A Sketch of the Programmer's Coach: Making Programmers More Effective

David Shepherd and Gail Murphy
Department of Computer Science
University of British Columbia
{dshepher, murphy}@cs.ubc.ca

ABSTRACT

As programmers work on source code, they ask an array of questions that are difficult to answer manually. To help answer these questions, programmers often employ software tools; often in attempting to use these tools, the programmers encounter many obstacles which frustrate their efforts and lead to less than optimal tool utilization. Possibly worse, programmers often intentionally under utilize available tools as they prefer to answer questions only with tools they have used before. We hypothesize that we can coach programmers towards a more systematic use of appropriate software tools that would enable the programmers to be more productive in the completion of their work. We propose to use activity logs collected automatically to deduce the questions a given programmer asks a frequently and then to coach the programmer automatically on appropriate, possibly unfamiliar, tools to answer those questions more effectively. By using activity logs to inform coaching decisions, our approach is based on an objective cost metric. We envision an environment that enables a programmer to learn how to use appropriate tools systematically.

Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Programming Environments

General Terms

Human Factors

Keywords

programmer activity, coaching

1. INTRODUCTION

Throughout the workday, a programmer performs many tasks as he works with a set of software artifacts. During the course of these tasks, a programmer typically poses a series of questions, such as "How is this program element defined or declared?" or "Is there a precedent or example for this code?" [14]. Answering these questions manually by looking through code is often tedious and time-consuming causing programmers to seek the assistance of software tools [7, 16]. If the programmer is aware of the appropriate tool to use, a question becomes easy to answer. For instance, to answer the question "How is this program element defined or declared?" a programmer working in the Eclipse development environment can use the "Open Declaration" command, which automatically finds and displays the definition of the given program element. In other cases, an appropriate tool to answer the desired question may not be available or may not be known to the programmer; the programmer must then transform the question to fit familiar tools, alter the question to fit available tools or abandon that particular line of inquiry. Even if an appropriate tool is located in a search, the programmer may be unaware of how to use the tool correctly. The large number of available tools, the difficulty of mapping from a question to a tool and the knowledge required to use each tool can overwhelm a programmer who is already occupied with the more important task of answering their initial question [6]. These factors often cause a programmer to employ a non-optimal, ad-hoc question answering strategy.

As discussed, it is difficult for programmers to obtain answers to their questions, yet there is another common cause of sub-optimal behavior, the programmers themselves. Programmers are focused on finishing the task at hand and they focus on answering pressing task-related questions, not on learning the functionality of the available tools. Among other non-optimal tendencies, we have observed programmers overusing familiar tools even when the tool provides a poor or incomplete answer to a given question because they are not aware that a more appropriate tool exists. Worse, programmers will often *intentionally* use familiar, but sub-optimal tools, because they have a subjective measure of cost-effectiveness [6]. Namely, they believe the cost of finding, learning, and using a new tool is unjustified in the context of the question at hand. This "production paradox" may cause programmers to answer some individual questions faster but their systematic avoidance of new tools eventually leads to wasted time [4].

We hypothesize that we could help a programmer answer questions more cost-effectively by coaching the programmer towards a more systematic, informed use of available tools. Our intended approach consists of the following steps:

1. Monitor a programmer's activity and automatically identify the questions the programmer most commonly struggles to answer.

2. Use a pre-defined mapping of known, commonly asked questions [14] to match the questions with an appropriate tool.

3. Briefly train the programmer on the use of that tool, using an example from the programmer's activity log.

4. Automatically coach the programer to use the tool in appropriate situations during his normal work flow.

Because we are monitoring a programmer's activity we can present compelling data, in terms of saved time, on the benefits to the programmer if he begins using the tool. This hard evidence should give the programmer confidence in the cost-effectiveness of using the given tool.

The expected contributions of this work are:

- We introduce a technique for interpreting a programmer's activity to infer the questions the programmer is seeking to answer.

- We create a framework for coaching the programmer towards more systematic *behavior*, i.e., tool use.

2. MOTIVATION

Recently our lab has conducted several studies of detailed activity logs from programmers [13, 10]. In these studies, we analyzed traces of programmers' interactions with a development environment collected automatically as the programmers worked. The traces analyzed were recorded using the Mylyn Monitor an Eclipse plug-in, that records each interaction a programmer has with the Eclipse development environment [1]. The monitor records each selection, edit and command performed by a programmer. For each of these interactions, the monitor records a timestamp, the kind of interaction, and the target of the interaction, such as a particular program element that was selected.

Our informal observations during these studies support the notion that many programmers are under-utilizing available tools. We observed several programmers that, in spite of the sophisticated navigation tools included in Eclipse, typically used the text search command to navigate Java files. This caused them to take more steps and spend more time to complete navigation tasks than if they had used available navigation tools. We also observed several programmers that spent significant time trying to reopen a class they had previously visited. Again, they needlessly wasted time and effort, as Eclipse provides an "Open Type" tool to automate this search.

During these studies, we also observed that each programmer employed a unique, small set of tools to answer most of their questions, even when more appropriate tools existed. This observation is consistent with the common belief that claims only 20% of application features are used by each user [2]. In our observations, using inappropriate tools to answer a question often caused the programmer to spend more time manually supplementing the answer or filtering the answer himself. A programmer that wants to view a method definition when viewing its call-site can use several different commands (e.g., "Open Declaration" or "Find References"). If the programmer prefers the "Find References" command then he will have to sort through false positives in the results, whereas with the "Open Declaration" command he will not. A programmer's tendency to rely too heavily on specific tools can lead to sub-optimal behavior.

[1]www.eclipse.org, verified January 2008

[2]http://www.joelonsoftware.com/articles/fog0000000020.html, verified January 2008

In our observations, we have seen that sub-optimal tool use can cost the programmer in terms of time and effort, yet we have yet to discuss the potential benefits of changing the programmer's behavior. Intuitively, it seems clear that correct use of a tool can lead to more effective programming. Our informal observations give further credence to this point. In a post-mortem analysis of the data from a study of programmers completing several maintenance tasks [13] we saw that programmers who used the navigational tools more often were more effective at completing the tasks. A study by Robillard et al. [12] finds that more effective programmers tend to use a methodical, structural approach when investigating code. These programmers tend to use navigation tools more frequently than ineffective programmers.

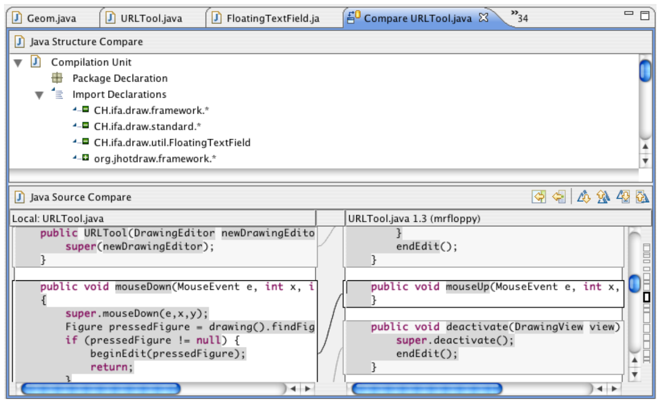
3. COMMON DIFFICULTIES

Our observations suggest that programmers are having difficulty answering questions effectively. These obsevatiosn have lead us to several hypotheses about why programmers are struggling to answer questions effectively. We present each hypothesis along with a concrete illustrative example.

Hypothesis 1: Programmers have difficulty mapping their questions onto an appropriate tool. In many situations, programmers pose a question that is not easily mapped onto a familiar tool. A programmer faced with the question, "What is the impact of changing this program element?", will likely have difficulty mapping this question onto a specific tool. In Eclipse, the programmer could consider running the JUnit test cases associated with the program element changed or the programmer could use structural navigation tools, such as the call graph hierarchy, to approximate the impact manually. Neither tool is directly intended to answer the original question and thus could provide incomplete or incorrect results. For instance, using the call graph to calculate impact could omit important data-flow relationships.

Hypothesis 2: Programmers have difficulty finding tools. In some situations, programmers can pose a question that maps directly onto a tool that likely exists. Consider the question, "What methods call this method?" A programmer that encounters this question can safely assume that a modern programming environment provides a call graph tool. However, finding this tool can still be difficult. A programmer trying to find the call graph tool in Eclipse might search for "call graph" in Eclipse's help search. This search returns no hits related to the Call Hierarchy View, which is Eclipse's call graph tool. Furthermore, because Eclipse has many features manually searching menus for the tool is prohibitively expensive.

Hypothesis 3: A programmer can only use a tool effectively if the programmer can assess the limitations and benefits of using the tool to answer a particular question. To use a tool effectively a programmer needs meta-knowledge of the tool, such as the tool's benefits and limitations. For instance, a programmer faced with the question, "How can I refactor this block of source code B into a new method?" might consider using the "Extract Method" refactoring command. If B does not meet strict pre-conditions then B cannot be refactored via this command and the programmer could have completed this task faster if done manually. However, the benefit of using this command, when possible, is time savings and correctness. The programmer not only has to map a question to a tool and know how to use the tool, but also has to track meta-knowledge about each tool and conduct a split-second cost-benefit analysis prior to tool invocation. These burdens may discourage a programmer from employing tools to answer questions.



```
    public URLTool(DrawingEditor newDrawingEdito
        super(newDrawingEditor);
    }

    public void mouseDown(MouseEvent e, int x, i
    {
        super.mouseDown(e,x,y);
        Figure pressedFigure = drawing().findFig
        if (pressedFigure != null) {
            beginEdit(pressedFigure);
            return;
```

```
        }
        endEdit();
    }

    public void mouseUp(MouseEvent e, int x,
    }

    public void deactivate(DrawingView view)
        super.deactivate();
        endEdit();
    }
```

Figure 1: The file compare view in Eclipse

Hypothesis 4: A programmer must be able to interpret the results of the tool. A programmer with the question "How has this file changed?" would likely seek the use of a tool to answer the question. In Eclipse, a programmer can use the Compare View, which many would consider a simple tool. In Figure 1, we show the Compare View comparing the file URLTool with an older version of itself. Even this simple view presents a lot of information and learning to interpret its results correctly can require valuable time. The links between code segments on the left and right are particularly confusing to interpret. One might assume they link the same code segments in the old and new version of the code but in this figure they link methods mouseUp and mouseDown and have no obvious meaning [3]. This learning curve discourages programmers from using unfamiliar tools.

4. OUR APPROACH

As a first step towards coaching programmers to more systematic behavior, we propose creating a tool that coaches a programmer's use of navigation tools. We chose to focus on navigation tools because, based on our manual study of activity logs, programmers spend a large percentage of time navigating code, often sub-optimally. Other studies have also shown that programmers spend a large percentage of their time navigating [11].

Our tool, the coach, will leverage automatically collected programmer activity logs as well as program information to infer what questions the programmer has been trying to answer. In Figure 2, we show the flow of information within coach. The Mylyn Monitor logs all programmer activity in Eclipse and records that data to a log. At the same time, the Java Analyzer component collects program structure information (i.e., calls, uses, and their inverses) from the programmer's workspace. Both of these inputs flow into the Pattern Analyzer that scans for known non-optimal usage patterns. The development of the Pattern Analyzer will be one of our major research contributions.

[3]The button labeled "Ignore whitespace where applicable" located outside of the Compare View with a confusing icon rectifies this mismatch, but the button is difficult to locate.

Given programmer activity information and knowledge about the program structure deducing a subset of a programmer's questions becomes feasible using (often) simple pattern matching. The program structure information gives insight into the possible goals of a programmer and the activity log records how the programmer reached that goal. Consider a programmer who wishes to open the declaration of class A that is used in method B. If the programmer does not know the "Open Declaration' command (which opens the declaration of the highlighted class), the programmer is likely to follow predictable behavioral patterns to find A's declaration. The programmer will often click in the file system explorer (i.e., the Package Explorer View) several times until the package which contains A is found. The Pattern Analyzer can detect that the programmer began a search in method B and ended at class A, which was used in method B. If the programmer's trace exhibits this pattern several times, it is evidence that the programmer does not know the "Open Declaration" command, which would have taken the programmer from B to A much faster.

If a sub-optimal pattern occasionally occurs in a programmer's trace then no coaching occurs. However, if the Pattern Analyzer finds many of the same, sub-optimal patterns occurring in a programmer's trace a coaching session will occur. The Pattern Analyzer popup a non-blocking dialog presenting the potential savings in time per day and the ability to learn more, as shown in Figure 3. If the user should choose to learn more he will be shown how he could have used the tool in his workflow, using data from his log as an example. After this session the Pattern Analyzer will monitor the programmer's behavior and provide reminders to use the tool when it automatically identifies relevant situations. Both the initial popup and subsequent reminders are presented as non-modal dialogs in the lower right-hand side of the environment so as not to interrupt the programmer's workflow.

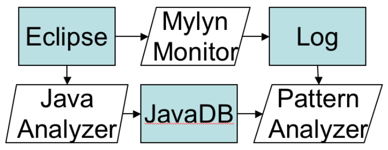
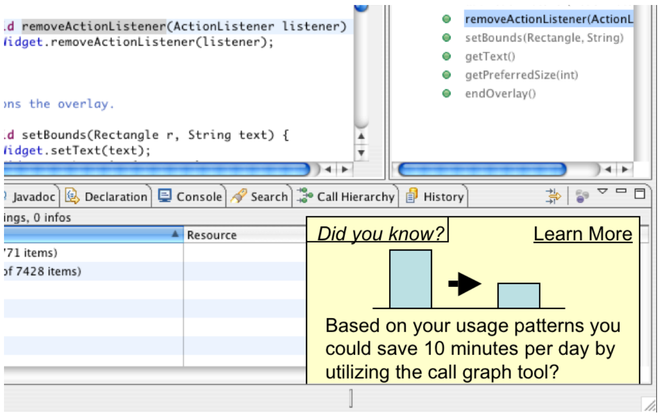


Figure 2: The flow of information for the tool

5. RELATED WORK

The face of our tool will be the programmer's personal coach. A variety of intelligent, automated feedback agents already exist, including intelligent tutoring systems [15] and software assistants, such as the Microsoft Office Assistant. Our coaches will take a history-based approach. Unlike other mentors and assistants which attempt to infer activities from the current state of the application our coach will infer activity from observing the programmer over a long period of time. Our coach has two advantages over previous mentors, ample processing time and a long history to verify his findings. Our mentors will take a critiquing approach,



```
.d removeActionListener(ActionListener listener)
/idget.removeActionListener(listener);


ons the overlay.

.d setBounds(Rectangle r, String text) {
/idget.setText(text);
```

```
removeActionListenerActionL
setBounds(Rectangle, String)
getText()
getPreferredSize(int)
endOverlay()
```

Based on your usage patterns you could save 10 minutes per day by utilizing the call graph tool?

Figure 3: A mockup of the non-modal coaching dialog

which Fischer describes as "the presentation of a reasoned opinion about a product or action" [5]. A key difference between our coach and an expert system is that the coach critiques the programmer's actions unprompted whereas the expert system can only help the programmer change a behavior if he seeks the system's help.

Several systems give programmers constructive feedback on part of their software process, but these systems usually focus on the process artifacts. ArgoUML [17], for instance, provides constructive criticism on the quality of UML designs, inserting to-do items with suggestions on problematic design features. Eclipse's incremental compiler provides visual clues that source code is not compiling as well as suggestions as to how to fix the issue. Many other tools can provide constructive feedback on software artifacts [1, 3].

The Lisp-Critic tool [6] is closest to our work, especially in terms of motivation. Its purpose is to provide constructive and analytic feedback to programmers about their LISP code. Similar to our work, it is motivated by the observed fact that programmers, even experts, tend to use only a small set of available features when programming. However, features in our work correspond to software tools whereas features in Lisp-Critic correspond to LISP functions, Lisp-Critic is passive whereas our coach is active, and Lisp-Critic is driven by rules while our coach is driven by pattern-matching. Lisp-Critic also differs from our work because it operates on software artifacts, whereas we seek to effect programmer behavior.

Researchers have long known the value of process visibility. If programmers are aware of the state of their project and their personal progress then they are able to adjust their behavior to perform better. Work in this area has been done on process-centered software-engineering environments (PSEEs) such as [2]. This work has not been successful because of their inflexible process model and cumbersome process documentation requirements. The Personal Software Process, while more widely adopted, also has the same drawbacks, namely it requires the programmer to self-document much of his process [8]. Our coach, in contrast, relies on automatically collected information with no burden on the programmer.

Finally, researchers have begun to automate the tedious information collection process associated with some of the above techniques [9]. Thus far, this work has focused on team-wide process improvement whereas our work focuses on individual optimization.

6. REFERENCES

[1] C. Artho. Jlint – find bugs in java programs. *http://jlint.sourceforge.net/*, 2006.
[2] Naser S. Barghouti. Supporting cooperation in the marvel process-centered sde. *SIGSOFT Softw. Eng. Notes*, 17(5):21–31, 1992.
[3] Oliver Burn. Checkstyle 4.4. *http://checkstyle.sourceforge.net/index.html*, 2008.
[4] John M. Carroll and Mary Beth Rosson. Paradox of the active user. *Interfacing thought: cognitive aspects of human-computer interaction*, pages 80–111, 1987.
[5] G. Fischer, A. C. Lemke, T. Mastaglio, and A. I. Morch. The role of critiquing in cooperative problem solving. *Trans. on Information Systems*, 1991.
[6] Gerhard Fischer. A critic for lisp. In *IJCAI*, pages 177–184, 1987.
[7] Reid Holmes, Robert J. Walker, and Gail C. Murphy. Strathcona example recommendation tool. In *Foundations of Softw. Eng.*, pages 237–240. ACM, 2005.
[8] Watts S. Humphrey. Introducing the personal software process. *Ann. Software Eng.*, 1:311–325, 1995.
[9] Philip M. Johnson. Requirement and design trade-offs in hackystat: An in-process software engineering measurement and analysis system. In *Empirical Softw. Eng. and Measurement*, pages 81–90. IEEE, 2007.
[10] Mik Kersten and Gail C. Murphy. Using task context to improve programmer productivity. In *Foundations of Softw. Eng.*, pages 1–11. ACM, 2006.
[11] Andrew J. Ko, Htet Aung, and Brad A. Myers. Eliciting design requirements for maintenance-oriented ides: a detailed study of corrective and perfective maintenance tasks. In *Int. Conf. on Softw. Eng.*, pages 126–135. ACM, 2005.
[12] Martin P. Robillard, Wesley Coelho, and Gail C. Murphy. How effective developers investigate source code: An exploratory study. *IEEE Trans. Softw. Eng.*, 30(12):889–903, 2004.
[13] Izzet Safer and Gail C. Murphy. Comparing episodic and semantic interfaces for task boundary identification. In *Conference of the Center for Advanced Studies on Collaborative Research*, pages 229–243. ACM, 2007.
[14] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. Questions programmers ask during software evolution tasks. In *Foundations of Softw. Eng.*, pages 23–34. ACM, 2006.
[15] Etienne Wenger. *Artificial intelligence and tutoring systems*. Morgan Kaufmann Publishers Inc., 1987.
[16] Yunwen Ye, Gerhard Fischer, and Brent Reeves. Integrating active information delivery and reuse repository systems. In *Foundations of Softw. Eng.*, pages 60–68. ACM, 2000.
[17] Mei Zhang. Argouml. *J. Comput. Small Coll.*, 21(5):6–7, 2006.