers can reasonably guess that for a given model class there exists a corresponding view class and a mechanism by which changes are propagated. A number of books catalog collections of design patterns and architectural styles. When these patterns are present, developers know they are present, and developers know them, developers could use this knowledge to answer questions.

Code quality likely influences how often developers can guess answers. In situations where the code contains hacks, the code no longer conforms to an underlying pattern or idiom – conformance has been sacrificed to achieve some other goal. In these cases, developers can no longer rely on knowledge of the pattern or idiom to answer their questions. And even if the code in question does not itself contain a hack, developers working in a codebase that often contains hacks might be more reluctant to assume that it works as expected and be more cautious in using their intuition.

As developers gain expertise in programming, they also gain knowledge about typical code idioms and patterns. Traditional studies of expertise have found that chess experts are not inherently smarter: their experience simply helps them recognize typical chess piece configurations [2]. This recognition permits reasoning about chess positions at the higher level of abstraction of configurations rather than pieces. Similarly, software developers comprehend code by recognizing idioms (e.g., iteration over a collection) rather than individual lines of code [4] which helps to answer questions. One study found that developers with more experience were able to answer a rationale question using their intuition that others could not answer by any means [9].

When developers know how an application is supposed to behave, they can use this knowledge to answer some questions about the code. Both how the code is built and how well the developer knows the application's behavior likely influences the use of this strategy. For example, in one of our studies, developers were able to use their knowledge to predict that scrolling should not influence the caret position displayed in a status bar [9]. But this led to a false belief, as the code was misleadingly named. Domain driven design argues that using knowledge of an application's behavior to understand code is so important that code should be specifically designed to maximize the situations in which this strategy is effective [5].

When code is unavailable or has not yet been written, developers can guess answers to questions that cannot be answered by code exploration. Developers implementing features may assume certain behavior is necessary for a feature that will soon be implemented. Or, for functionality exposed in an API, intuition may help developers predict API clients use cases and what changes might be possible without breaking these use cases.

3.3 Other strategies

Developers have many other types of strategies they could use. Developers can *check the code history* – records of line changes and associated checkin messages – maintained by version control systems. We have observed developers using these messages to answer rationale questions by determining the feature or bug that precipitated some seemingly bizarre functionality to be added. Developers often answer questions by *asking a teammate* [13][9]. But this interrupts the teammate and does not work when they are unavailable or busy. And developers often conduct due diligence to attempt first to begin understand a complex issue themselves [13]. Finally, developers use both static investigation (e.g., read the code, use IDE code browser tools) and dynamic investigation (e.g., debugger, logging, tracing) to explore the code.

4. AN EXAMPLE

While writing this paper, the first author observed an example that illustrates strategy choice and using multiple strategies. A developer was wondering why four lines had been commented out. The lines contained functionality she knew would help implement a new feature. But why had they been commented out? She first tried to *guess the answer* about why this code might have caused a bug. As the code had been commented out and not removed, she knew the change was likely a quick hack rather than a well-considered change. But she did not see how the commented code might break anything.

She next *checked the code history* and found that she had herself commented out the lines over 2 years ago. But the change had been committed with several others, so the change log did not suggest why this change had been made. Uncommenting the lines, she used *implement and test* to verify that the functionality did indeed help implement the new feature and that all her tests now passed. But she was still mystified as to why the code had been commented out and worried that something might be broken. Finally, she *asked her teammates* by sending an email to those who had worked on this code. One recalled that the code might not work correctly for some rare input values. Another suggested alternative code that would fix this problem. Having finally answered her question about why the code had been commented out, she used a fixed version of this code.

5. CONCLUSIONS

Understanding the strategies by which developers answer questions holds the potential to both reveal new opportunities for tools and to make it easier to understand how and why developers use the tools they do. We believe that a theory of developer activity describing how developers choose strategies could make understanding strategies in these studies easier.

6. ACKNOWLEDGMENTS

This research was funded in part by the National Science Foundation, under grant CCF-0811610. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect those of the National Science Foundation.

7. REFERENCES

[1] Beck, K. (2002). *Test-Driven Development*. New York: Addison-Wesley.

[2] Chase, W. G. & Simon, H. A. (1973). Perception in chess. In *Cognitive Psychology*.

[3] Clarke, S. (2004). Measuring API Usability. In *Dr. Dobbs Journal*, S6-S9.

[4] Détienne, F. (1990). Program understanding and knowledge organization: the influence of acquired schemata. In *Cognitive Ergonomics: Understanding, Learning and Designing Human- Computer Interaction,* 245-256.

[5] Evans, E. (2003). *Domain driven design*. New York: Addison-Wesley.

[6] Hanenberg, S. (2009). What is the impact of static type systems on programming time? In *Proc. PLATEAU Workshop* at *OOPSLA*.

[7] Hannay, J. E., Sjberg, D.I.K., and Dybå, T. (2007). A systematic review of theory use in software engineering experiments. In *Transactions on Software Engineering (TSE),* 33(2), 87-107.

[8] Ko. A. J., Myers, B.A., Coblenz, M. & Aung, H. H. (2006). An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. In *Transactions on Software Engineering (TSE)*, 32(12).

[9] Ko., A. J., DeLine, R. & Venolia, G.. (2007). Information needs in collocated software development teams. In *Proc. Int'l Conf. Software Eng (ICSE)*.

[10] LaToza, T.D., & Myers, B.A. (2010). Developers ask reachability questions. In *Proc. Int'l Conf. Software Eng (ICSE).*

[11] LaToza, T.D., & Myers, B.A. Searching across paths. In *Proc. of the Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation, ICSE.*

[12] LaToza, T.D., Garlan, D., Herbsleb, J.D., & Myers, B.A. (2007). Program comprehension as fact finding. In *Proc. ESEC/FSE.*

[13] LaToza, T.D., Venolia, G., and DeLine, R. (2006). Maintaining mental models: a study of developer work habits. In *Proc. Int'l Conf. Software Eng (ICSE)*.

[14] Llopis, N. (20 February 2005). Stepping through the looking glass: test-driven game development (part 1). In *Games from Within*. Accessed 1/27/2010. Available at http://gamesfromwithin.com/stepping-through-the-looking-glass-test-driven-game-development-part-1.

[15] Luff, M. (2009). Empirically investigating parallel programming paradigms: a null result. In *Proc. PLATEAU Workshop* at *OOPSLA*.

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

[17] Stylos, J., & Clarke, S. (2007). Usability implications of requiring parameters in objects' constructors. In *Proc. Int'l Conf. Software Eng (ICSE).*

[18] Vans, A.M., von Mayrhauser, A., & Somlo, G. (1999). Program understanding behavior during corrective maintenance of large-scale software. In *Int'l J. Human-Computer Studies*, 51(1), 31-70.