From Quick Fixes to Slow Fixes: Reimagining Static Analysis Resolutions to Enable Design Space Exploration

Titus Barik, Yoonki Song, Brittany Johnson, and Emerson Murphy-Hill
Computer Science Department
North Carolina State University, USA
tbarik@ncsu.edu, ysong2@ncsu.edu, bijohnso@ncsu.edu, emerson@csc.ncsu.edu

***Abstract*—Quick Fixes as implemented by IDEs today prioritize the speed of applying the fix as a primary criteria for success. In this paper, we argue that when tools over-optimize this criteria, such tools neglect other dimensions that are important to successfully applying a fix, such as being able to explore the design space of multiple fixes. This is especially true in cases where a fix only partially implements the intention of the developer.**

In this paper, we implement an extension to the FindBugs defect finding tool, called FIXBUGS, an interactive resolution approach within the Eclipse development environment that prioritizes other design criteria to the successful application of suggested fixes. Our empirical evaluation method of 12 developers suggests that FIXBUGS enables developers to explore alternative designs and balances the benefits of manual fixing with automated fixing, without having to compromise in either effectiveness or efficiency. Our analytic evaluation method with six usability experts identified trade-offs between FIXBUGS and Quick Fix, and suggests ways in which FIXBUGS and Quick Fix can offer complementary capabilities to better support developers.

I. INTRODUCTION

Static analysis tools like FindBugs [1] and Clang [2] can help developers improve software code quality. Research on these tools has primarily focused on finding new types of defects, such as coding errors, style violations [3], and security vulnerabilities [4]. In our previous study, developers indicated that the difficulty of understanding static analysis tool notifications is a considerable barrier to usage, and that having the tool offer automated suggestions for resolving the identified defects would ameliorate this difficulty [5]. Moreover, developers suggested that automatically *fixing* software defects is as important to improving software quality as *finding* them.

The typical ways tools support fixing defects, if they support automatic fixes at all, is by offering a one-shot, *non-interactive* resolution. A one-shot approach is one in which the tool requires no developer intervention to resolve the defect after initial selection. As one example, Eclipse [6], ReSharper [7], and other environments and tools implement such a strategy through the use of *Quick Fixes*. A Quick Fix provides a list of suggestions along with a code preview or description of the resolution. For instance, the "Add final-modifier to field" Quick Fix detects occurrences of fields that are potentially missing the `final` property. The resolution of this fix is straightforward and even self-describing: add the `final` keyword to the corresponding field. Applying this fix quickly silences analysis tools that complain about non-final fields.

But just silencing the static analysis tool is often insufficient. It's also important that the developer has the opportunity to *evaluate the design space of alternative fixes*, which entails both understanding the impact of the change introduced by the fix and giving the developer the ability to influence the fix in the event that the automatic fix pattern only partially supports their intentions [8], [9]. Such resolutions with a non-trivial design space are common; when we randomly inspected 131 of the bug defect patterns available within FindBugs, we found that 80 of them (61%) had more than one plausible change that would resolve the defect.

Just because the design space of static analysis resolutions is very large does not mean that the developer must resolve the problem manually. The IDE can still help by providing (and withdrawing) relevant tools at the right place and time, by ensuring that the interactions between different tools work cohesively, and by leveraging tools' capabilities while simultaneously respecting developers' intentions. How, then, do we balance the automation provided by development tools with developers' need to explore the design space?

In this paper, we present FIXBUGS, an alternative to traditional Quick Fixes for resolving static analysis warnings. Through FixBugs, we advocate the idea of *slow fixes*, which draw on the analogy of *slow food*, a reactionary movement against fast food [10]. Likewise, the design philosophy of FixBugs rejects the notion that speed is the top priority when maintaining software.

The two main contributions of this paper are:

- An implementation of the slow fix idea through FIXBUGS, a tool designed to help developers.
- Two studies of FIXBUGS that demonstrate the benefits and trade-offs of our approach, compared to manual fixing and Quick Fixes.

II. MOTIVATING EXAMPLE

We illustrate some of the limitations of existing one-shot Quick Fix approaches when the tool does not facilitate fix exploration. Consider a hypothetical developer, Kevin, who applies a defect finding tool on source code from Apache Ant [11]. FindBugs detects a Catch Exception defect and

places a bug marker to indicate the defect location to him, shown in Figure 1. Specifically, this `catch` block is considered "dodgy code" because the construct accidentally catches `RuntimeException` as well, masking potential defects.

He then clicks on the marker to invoke Quick Fix for the suggested solutions. Quick Fix, through FindBugs, offers three suggestions, but since Kevin wants a combination that includes both throwing and catching exceptions, none of the available options are entirely suitable.

Because the tool is not designed for exploration, Kevin must fix the defect manually, or select one of the existing options that appears closest to his desired solution. Unfortunately, this comparison is tricky, because the code preview has different formatting and occludes the original code in the editor. Thus, Kevin selects the first suggestion, visually identifies what the fix has changed in his source code, and then manually manipulates the text to mold the transformed code into his desired code.

In contrast to a one-shot tool, a tool that allows for fix exploration would continue to provide Kevin with support to correct the defect, even if the tool is unable to identify a fix that is immediately satisfactory to Kevin.

III. FixBugs Workflow

In this section, we contrast the interaction from the existing Quick Fix feature with that of FixBugs. To do so, let us consider another hypothetical developer, Julia, who must resolve the same Catch Exception defect as Kevin but has FixBugs available to her in the IDE. As with Quick Fix, Julia initiates the tool by clicking on the bug marker icon in the gutter. From the list of options, she then selects the option to open FixBugs.

From this point forward, the interaction with FixBugs is substantially different from Quick Fix. When FixBugs is invoked, the tool immediately opens a popup with suggestions and applies a default suggestion (Figure 2). She decides that the default suggestion isn't satisfactory, so she abstains from pressing the close button on the popup, which would exit FixBugs. In this case, no other suggestions are present, and so Julia uses the drag-and-drop structured editing mode to move some of the exceptions to other valid drop targets. For example, when she drags the `SecurityException` from the `catch` block and drops it after the method signature, FixBugs will automatically fill in syntactically correct `throws SecurityException` code snippet to support her intentions.

During the task, she also notices a `TODO` comment which she does not want (Figure 3). While still within the FixBugs tool, she proceeds to delete this comment. FixBugs temporarily suspends its fix to allow her to make these edits. When finished, she resumes the task, and FixBugs offers a best-effort to support resumption. If she has changed the code significantly, resumption is not always possible. In that case, Julia may return to any of the previous known-good states, using the suggestion popup.

In short, by supporting design exploration in FixBugs, Julia is supported not only for the initial fix, but until the task is successfully completed. In addition, the tool enables her to explore multiple options, and understand the impact of the fix through color annotations before accepting the change, even when she has deviated from one of the prepackaged fixes.

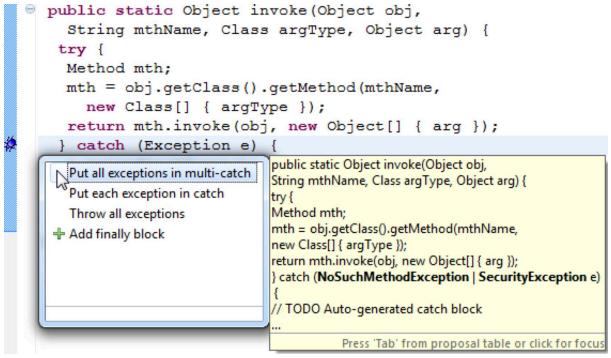

Fig. 1. The original Quick Fix feature provided by Eclipse.

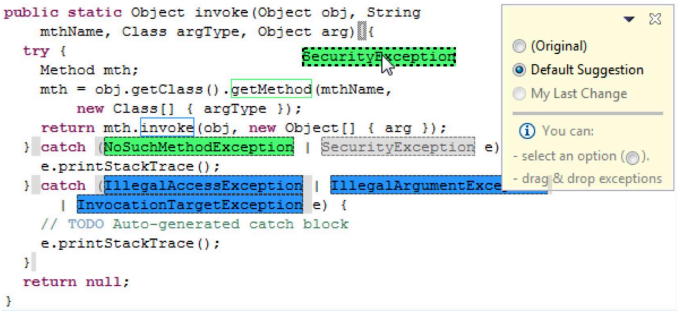

Fig. 2. Suggested fix for the Catch Exception defect in Figure 1. FixBugs allows developers to throw or to catch exceptions by supporting the drag-and-drop action. Here, a developer drags a `SecurityException`. FixBugs displays drop zones in light grey and highlights the closest drop zone with a thicker border.

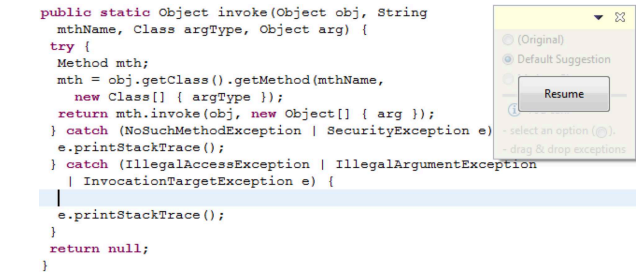

Fig. 3. A developer makes edits to the source code while in FixBugs. FixBugs will temporarily suspend until the developer presses 'Resume'.

IV. Approach

In this section, we describe our approach to implementing fix exploration in a tool we designed and built for the Eclipse development environment, called FixBugs. FixBugs is an extension of FindBugs, a static analysis tool for Java and detects defects. Of the defects detected by FindBugs that do not have a single solution, we highlight three that demonstrate the benefits of enabling design exploration: Catch Exception, String Concatenation, and Open Stream Exception Path.

A. Catch Exception

Figure 2 contains code that demonstrates the Catch Exception defect. To resolve this defect, for Java 7 and onwards FIXBUGS first suggests multi-catch [12] as a default suggestion, which catches multiple exceptions of different types in a catch clause and handles the exceptions in the same way. For Java 6 and earlier, a list of single catch clauses are suggested by default.

FIXBUGS uses colors to map method invocations to their respective thrown exceptions. Using the mapping, developers can associate the invoke method with the catching of IllegalAccess-, IllegalArgument-, and InvocationTarget-exceptions. Although not demonstrated in Figure 2, if two or more method invocations throw the same exception, then FIXBUGS presents the exception in a separate catch clause with a dark gray highlight.

Suggestion Popups. As shown in Figure 2, when the developer invokes FIXBUGS, a movable suggestion popup opens that presents prepackaged solutions to use while fixing the defect. The "(Original)" option is for reverting to the original *buggy* code, which enables comparison of the original code with the modified code. FIXBUGS enables "My Last Change" when the developer performs drag-and-drop at least once. This option helps the developer revert to their changes made while using FIXBUGS if they want to compare them to, say, a prepackaged fix.

To provide a default suggestion for the Catch Exception defect, and other defects in this paper, we mined 24 open source projects. The data mining revealed that developers, when dealing with exceptions, append a throws clause to the method signature 45.1% (4,462) of the time and catch the exception the remaining 54.9% (5,424) of the time. Consequently, adding a catch clause is FIXBUGS' default suggestion.

Drag-and-drop. FIXBUGS continues to support the developer if they are not satisfied with the prepackaged fixes. The tool allows them to use drag-and-drop to visually manipulate where the exceptions should be handled as demonstrated in Figure 2. FIXBUGS displays a dotted border on each exception to indicate that exceptions are draggable and to differentiate between non-draggable items, such as method invocations. When the developer begins to drag an exception, FIXBUGS displays allowable drop zones with light grey boxes, and dynamically highlights the closest drop zone to the current location of the mouse cursor. After the developer drops the exception on a preferred drop zone, the drop zones disappear and FIXBUGS automatically applies the code change. The exception colors remain until the developer indicates they have finished.

B. String Concatenation

Figure 4 shows a getAddress method from log4j [13]. The method contains a for statement (Lines 8 through 13) and a String object (Line 7) that is concatenated in the loop. This code is inefficient because string concatenations in a loop create many temporary objects that must be garbage collected [14]. To improve efficiency, FindBugs suggests

```
1  public String getAddress() {
2   String result = null;
3   try {
4    InetAddress addr = InetAddress.getLocalHost();
5    byte[] ip = addr.getAddress();
6    int i = 4;
7    String ipAddr = "";
8    for (byte b : ip) {
9     ipAddr += (b & 0xFF);
10    if (--i > 0) {
11      ipAddr += ".";
12    }
13   }
14    result = ipAddr;
15  } catch (UnknownHostException e) {
16   e.printStackTrace();
17  }
18  return result;
19 }
```

Fig. 4. Code containing the String Concatenation defect. The getAddress method has a for loop and a variable of ipAddr that is concatenated in the loop.

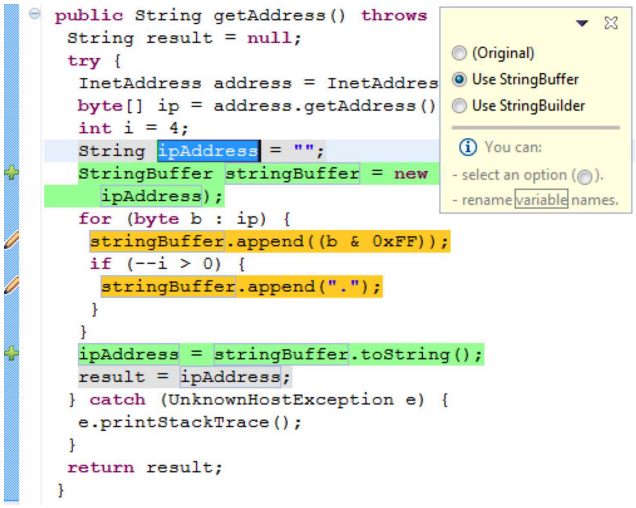

Fig. 5. Suggested fix for the String Concatenation defect shown in Figure 4. In top right is the suggestion popup. Color annotations help developers determine the impact of the change on their code.

using a StringBuffer or StringBuilder. The primary difference between StringBuilder and StringBuffer is that StringBuilder is not synchronized.

As shown in Figure 5, FIXBUGS provides a default suggestion to use StringBuffer. Mining our open source corpus, we found that StringBuffer is used more often (70.1%) than StringBuilder (29.9%).

FIXBUGS first finds the String variable being concatenated in a loop and creates a StringBuffer variable right before the loop. Then, FIXBUGS replaces all concatenations in the loop with the append method before copying the value of the StringBuffer to the original String.

Color Annotations. FIXBUGS is designed to support change comprehension after selecting a resolution. Specifically, FIXBUGS applies color annotations directly on the editor to denote the parts of code that have been added or changed. In addition, the gutter shows a green plus icon to indicate added

```
1  public void read() {
2   try {
3    InputStream in = new FileInputStream("f.txt");
4    Preferences.importPreferences(in);
5    in.close();
6   } catch (IOException e) {
7     e.printStackTrace();
8   } catch (InvalidPreferencesFormatException e) {
9     e.printStackTrace();
10  }
11 }
```

Fig. 6. Code containing the Open Stream Exception Path defect.

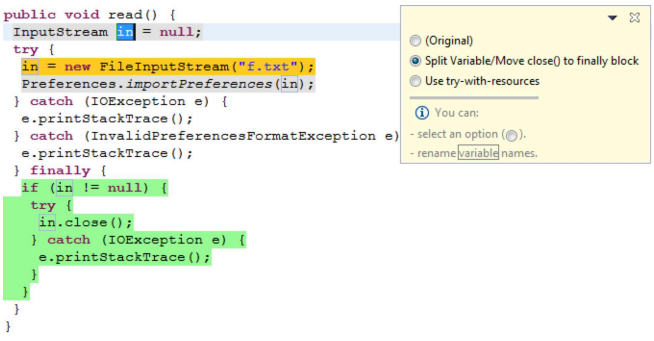

Fig. 7. Suggested fix for the Open Stream Exception Path defect shown in Figure 6.

code, and a pencil icon to indicate a section of changed code.

C. Open Stream Exception Path

Figure 6 shows an example of FindBugs' Open Stream Exception Path defect. The invoked `importPreferences` method (Line 4) may throw `InvalidPreferencesFormatException` before executing the `close` method (Line 5). If this exception is thrown, the stream will not be closed, and a file descriptor leak will occur. It is better to create a `finally` block and call the `close` method within that block to ensure that the stream object is closed. Java 7 has an additional approach to fix this defect, known as "`try`-with-resources [15]", for cleaner resource management. The solutions are illustrated in Figure 7, with the Split/Variable fix shown.

As with the String Concatenation defect, with this defect FIXBUGS also supports fix design exploration using suggestion popups and color annotations. Note that the tool does not provide String Concatenation or Open Stream Exception Path with drag-and-drop, because the feature is not relevant to solving either of these defects.

V. EMPIRICAL EVALUATION METHOD

After we designed and implemented FIXBUGS, a prototype fix exploration tool, we performed an experiment to evaluate its usability.[1] In this evaluation, we investigated the standard usability metrics [16] of effectiveness, efficiency, and user satisfaction, using the following research questions:

[1]All materials used for the study, including the FIXBUGS source code, are available at http://go.barik.net/fixbugs.

TABLE I
PARTICIPANTS' EXPERIENCE IN EMPIRICAL EVALUATION

Participant	Dev. (yrs)	Java (yrs)	Eclipse (yrs)
P1	20	17	10
P2	9	6	2
P3	7	6.5	6
P4	6	6	6
P5	7	3	2
P6	3	3	3
P7	3	2	2
P8	8	5	5
P9	5	5	3
P10	1	1	1
P11	3.5	1	1
P12	1.5	1	1
Mean	6.2	4.7	3.5
Std. Dev.	5.1	4.4	2.7

RQ1 How effective and efficient are Quick Fix and FIXBUGS in fixing defects against manual approaches?

RQ2 How satisfied are developers with fixing defects when using Quick Fix, FIXBUGS, and manual approaches?

A. Study Design

Participants. We recruited 12 graduate students (henceforth, participants P1-P12) for our study through a flyer and a mailing list in the Department of Computer Science at our University (Table I). On average, participants had 6.2 years ($sd = 5.1$) of development experience, 4.7 years ($sd = 4.4$) of experience with Java, and 3.5 years ($sd = 2.7$) of experience with Eclipse. All participants were familiar with Eclipse and seven of them had used static analysis tools like FindBugs. Only P11 had no prior experience with the Quick Fix feature. No participant had seen FIXBUGS prior to the study. Four participants were women.

Tools. Participants fixed defects in three ways: manually fixing (MF), using the Eclipse Quick Fix tool (QF), and using FIXBUGS (FB). During the study, we called Eclipse Quick Fix "Quick Fix Version 1" and FIXBUGS "Quick Fix Version 2" so as not to bias participants by letting them know which version we implemented. We compared these approaches on the three defects described in Section IV: Catch Exception (CE), String Concatenation (SC), and Open Stream Exception (OSEP). Although FindBugs did not provide Quick Fixes at the time of the study for these three defects, we implemented them ourselves in Eclipse.

Counterbalancing. We used a within-subjects design for the participants to be exposed to all tools. To control for order effects, we asked participants to use the tools in six different orders, because we had three tools.

Tasks. We gave participants 18 tasks (henceforth, T1-T18) to complete. We gave participants four minutes for each of the first three tasks, intended as warm-up tasks that minimize the problem of practice effects [17], and then three minutes for the remaining tasks. Once time expired, if the defect was not fixed, we marked the task as incomplete.

Each task contained one of the three defects mentioned previously. When participants encountered the Catch Exception defect, we explicitly asked them to throw or catch exceptions based on a pre-specified list of exceptions.

We chose example code from FindBugs, from a paper on static analysis tools [18], and from well-known, real-world, open source projects such as Apache Ant, Apache Lucene [19], ArgoUML [20], HtmlUnit [21], iText [22], JHotDraw [23], and log4j [13]. We ordered the tasks using a metric for readability [24]; the readability of the code generally decreased over the course of each participant's session. To allow participants to focus on fixing the defects, we modified the code to isolate the defects, and removed extraneous code.

B. Pilot Study

We conducted a pilot study with 12 participants. From the pilot study, we obtained parameters such as the selection of training examples, selection of tasks, and time limits for each task so that the study could be completed within an hour. We do not include the results of the pilot study in our analysis.

C. Procedure

Training. Our experiment was conducted by two investigators. The first investigator (third author) conducted the training session, and a second investigator (second author) conducted the actual experiment. This was done to reduce participant response bias by confounding the creator of the tool [25].

We used three examples for the training session. These examples demonstrated the drag-and-drop, suggestion popups, and color annotation features of the tool to the participant. We also explained how to navigate between defects using FindBugs' "Bug Explorer" and how to browse detailed information in FindBugs' "Bug Info" view. The training examples were contextualized through some of the new features of Java 7, such as `try`-with-resources and multi-`catch` exception. A previous pilot study showed that participants did not have familiarity with these features.

This training took about 10-15 minutes. Training was provided to all participants regardless of prior experience.

Experiment. The second investigator conducted the actual experiment. During the study, we allowed participants to use any information from FindBugs and developer community sites such as StackOverflow [26]. We allowed them to use any Eclipse Quick Fixes, such as "add `throws` declaration" or "surround with `try/catch`," because these could have been part of participants' workflows in the wild.

We did not allow participants to go back to previous tasks to prevent them from correcting earlier manual fixes using information learned from later Quick Fixes. Including the training tasks, each session took about 45-60 minutes. Afterwards, we asked each participant to fill out a post-questionnaire regarding user satisfaction. We did not provide compensation for participation.

D. Analysis

1) RQ1: How effective and efficient are Quick Fix and FIXBUGS *in fixing defects against manual approaches?:* We considered the effectiveness (tasks successfully completed) and efficiency (total time taken to complete task) of the tools for each bug type, and compared these results against the manual approach. An initial Shapiro-Wilk normality test identified several of the dimensions as being significantly different from a normal distribution, and so we used a matched-pairs Wilcoxon signed rank test ($\alpha = .05$) for these analyses.

2) RQ2: How satisfied are developers with fixing defects when using Quick Fix, FIXBUGS*, and manual approaches?:* We asked questions about user satisfaction in the post-questionnaire. Each question used a 5-point Likert scale [27] ranging from "Strongly disagree (1)" to "Strongly agree (5)". We used a Wilcoxon signed rank test to compare responses.

VI. Analytic Evaluation Method

In comparison with empirical user evaluation methods, a heuristic evaluation is an analytic form of user evaluation [28]. Whereas empirical methods typically measure the participants' performance directly, such as effectiveness or efficiency, analytic user evaluation methods have expert observers examine the interface or aspects of the interaction to identify usability problems. The output of a heuristic evaluation is a list of usability issues identified in the context of how these issues violate recognized design principles, called *heuristics* [29]. Importantly, the output explains the trade-offs between two or more user interfaces.

Empirical methods and analytic methods are complementary. In empirical user evaluation methods, a tool expert is able to perform and provide feedback on tasks, even if they are not experts on user interface design. In analytic user evaluation methods, an expert evaluator can identify user interface design issues even if they are not experts on the tool itself.

Thus, the use of the analytic, heuristic evaluation allowed us to investigate an additional research question:

RQ3 What interface trade-offs do usability experts identify between Quick Fix and FIXBUGS?

A. Study Design

Participants. We recruited expert evaluators using convenience sampling. We emailed researchers and engineers using personal contact lists. Nielsen recommends the use of three to five expert evaluators [29], and so we contacted potential evaluators from academic computer science, academic psychology and human factors, and industrial UX designers and software engineers. In total, we contacted 11 evaluators and received six acceptances. All evaluators had doctoral degrees in their respective specialization. As a thank you for participating, we offered to acknowledge these researchers in any published materials which used their responses.

Tasks. We used Nielsen's methodology for heuristic evaluation [29]. Each evaluator received an e-mail containing a hyperlink to the evaluation materials. The evaluation materials provided a brief description of each tool, referring simply to Quick Fix as Quick Fix Version 1 and FIXBUGS as Quick Fix Version 2. We asked the evaluators to watch four, two-minute videos of both tools demonstrating actions that we observed

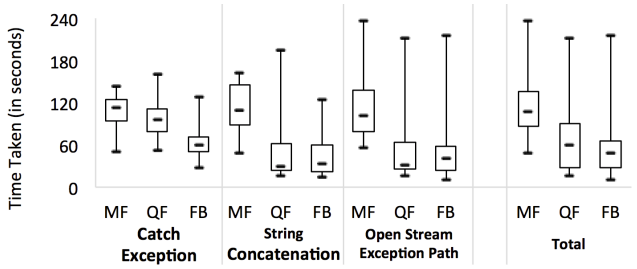

Fig. 8. Time taken by participants to complete given tasks manually, with Quick Fix, and with FIXBUGS. Overall, participants fixing defects manually took longer than participants using Quick Fix and FIXBUGS.

participants taking in the empirical user evaluation. During the task, we asked each evaluator to go through the interface and inspect the various interactions in the videos. In addition, we provided the evaluators with a questionnaire consisting of the ten usability heuristics defined by Nielsen. For each heuristic, we asked evaluators to rate adherence using a 5-point acceptability scale from "Very Poor (1)" to "Very Good (5)" for each tool. For each heuristic, we asked evaluators to provide a written rationale for their ratings in a textbox. We expected the evaluation to take 30-45 minutes to complete. On average, participants completed the task in 41 minutes.

B. Analysis

To understand the overall sentiment of the evaluator, we calculated the difference in ratings for each tool and for each heuristic. For example, if the evaluator marked "Very Good" for Quick Fix and "Acceptable" for FIXBUGS, we tabulated their score for that heuristic as -2.

To identify the trade-offs evaluators found between Quick Fix and FIXBUGS, we framed Nielsen's heuristics as *themes* and conducted a thematic analysis of the responses [30]. To do so, we first removed all heuristics in which two or more evaluators indicated the heuristic to be inapplicable to the current evaluation.[2] Next, we used the ATLAS.ti [31] data analysis software to qualitatively code the data over multiple iterations. In the first cycle, we used descriptive coding to classify phrases in the responses as either applicable to Quick Fix, FIXBUGS, or both. In the second iteration, we used axial coding for each heuristic to organize and identify common patterns between the evaluators and tools.

VII. RESULTS

A. RQ1: How effective and efficient are Quick Fix and FIXBUGS in fixing defects against manual approaches?

In terms of effectiveness, participants manually fixing defects successfully completed 38 out of the 72 tasks (52.8%), whereas participants using Quick Fix completed 69 out of 69 tasks[3] (100.0%) and participants using FIXBUGS completed all 72 tasks (100.0%).

[2]However, the full responses for all heuristics can still be found within our online materials, if the reader is interested.

[3]Three tasks had to be discarded. For details, see Section VIII (Limitations).

In terms of efficiency, Figure 8 shows the time taken (in seconds) to complete the task. Overall, the mean time taken for tasks completed manually was 111.9 seconds ($sd = 38.3$). In comparison, the mean times taken using Quick Fix and FIXBUGS were 65.8 ($sd = 45.9$) and 54.1 ($sd = 35.0$) seconds, respectively.

For the Catch Exception defect, we found that FIXBUGS outperformed Quick Fix ($p < .001$) and manual fixing ($p = .001$). For the String Concatenation defect, the tools outperformed manual fixing (QF: $p < .001$, FB: $p < .001$), though FIXBUGS and Quick Fix were not identified to be significantly different from one another ($p = 1$). Similarly, for the Open Stream Exception Path defect, we were unable to identify any significant differences in task completion time between FIXBUGS and Quick Fix ($p = .957$), but again found that manual fixing took significantly more time than either of the other tools (QF: $p < .001$, FB: $p < .001$).

B. RQ2: How satisfied are developers with fixing defects when using Quick Fix, FIXBUGS, and manual approaches?

Table II shows the results from our post-questionnaire. Columns 2 through 4 visually represent the distribution of ratings for manual fixing, Quick Fix, and FIXBUGS, respectively, where green means strong agreement and red means strong disagreement. The remaining columns contain p-values indicating the statistical significance of differences among manual fixing, Quick Fix, and FIXBUGS.

Participants thought FIXBUGS helped them to quickly and effectively understand what will happen to the code once a fix is applied (Q1, QF-FB: p = .006, FB-MF: p = .005). They felt FIXBUGS minimized the work required to fix defects (Q2, QF-FB: p = .014, FB-MF: p = .002). Participants also said that FIXBUGS makes it easy to differentiate between changes in code, presumably due to the color annotation feature (Q3, QF-FB: p = .003, FB-MF: p = .003). Both FIXBUGS and Quick Fix allowed participants to quickly make changes to their code (Q4, MF-QF: p = .002, FB-MF: p = .003). Participants felt that all three tools allowed them to solve the defect how they wanted (Q5, p = .083). Finally, although participants would use either tool if available, participants indicated a significant preference for FIXBUGS over Quick Fix (Q6, FB-QF: p = .014).

C. RQ3: What interface trade-offs do usability experts identify between Quick Fix and FIXBUGS?

The evaluators have been labeled as E1 through E6, with their affiliation — academic computer science (CS), academic psychology (PSY), and industry (IND) indicated in subscripts. Table III summarizes their relative 5-point Likert responses between Quick Fix and FIXBUGS for each heuristic to give an overall sense of each evaluator's assessments. For example, $E5_{IND}$, an industry software engineer, is more critical of FIXBUGS than the other evaluators.

Visibility of system status. *Heuristic:* The system should always keep users informed about what is going on, through appropriate feedback within reasonable time. *Evaluation.* $E1_{CS}$

TABLE II
USER SATISFACTION, BASED ON POST-QUESTIONNAIRES

Question		MF QF FB	MF-QF (p)	QF-FB (p)	FB-MF (p)
Q1	{MF, QF, FB} helps me to quickly and effectively understand what will happen to my code once the fix is applied.		.374	**.006**	**.005**
Q2	{MF, QF, FB} reduces the amount of work required to fix a bug.		**.002**	**.014**	**.002**
Q3	{MF, QF, FB} makes it easy to differentiate between original, new and modified code.		.792	**.003**	**.003**
Q4	With {MF, QF, FB}, I could quickly make the changes to my code.		**.002**	.655	**.003**
Q5	{QF, FB} fixed my code the way I wanted it fixed (i.e. made the changes I wanted made).		–	.083	–
Q6	If {QF, FB} was available in my favorite development environment, I would use it when I programmed.		–	**.014**	–

p-values indicate the likelihood that responses for one of the tools were significantly different than for the others. Overall, our participants preferred FIXBUGS to both Quick Fix and manual fixing (5-point Likert scale from "Strongly disagree (1)" to "Strongly agree (5)").

TABLE III
RELATIVE LIKERT SCORES FOR ANALYTIC EVALUATION

	CS		PSY		IND	
Heuristic	E1	E2	E3	E4	E5	E6
VISIBILITY	+2		+1	+1	−4	+1
USER CONTROL/FREEDOM	+1	+2	+1	+2	+2	+1
RECOGNITION/RECALL	+2	+2		+4		+1
AESTHETIC/MINIMALISTIC		+1		+2	−3	−1
RECOGNIZE/DIAGNOSE	+1	·	+1	+2	−2	+2

Positive scores indicate preference towards FIXBUGS and negative scores indicate preference towards Quick Fix. A raised dot (·) indicates that the evaluator did not find the criteria applicable.

summarizes the information and feedback from FIXBUGS through three features provided by the tool: "1) highlighting what it thinks are important changes to the code, 2) offering an explicit toggle to go back and forth between the original suggestion and last change by the user, and 3) offering drag-and-drop on the highlighted changes."

Unlike Quick Fix, the feedback in FIXBUGS makes it "obvious when the tool is in use and what state" ($E3_{PSY}$), and it is easy to tell when FIXBUGS is active and when it is not. In addition, through the suggestion popup, the developer can "easily see which option was chosen" ($E3_{PSY}$), and through color annotations, the changes are presented within the code, instead of a preview with different formatting ($E3_{PSY}$, $E6_{IND}$).

Other evaluators found the feedback from the two versions to be about the same in terms of information, but just presented at different times ($E1_{CS}$, $E2_{CS}$, $E5_{IND}$). Quick Fix has an advantage in that it provides fairly verbose explanations and examples of the suggested fix before having to activate the tool at all ($E1_{CS}$, $E5_{IND}$). In contrast, FIXBUGS provides less upfront information, but elaborates more on the suggested approach and rationale to solving the problem only after the tool is invoked ($E1_{CS}$, $E4_{PSY}$).

An interesting observation of FIXBUGS is that the syntax of the language becomes visible through usage of the drag-and-drop capability ($E1_{CS}$). For example, in the Catch Exception defect, the developer does not have to worry about the correct syntax for the various locations where an exception may be placed; the tool will automatically make the proper syntax visible to the developer.

User control and freedom. *Heuristic.* Users often choose system functions by mistake and will need a clearly marked "emergency exit" to leave the unwanted state without having to go through an extended dialogue. Support undo and redo. *Evaluation.* Evaluators generally compared the undo-redo interactions available in Quick Fix and FIXBUGS ($E3_{PSY}$, $E6_{IND}$, $E5_{IND}$), and identified that Quick Fix uses the familar undo-redo operations (e.g., `CTRL-Z`), a "commonly known convention" ($E6_{IND}$, $E3_{PSY}$, $E5_{IND}$) available in most text editors.

In comparison to FIXBUGS, evaluators found that both tools supported undo and redo, but that FIXBUGS supports these operations more explicitly ($E1_{CS}$, $E2_{CS}$, $E4_{PSY}$). In particular, the suggestion popup in FIXBUGS gives the developer "improved control and the ability to easily switch between different suggestions" ($E2_{CS}$).

Two evaluators ($E6_{IND}$, $E3_{PSY}$) identified design failures in FIXBUGS. Although the suggestion popup is more useful than in Quick Fix, "the placement of the suggestion popup is not intuitive" ($E6_{IND}$) because it appears on the opposite side of the screen from the bug marker. As a second failure, $E3_{PSY}$ noted the lack of an "emergency exit," in that it is not obvious in FIXBUGS how the developer should accept or cancel the change. Unintuitively, clicking the `X` button finalizes the change, rather than cancelling it. Adding an explicit "finalize change" button to FIXBUGS would clarify this issue.

Recognition rather than recall. *Heuristic.* Minimize the

user's memory load by making objects, actions, and options visible. The user should not have to remember information from one part of the dialogue to another. Instructions for use of the system should be visible or easily retrievable whenever appropriate. *Evaluation.* Quick Fix requires much more recall and clerical effort than FIXBUGS ($E1_{CS}$, $E2_{CS}$, $E3_{PSY}$, $E4_{PSY}$). Lacking color annotations, Quick Fix simply applies the requested change, and then leaves it to the developer to maintain an internal "mental model state" of what has changed ($E1_{CS}$, $E3_{PSY}$), to see the impact of the changes the system suggestions ($E4_{PSY}$), and to compare changes ($E2_{CS}$). To take advantage of the color annotations successfully in FIXBUGS, however, requires that the developer "learns what the colors mean" ($E6_{IND}$). For example, in the Catch Exception defect, they must recognize that the outline color of the method corresponds to the fill color of the exception itself.

$E3_{PSY}$ suggests that Quick Fix should actually be better for recognition than recall, if not for several design flaws. This is for the simple reason that Quick Fix offers a preview with the original source code so that both can be compared simultaneously; in FIXBUGS, this comparison is serialized ($E3_{PSY}$, $E4_{PSY}$), and you can't look at both at the same time to compare. Although the current implementation of FIXBUGS requires recall, this need not be the case. For example, the use of split-panes could be used to show both versions of the code side-by-side.

Aesthetic and minimalist design. *Heuristic.* Dialogues should not contain information which is irrelevant or rarely needed. Every extra unit of information in a dialogue competes with the relevant units of information and diminishes their relative visibility. *Evaluation.* Evaluators found that Quick Fix and FIXBUGS implement aesthetics in different ways ($E1_{CS}$, $E2_{CS}$). In Quick Fix, the information is tailored to the specific task needing to be carried about, although the developer still has to dedicate time to read and understand it ($E1_{CS}$, $E5_{IND}$). FIXBUGS uses color annotations on source code as a minimalist design to identify what will change with the fix, but leaves it up to the developer to figure out why that suggestion is good or bad ($E1_{CS}$).

$E6_{IND}$ noted that standard UI conventions differ between Quick Fix and FIXBUGS, with Quick Fix being already familiar and not very different from standard interactions within Eclipse.

Help users recognize, diagnose, and recover from errors. *Heuristic.* Error messages should be expressed in plain language (no codes), precisely indicate the problem, and constructively suggest a solution. *Evaluation.* Quick Fix offers verbose information to diagnose errors, but this information is not presented in a user-friendly way ($E1_{CS}$, $E4_{PSY}$). $E1_{CS}$ suggests that in Quick Fix, the verbose information isn't useful to a developer anyway. In such a case, direct interaction, such as the structured drag-and-drop that FIXBUGS provides, is more useful. $E5_{IND}$ differed from the other evaluators, and felt that the constructive solutions in FIXBUGS require more effort to obtain from the tool, whereas Quick Fix provides an immediate preview.

Because of the color annotations, some evaluators found it easier to recognize "what the system thinks are areas of concern" ($E1_{CS}$) with FIXBUGS ($E1_{CS}$, $E4_{PSY}$, $E6_{IND}$). If the developer makes a mistake during the fix, it is also easier to recover the original state using the suggestion popup ($E3_{PSY}$). $E4_{PSY}$ concludes that, with FIXBUGS, "the suggested edits are much easier to evaluate, compare, and execute."

VIII. LIMITATIONS

In this section, we discuss the limitations and threats to validity in our study, starting with task issues. We inadvertently included warm-up tasks during the counterbalancing procedure and could not discard them from our analysis as intended. A bug in our implementation required us to discard two trials: P8 (T1) and P12 (T1). A third trial, P12 (T15) had to be discarded because the participant was asked to use Quick Fix, but the system was accidentally left in manual-only mode. In total, three tasks were discarded.

Because of the small number of participants, the statistical power of the the Wilcoxon signed rank test is low. In addition, the test does not allow for missing data. For the discarded tasks, we performed mean substitution as an approximation. Overall, this affected 3 out of 216 tasks. For incomplete manual tasks, we made the assumption that the task would have been completed *just after* the time limit. For T1-T3, this is four minutes, and three minutes for others. Note this procedure only gives an advantage to manual fixing, and not to our tools.

A second issue relating to sample size is that the task efficiency may be confounded with participant experience. To understand the extent of this threat, we constructed a decision tree using task data and pre-questionnaire responses as inputs to the model [32]. The inputs include the tool (MF, QF, FB), the task (T1-T18), the bug type (CE, SC, OSEP), and the self-reported experience of the developer for general software experience, Java experience, and Eclipse experience. The response variable is the time to complete the task. Because we are interested in the influence of experience for this *particular* data set, we intentionally overtrained the model to maximize the coefficient of determination, which occurred at $R^2 = 0.656$. Under this model, the input contributions are: tool (36%), bug type (18%), task (16%), Java experience (14%), developer experience (12%), and Eclipse experience (2%). Therefore, some form of experiences explains 28% of the observed task efficiency.

Finally, although we named our tools Quick Fix Version 1 and Quick Fix Version 2, the use of a Version 2 label inherently suggests that it is an improvement over Version 1. And because participants in our study had prior experience with Eclipse, they likely guessed that Quick Fix Version 2 was our tool. Thus, a potential threat is that these participants experienced social desirability bias [33], and gave more favorable post-questionnaire responses to our tool than they would have otherwise. To mitigate this threat, we specifically recruited psychologists and human factors researchers without prior experience in programming or Eclipse in our analytic heuristic evaluation. In our analytic evaluation, we also explicitly asked respondents to not be concerned with "being too critical of the

tool" and indicated that "strong criticism [was] expected and encouraged."

IX. Related Work

Our work is related to research on exploring design spaces, improving Quick Fix automation, code suggestion and completion, and visual aides.

Exploring design spaces. Studies have investigated how analysis tools can help users explore design spaces. For example, Oney and Brandt [34] explored the effectiveness of using interactive helpers to aide developers in understanding and using outside coding examples after they have copied and pasted the code into their own. Ferriera and colleagues [35] created a set of guidelines and implemented a tool for increasing user confidence when identifying uncertainty in charts and conducting common data exploration. Xie and colleagues [36] investigated the use of automatic code generation and interactive guidance to support developers writing secure software. When debugging, Ko and Myers [37] found that developers significantly benefit when the task aligns with the developers' intentions. Another type of design exploration, found by Piorkowski and colleagues [38], occurs when attempting to balance learning about a defect with fixing the defect. They discovered that learning and fixing can require different information and tactics.

Improving Quick Fix automation. Other research has borrowed from or built on features found in the Eclipse IDE, such as Quick Fixes, to improve those features' usability. Vakilian and colleagues [39] proposed a tool called Keshmesh for detecting and fixing concurrency defects in Java programs. Their approach provides code suggestions using Eclipse's Quick Fix interface. Muşlu and colleagues [40] recently proposed Quick Fix Scout to analyze suggested code changes before applying them and presents the number of remaining compilation errors in front of the suggested fixes. Given a fix, it informs the developer of how many errors the fix will introduce or remove.

Our approach is complementary to improvements in static analysis algorithms and can take advantage of these technical improvements. For example, our approach can aid Quick Fix Scout by directly applying code suggestions to existing code in the editor (e.g., through color annotations), so that developers can easily navigate suggestions and observe the impact of applying a suggestion.

Code suggestion and completion. Mooty and colleagues [41] proposed Calcite, which suggests code examples for constructing a given class or interface. The tool is integrated into Eclipse and uses the default completion proposal context menu to allow developers to choose their suggestions. However, Calcite provides snippets from the web, which could lead to compilation errors when applied and does not allow for fix previews. Similar to Calcite, Omar and colleagues [42] proposed a code completion tool for Eclipse, called Graphite. They described new user interfaces and the design constraints for their technique. FixBugs is designed to help developers fix defects, whereas their approach helps developers in creating objects or regular expressions. Our approach also actively uses JDT to analyze the underlying context in the editor and to provide context-sensitive code suggestions to fix defects; Graphite uses HTML and Javascript to support extensibility.

Visual aides. Henley and Fleming [43] proposed an approach called Patchworks to reduce the time and errors introduced when navigating large codebases using existing tools, such as with the "Outline View" in Eclipse. Dekel and Herbsleb [44] proposed an approach, called eMoose, designed to improve API documentation usability by decorating method invocations with annotations to increase awareness of important information. Similarly, FixBugs decorates code with annotations to increase awareness of what has been changed. Moreover, FixBugs provides a color mapping between method invocations and exceptions that the invocations may throw, and provides two types of border annotations (i.e., solid or dashed border) to differentiate between draggable and non-draggable code elements. Cottrell and colleagues [45] proposed an approach, called Jigsaw, that helps with code reuse tasks by comparing structural correspondences between originally copied seed code and paste seed code. Jigsaw uses colors based on similarity scores to draw developers' awareness, while FixBugs uses colors to differentiate which part of code is added, changed, or related from original code.

X. Discussion

A. Implications

The triangulation of our empirical evaluation method and analytic evaluation method suggest several design techniques that benefit both Quick Fix and FixBugs types of tool experiences.

1) Enable exploration of alternative design spaces: Although Quick Fixes are fast to invoke and apply, additional developer effort is required when they wish to evaluate multiple fixes, for example, by repeatedly undoing and redoing fixes. In FixBugs, a suggestion popup makes it easier to toggle between multiple fixes. Even when tools supporting exploration are available in IDEs, the developer must wrangle with the IDE to bring these tools to their task context. For example, the color annotations in FixBugs could theoretically be achieved using existing tools, such as IntelliJ's "Compare Files" [46]. However, to compare a design space, the developer would first need to take a snapshot of the appropriate source files, apply a fix, and finally, activate the "Compare Files" tool to open a separate differences-viewer window. If the developer performed any of these operations incorrectly, or wanted to explore an alternative design space, they would need to redo the sequence of steps. In contrast, the novelty of FixBugs is that it facilitates exploration of a change by automatically providing the developer with relevant tools during the fix process. Nevertheless, in cases where the desired fix is unambiguous, developers don't seem to benefit from the verbosity provided by FixBugs. In this scenario, a simple tooltip indicating the change is likely sufficient for the developer.

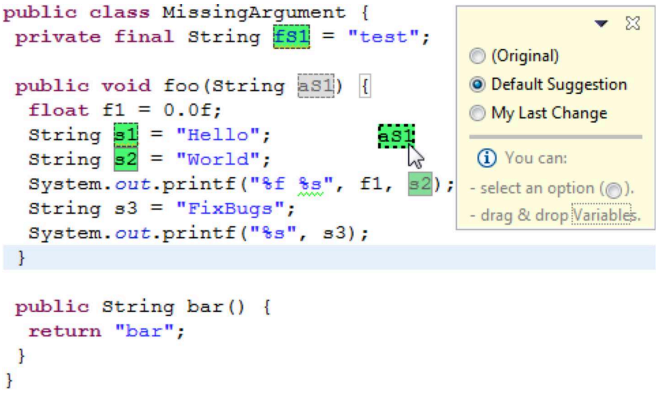

Fig. 9. Suggested fix for the Missing Argument defect. The variables in green boxes are candidates for `%s`. The dotted borders of the variables denote that the variables can be drag-and-dropped. Currently, `aS1` is being dragged and about to be dropped to replace the already suggested variable `s2`.

2) Provide direct feedback: The heuristic evaluation revealed that, strictly speaking, Quick Fix is better for recognition because it simultaneously displays both the original code as well as a preview of the intended fix. However, this advantage is largely negated against the color annotations of FIXBUGS because the preview is neither formatted the same as the source code, nor directly displayed in terms of the source the developer is already editing. Consider using techniques such as a split-pane view, to enable developers to not only understand what changes after the fix, but also to identify *what* the fix changes with respect to the original source code.

3) Offer structured manipulation: There are times, as with the Catch Exception defect, when the design space is large yet the types of manipulations that the developer needs to make are still structured. In such cases, one can use existing static analysis techniques to support the developer and provide smarter editing modes, such as drag-and-drop. Even with experts, the drag-and-drop approach is faster and less error-prone than fully manual approaches.

B. Extending FIXBUGS

Using the design techniques we have just outlined, we illustrate how these techniques can be extended to other static analysis fixes.

Consider the code example containing a Missing Argument defect, shown in Figure 9. The problem is that the argument for `%s` is missing, and when the method is executed, a runtime exception will occur. The solution to the defect is to provide an expression of type `String` for `%s`.

For this defect, FIXBUGS first notices the incorrect format string and draws a green, wavy underline on `%s` indicating that an actual argument has been missed. Then, it finds all candidate variables of that type (`String`) in the scope of the `MissingArgument` class to be used as suggestions. In this example, candidate suggestions will be `fS1`, `aS1`, `s1`, and `s2`.

Because the developer is structurally manipulating the code directly through drag-and-drop, it is less likely that they will need a split-pane view to compare their change against the original. To enable design space exploration, users may prefer other expressions to `String` variables for the missing argument, so FIXBUGS allows users to edit code to provide an expression manually. For example, a developer might want to make a function call like `System.out.printf(‘‘%f %s’’, f1, bar());`.

XI. CONCLUSION

In this paper, we examined how moving beyond speed as a primary criteria for fixes can benefit developers. Through an empirical evaluation, we found that developers using FIXBUGS are able to understand, differentiate, and make changes to their code, while still maintaining comparable performance to speed-based Quick Fix tools. Put another way, slow fixes need not be significantly slower than quick fixes. Despite the additional actions that slow fixes sometimes require, slow fix tools such as FIXBUGS make up for this deficiency through the exploratory affordances that enable developers to better understand and evaluate the impact of their fixes.

Our analytic evaluation identified trade-offs between the design of Quick Fix and FIXBUGS, and suggested design techniques that can be applied to improve both. Our findings suggest that the design of Quick Fix and FIXBUGS are in many ways complementary, with Quick Fix remaining the tool of preference for one-shot fixes, and where minimal or no design exploration is necessary to apply the fix. FIXBUGS excels in situations where fixes can be provided, but the permutations of possible design choices within that space have more variation.

Thinking in terms of quick and slow paradigms can improve developer tool experiences in other domains. For example, straightforward refactoring changes such as renaming might be better suited to fast fixes. When performing a more complex refactoring, however, such as extracting to a superclass, developers may appreciate the additional affordances provided by slow fixes. In short, our tools should sometimes be fast and sometimes be slow — because developers think both fast and slow [47].

ACKNOWLEDGMENTS

We thank the expert evaluators, Jing Feng, Christopher G. Healey, Anne McLaughlin, Kivanç Muşlu, Kenya Oduor, and Robert St. Amant, for their time and detailed feedback. This material is based upon work supported by the National Science Foundation under Grant No. 1217700 and through a Google Faculty Research Award.

REFERENCES

[1] D. Hovemeyer and W. Pugh, “Finding bugs is easy,” *SIGPLAN Not.*, vol. 39, no. 12, pp. 92–106, 2004.
[2] “LLVM/Clang,” http://clang-analyzer.llvm.org/.
[3] “CheckStyle,” http://checkstyle.sourceforge.net/.
[4] V. B. Livshits and M. S. Lam, “Finding security vulnerabilities in Java applications with static analysis,” in *SSYM*, 2005, pp. 271–286.
[5] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, “Why don’t software developers use static analysis tools to find bugs?” in *ICSE*, 2013, pp. 132–141.
[6] “Eclipse,” http://www.eclipse.org/.
[7] “Resharper,” http://www.jetbrains.com/resharper/.

[8] M. Palyart, G. C. Murphy, E. Murphy-Hill, and X. Blanc, "Speculative reprogramming," in *FSE*, 2014, pp. 837–840.

[9] J. Y. Gil, I. Maman, J. Y. Gil, and I. Maman, "Micro patterns in Java code," in *OOPSLA*, 2005, pp. 97–116.

[10] B. Pietrykowski, "You are what you eat: The social economy of the slow food movement," *Review of Social Economy*, vol. 62, no. 3, pp. 307–321, 2004.

[11] "Ant," http://ant.apache.org/.

[12] "Multi-catch clause," http://docs.oracle.com/javase/specs/jls/se7/html/jls-14.html#jls-14.20.

[13] "log4j," http://logging.apache.org/log4j/.

[14] J. Bloch, *Effective Java*. Prentice Hall, 2008.

[15] "The try-with-resources statement," http://docs.oracle.com/javase/specs/jls/se7/html/jls-14.html#jls-14.20.3.

[16] A. Dix, J. Finlay, G. D. Abowd, and R. Beale, *Human-Computer Interaction*, 3rd ed. Prentice-Hall, 2004.

[17] B. A. Sheil, "The psychological study of programming," *ACM Computing Surveys*, vol. 13, no. 1, pp. 101–120, 1981.

[18] N. Rutar, C. B. Almazan, and J. S. Foster, "A comparison of bug finding tools for Java," in *ISSRE*, 2004, pp. 245–256.

[19] "Apache Lucene," http://lucene.apache.org/core/.

[20] "ArgoUML," http://argouml.tigris.org/.

[21] "HtmlUnit," http://htmlunit.sourceforge.net/.

[22] "iText," http://itextpdf.com/.

[23] "JHotDraw," http://www.jhotdraw.org/.

[24] R. P. L. Buse and W. R. Weimer, "A metric for software readability," in *ISSTA*, 2008, pp. 121–130.

[25] N. Dell, V. Vaidyanathan, I. Medhi, E. Cutrell, and W. Thies, ""Yours is better!": Participant response bias in HCI," in *CHI*, 2012, pp. 1321–1330.

[26] "StackOverflow," http://www.stackoverflow.com/.

[27] R. Likert, "A technique for the measurement of attitudes," *Archives of Psychology*, vol. 22, no. 140, pp. 1–55, 1932.

[28] W. D. Gray and M. C. Salzman, "Damaged merchandise? A review of experiments that compare usability evaluation methods," *Human-Computer Interaction*, vol. 13, no. 3, pp. 203–261, 2009.

[29] J. Nielsen, "Heuristic evaluation," *Usability inspection methods*, vol. 17, no. 1, pp. 25–62, 1994.

[30] J. Saldaña, *The Coding Manual for Qualitative Researchers*. SAGE Publications, 2009.

[31] "Atlas.TI," http://atlasti.com/.

[32] M. Gaudard, P. Ramsey, and M. Stephens, "Interactive data mining and design of experiments: The JMP partition and custom design platforms," *North Haven Group*, 2006.

[33] R. J. Fisher, "Social desirability bias and the validity of indirect questioning," *Journal of Consumer Research*, pp. 303–315, 1993.

[34] S. Oney and J. Brandt, "Codelets: Linking interactive documentation and example code in the editor," in *CHI*, 2012, pp. 2697–2706.

[35] N. Ferreira, D. Fisher, and A. C. Konig, "Sample-oriented task-driven visualizations: Allowing users to make better, more confident decisions," in *CHI*, 2014, pp. 571–580.

[36] J. Xie, H. Lipford, and B.-T. Chu, "Evaluating interactive support for secure programming," in *CHI*, 2012, pp. 2707–2716.

[37] A. J. Ko and B. A. Myers, "Finding causes of program output with the Java Whyline," in *CHI*, 2009, pp. 1569–1578.

[38] D. Piorkowski, S. D. Fleming, C. Scaffidi, M. Burnett, I. Kwan, A. Z. Henley, J. Macbeth, C. Hill, and A. Horvath, "To fix or to learn? How production bias affects developers' information foraging during debugging," in *ICSME*, 2015, pp. 11–20.

[39] M. Vakilian, S. Negara, S. Tasharofi, and R. E. Johnson, "Keshmesh: A tool for detecting and fixing Java concurrency bug patterns," in *SPLASH (Companion)*, 2011, pp. 39–40.

[40] K. Muşlu, Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Speculative analysis of integrated development environment recommendations," in *OOPSLA*, 2012, pp. 669–682.

[41] M. Mooty, A. Faulring, J. Stylos, and B. A. Myers, "Calcite: Completing code completion for constructors using crowds," in *VL/HCC*, 2010, pp. 15–22.

[42] C. Omar, Y. Yoon, T. D. LaToza, and B. A. Myers, "Active code completion," in *ICSE*, 2012, pp. 859–869.

[43] A. Z. Henley and S. D. Fleming, "The Patchworks code editor: Toward faster navigation with less code arranging and fewer navigation mistakes," in *CHI*, 2014, pp. 2511–2520.

[44] U. Dekel and J. D. Herbsleb, "Improving API documentation usability with knowledge pushing," in *ICSE*, 2009, pp. 320–330.

[45] R. Cottrell, R. J. Walker, and J. Denzinger, "Semi-automating small-scale source code reuse via structural correspondence," in *FSE*, 2008, pp. 214–225.

[46] "IntelliJ IDEA," http://www.jetbrains.com/idea/.

[47] D. Kahneman, *Thinking, Fast and Slow*. Macmillan, 2011.