A Human Study of Fault Localization Accuracy

Zachary P. Fry
University of Virginia
Email: zpf5a@virginia.edu

Westley Weimer
University of Virginia
Email: weimer@virginia.edu

***Abstract*—Localizing and repairing defects are critical software engineering activities. Not all programs and not all bugs are equally easy to debug, however. We present formal models, backed by a human study involving 65 participants (from both academia and industry) and 1830 total judgments, relating various software- and defect-related features to human accuracy at locating errors. Our study involves example code from Java textbooks, helping us to control for both readability and complexity. We find that certain types of defects are much harder for humans to locate accurately. For example, humans are over five times more accurate at locating "extra statements" than "missing statements" based on experimental observation. We also find that, independent of the type of defect involved, certain code contexts are harder to debug than others. For example, humans are over three times more accurate at finding defects in code that provides an array abstraction than in code that provides a tree abstraction. We identify and analyze code features that are predictive of human fault localization accuracy. Finally, we present a formal model of debugging accuracy based on those source code features that have a statistically significant correlation with human performance.**

I. Introduction

Maintenance typically dominates the life cycle costs of modern software projects by as much as 70% [4]. A key task in software maintenance is the process of reading and understanding code for the purposes of debugging or evolving it [22], [23]. Software is read and corrected more than it is freshly written, but some factors that influence the difficulty of code inspection tasks remain poorly understood.

Broadly speaking, fault localization is the task of determining if a program or code fragment contains a defect,[1] and if so, locating exactly where that defect resides. Typically, after a defect has been reported, triaged, and assigned to a developer, that developer will attempt to localize the defect and then repair it. A number of automated approaches have been proposed for fault localization — for generic defects [14], bugs found by tools [2], and harmful program evolutions [29] — but much fault localization in practice remains manual.

Bugs are plentiful, and the number of outstanding software defects typically exceeds the resources available to address them [1]. Bug repair is time-consuming, with half of all fixed defects in Mozilla requiring over 29 days from start to finish [12]. Since fault localization is a key component of debugging, we investigate which features of defects and programs correlate with human performance at this task.

[1]We use the terms *bug*, *defect*, and *fault* interchangeably in this paper to describe semantic errors in programs.

We hypothesize that four broad classes of features can help to explain human performance at fault localization tasks. First, the form of the bug itself is directly relevant: defects related to uninitialized variables may be hard to locate, for example. However, context features related to the software surrounding the bug, independent of the type of the bug, are also important. Our second class of features relates to syntax and surface presentation. For example, comments may make defects easier to locate. A third class of features relates to control flow. For example, three sequential loops may be easier to debug than three nested loops. Finally, our fourth class of features relates to program abstraction and other properties that may be difficult to discover automatically. For example, the use of trees instead of arrays may hinder human debugging efforts. This inclusive set of features is in contrast to previous work, which tends to focus on one sort of context at the expense of others (e.g., complexity metrics often use only control flow information, readability metrics use surface features, etc.).

To test this hypothesis, we conducted a human study of 65 participants. The base programs used in the study were taken from five popular Java textbooks to control for general code quality and readability. Faults were manually injected into the Java code following a frequency distribution obtained from actual bug reports and fixes in the Mozilla project.

A firmer understanding of the manual fault localization process could help inform software design for maintainability, guide code reviews to difficult-to-debug spots, focus code understanding tools on areas where humans are inaccurate, and potentially influence training and pedagogy. The four main contributions of this paper are:

- A human study of 65 participants on a concrete fault localization task. To the best of our knowledge, this is the first published human study of experienced programmer performance at source-level fault localization.
- A quantitative analysis relating the type of defect to human fault localization accuracy.
- A quantitative and qualitative analysis relating surface, control flow, and abstraction features to human fault localization accuracy.
- A formal model of fault localization difficulty that correlates with human accuracy in a statistically significant manner. Our model correlates at least four times more strongly than do common baselines (e.g., readability, Cyclomatic complexity, textbook positioning).

26th IEEE International Conference on Software Maintenance in Timişoara, Romania
978-1-4244-8628-1/10/$26.00 ©2010 IEEE

```
1  /** Move a single disk from src to dest. */
2  public static void hanoi1(int src, int dest){
3    System.out.println(src + " => " + dest);
4  }
5  /** Move two disks from src to dest,
6      making use of a spare peg. */
7  public static void hanoi2(int src,
8                            int dest, int spare) {
9    hanoi1(src, dest);
10   System.out.println(src + " => " + dest);
11   hanoi1(spare, dest);
12 }
13 /** Move three disks from src to dest,
14     making use of a spare peg. */
15 public static void hanoi3(int src,
16                            int dest, int spare) {
17   hanoi2(src, spare, dest);
18   System.out.println(src + " => " + dest);
19   hanoi2(spare, dest, src);
20 }
```

Fig. 1. Towers of Hanoi explanation code (Drake [9]) with seeded fault. The bug is that **dest** on line 9 should be **spare**. Only 33% of participants were able to locate the line containing the bug.

II. Motivating Example

The difficulty of localizing a fault may depend on many factors: the type of the fault, surface features, control-flow features, and deeper abstract features of the surrounding code. In this section we motivate this reasoning with two conceptually-similar algorithms implemented in different ways. Figure 1 and Figure 2 present two syntactically-different but semantically-similar explanations of the "Towers of Hanoi" problem taken from Java textbooks. Each listing contains a single bug.

The two excerpts complete the same task. For example, `hanoi1` in Figure 1 loosely corresponds to `moveOneDisk` in Figure 2 (i.e., printing out a move), while `hanoi2` and `hanoi3` in Figure 1 loosely correspond to `moveTower` in Figure 2 (i.e., moving some number of disks from one place to another).

In this example, the faults injected into the programs were slightly different. In Figure 1, line 9 incorrectly references `dest` instead of `spare` which would be semantically correct. In Figure 2, line 20 should read `moveOneDisk(start,end)`: the entire method call, including all of its arguments, is incorrect.

The two listings also differ in a number of contextual ways. At the surface level, Figure 1 is shorter, contains fewer comments, and has shorter identifier names. At the control-flow level, Figure 2 is more complex, containing both a conditional branch as well as a recursive call but also is better documented and contains descriptive identifiers. In terms of design and abstraction, Figure 2 presents a general solution for $n \geq 1$ disks while Figure 1 only handles $1 \leq n \leq 3$ by explicit enumeration.

This paper presents a study of human fault localization accuracy. The details are elaborated in subsequent sections, but at a high level, only 33% of participants were able to indicate that the bug in Figure 1 is on line 10, while 53% were able to indicate that the bug in Figure 2 is on line 22. While the difference is significant, its causes and correlates may not be well understood. The topical similarity of the examples

```
1  /*****************************************
2    Performs the initial call to moveTower
3    to solve the puzzle.  Moves the disks
4    from tower 1 to tower 3 using tower 2.
5  *****************************************/
6  public void solve () {
7    moveTower (totalDisks, 1, 3, 2);
8  }
9
10 /*****************************************
11   Moves the specified number of disks
12   from one tower to another by moving a
13   subtower of n-1 disks out of the way,
14   moving one disk, then moving the
15   subtower back. Base case of 1 disk.
16 *****************************************/
17 private void moveTower (int numDisks,
18                         int start, int end, int temp) {
19 if (numDisks == 1)
20     moveTower(numDisks-1, temp, end, start);
21   else {
22     moveTower (numDisks-1, start, temp, end);
23     moveOneDisk (start, end);
24     moveTower (numDisks-1, temp, end, start);
25   }
26 }
27 /*****************************************
28   Prints instructions to move one disk
29   from the specified start tower to the
30   specified end tower.
31 *****************************************/
32 private void moveOneDisk (int start, int end) {
33   System.out.println ("Move one disk from "
34       + start + " to " + end);
35 }
```

Fig. 2. Towers of Hanoi explanation code (Lewis and Chase [19]) with seeded fault. The bug is that line 20 should read `moveOneDisk(start,end)`. Over 53% of participants were able to locate the line containing the bug.

suggests that the type of the defect and features related to the code context contributed to the disparity in debugging accuracy. We thus desire to study the interplay of such features more formally.

III. A Model of Fault Localization Accuracy

In this paper we define *fault localization* to be the task of indicating the source code line that contains the defect (or correctly indicating "none" if no defect exists) given a source code file and a range of lines in which at most one defect is present. A single fault localization task might be to determine where the fault exists in lines 400–500 of `Queue.java` (or to determine that no fault exists in lines 400–500). The line range allows the task to incorporate external debugging information. For example, coverage information or test cases may indicate that the bug could only be within the `insertElement()` method on lines 400–500 of `Queue.java`. If Y separate human fault localization judgments are made and X of them give the correct answer (i.e., the exact line if a bug is present, or "none" otherwise), then the corresponding *fault localization accuracy* is X/Y. Note that in this definition of accuracy, the fault localization must be exact (i.e., cannot be off by even one line). One minor exception is related to faults of omission: if the bug is that an entire line has been removed between lines L and $L+1$, then either L or $L+1$ is deemed a correct answer.

Not all faults are equally easy to localize. Since software maintenance, fault localization and debugging remain critical tasks, an understanding of the factors that relate to the success of manual fault localization would be of general use in software engineering (e.g., to help design for maintainability, guide code reviews, focus understanding tools, and direct training). We thus desire an accurate, formal model that relates basic, understandable features of defects and programs to human programmer accuracy at locating said defects.

We believe that understanding code is a critical aspect of localizing faults in it: ultimately a defect is an instance where a program's implementation does not adhere to its specification. Localizing such a disagreement requires understanding both the implementation and the specification. The specification is not always made explicit by the code: in practice, desired behavior is often understood in terms of implicit universal specifications (i.e., do not crash or loop forever), common background knowledge (i.e., what does it mean to be a "balanced tree"?), and comments and identifier names. The implementation is the code itself, and in a language such as Java its behavior is understood in terms of imperative state manipulations and object-oriented method invocations and class structures. We hypothesize that four classes of features are relevant to programmer understanding and thus to fault localization: the type of defect, surface and syntactic code features, control-flow code features, and features related to abstraction and deeper reasoning.

We propose to model fault localization accuracy using a combination of these features. In particular, we aim to model the correlation between defect type and accuracy, as well as to separately model the correlation between all other types of code-related features and accuracy. The former model helps to answer the research question:

> (Q1) Ignoring code context, which types of defects are harder for humans to localize?

while the later model helps to answer the research question:

> (Q2) Ignoring the type of fault, which pieces of code are harder for humans to localize faults in and why?

A. Model Features

We now describe the four feature categories in detail, enumerating the particular features they contain.

Defect category. While the exact type of defect typically cannot be known in advance and is thus not suitable for a predictive model, we are still interested in the correlation between defect types and programmer accuracy. For example, while Knight and Ammann [16] have shown that "it is possible to seed errors using only simple syntactic techniques that are arbitrarily difficult to locate" using test suites, it is generally unknown which categories of defects are more difficult for humans to localize. For the purposes of this paper we have adapted the fault taxonomy of the above authors, which loosely classifies certain defects in terms of surface features. We have expanded this classification to include all faults we observed in a real system in keeping with the original granularity. Defect categories include notions such as "missing conditional clause", "extraneous statement", "wrong parameter passed", and over a dozen others. We claim no new results in fault classification, instead empirically correlating existing understanding of defect categories with localization accuracy.

Surface and Syntax. We hypothesize that surface features, such as identifier lengths, and syntactic features, such as counts of variable declarations and methods calls, influence human ability to understand a program and thus to locate defects in it. Intuitively, this category of features includes elements most naturally determined with a lexer, parser or tool such as `grep`. Features measured include raw counts, averages, minima and maxima of elements such as identifiers, comments, whitespace, loops, if statements, declarations, mathematical operators, assignments, and method calls. For example, frequent comments are predicted to make fault localization easier.

Control Flow. We hypothesize that control-flow features, such as looping or branching structures, influence human ability to understand whether a program is correct along all of its paths. Previous work has shown that programmers are more likely to make mistakes in the presence of multiple control-flow paths, especially near "hidden" control-flow paths related to exceptions [27]. Intuitively, this includes elements most naturally determined with a control flow graph or program dependence graph. Features measured include the number of nodes and cycles in the control flow graph, the in- and out-degree of nodes, the maximal nesting depth, the presence of recursion, the number of leaf nodes, and the number of calls to methods in the same class. For example, the average number of CFG out edges relates code to its inherent degree of conditional divergence, which may make it harder to debug.

Abstraction. We hypothesize that design choices related to algorithm implementation and abstraction influence human ability to manage implementation complexity and understand what a program actually does, and thus whether or not it has a bug. Features in this category are typically not easily measured automatically; they often rely on high-level human classifications. For example, consider code to maintain a sorted sequence of records using an array data structure, and similar code that uses a B+ tree. While the tree data structure may be more efficient overall, its associated code may involve more complex invariants and its implicit correctness argument is thus more complicated. Features in this category include the presence of specific data structures (e.g., hashmaps, linked lists, arrays, queues, etc.) and specific algorithms (e.g., list sorting, list traversal, tree merging, etc.).

B. Feature Weights and Formal Model

As the motivating example in Section II suggests, it is not clear *a priori* whether a feature is a strong or weak predictor of accuracy, and whether a feature is a positive or negative predictor of accuracy. We thus empirically learn relative weights for these features from human fault localization judgments. In particular, we use linear regression to learn a coefficient

corresponding to each feature. Our final formal model is thus:

$$\mathsf{predicted\_accuracy}(f) = c_0 + \sum_i c_i f_i$$

Where $f = f_1 \ldots f_n$ are the feature values associated with a fault localization instance and $c_0 \ldots c_n$ are the constants learned by linear regression. More complicated machine learning algorithms are available, but our primary goal is explanatory power rather than automated predictive performance, especially since many of our features cannot be obtained automatically. Basic regression has the advantages of being well-understood and of directly admitting analyses such as ANOVAs, allowing for a clear description of feature power. The next section describes training and evaluating this model.

IV. Experimental Methodology

We propose to relate defect categories to human fault localization accuracy (Research Question Q1) as well as to create a formal model that explains human fault localization accuracy in terms of features of the surrounding code (Research Question Q2). Both goals require knowledge of how humans perform on fault localization tasks. We thus obtained local IRB approval and conducted a human study (i.e., an experiment involving human subjects).

The human study was constructed so as to recreate a software engineering task in a manner close to reality, while simultaneously scientifically controlling as many variables as possible. There are four key issues of the experimental design: the experimental protocol (i.e., the information shown to participants and the responses gathered), the selection of the code in which the faults were placed, the selection of the faults themselves, and the selection of the human participants. We detail each of those design decisions in turn.

A. Human Study Protocol

Each participant was presented with a random selection of thirty separate fault localization tasks using a web interface. In each task the human was shown a Java file with a range of 20 consecutive lines visually marked off; participants were told that the file either contained no error or contained exactly one error within that 20 line window. This setup simulates a scenario in which triage and other information have limited possible defect locations to a particular region of the code (e.g., a particular method) while also controlling the search space size throughout the study. Participants were asked:

- Is there a bug in the outlined search space? (Yes/No)
- If yes, which line contains the bug? (Number)
- How difficult do you feel this code is to understand and debug? (1–5 Likert scale)

Notably, the participants were not permitted to run the code or use any external debugging or search tools as such aids could account for further experimental bias based on the tools chosen and respective user familiarity.

B. Code Selection

We chose to take subject code from textbooks indicative of the first few years of computer science undergraduate education. Using textbook code rather than commercial or open source code has a number of experimental benefits. First, it helps to control for readability: textbook code is presumably crafted for pedagogical purposes, and all code in a single textbook presumably adheres to the same style. Second, it helps to control for complexity: two different implementations of the same algorithm, such as quicksort, necessarily have the same inherent complexity and differ only in their presentation. Third, it helps to control for subject understanding of implicit specifications: practitioners are likely to be familiar with the correct operation of heaps, trees, arrays, sorting, and other textbook topics. Fourth, it helps to control for the presence of unknown or unintended faults: we require at most one fault per localization task, and textbook code is less likely to contain a latent error before we inject one. These textbook properties are explicit assumptions of this paper. Despite the potential drawbacks of using textbook code rather than explicitly commercial code (discussed in Section V-D), we chose it because the ability to control external quality factors is of paramount importance in such a human study and textbook code readily facilitates such standardization.

An initial set of 76 textbooks was selected by searching publisher websites for Java textbooks in categories matching "introduction to computer science", "introduction to programming", "intermediate or advanced programming", "data structures", "programming languages", or "algorithms". Of those, nineteen were found that claimed to include freely-available on-line instructor materials for code or labs. Of those nineteen, five contained code that was pure Java. For example, the popular *Algorithm Design* by Kleinberg and Tardos could not be included because it presents algorithms using a high level Java-like pseudocode that includes symbols such as $\leftarrow$ and $\cap$ as punctuation. Table I details the final five subject textbooks.

TABLE I
Subject textbooks used in our human study.

Author	Textbook Title and Edition
Carrano, Frank M.	Data Structures and Abstractions with Java, 2e [7]
Drake, Peter	Data Structures and Algorithms in Java, 1e [9]
Lafore, Robert	Data Structures and Algorithms in Java, 2e [18]
Lewis, John; Chase, Joseph	Java Software Structures: Designing and Using Data Structures, 2e [19]
Savitch, Walter	Absolute Java, 3e [25]

We then selected 45 Java classes from the source code included with those five textbooks. Thirty classes were randomly selected so as to be evenly distributed with respect to their normalized point of introduction (i.e., one-fourth were chosen from the first 25% of their respective textbooks, one-fourth from the next 25%, etc.). Fifteen classes were selected that covered similar concepts from multiple textbooks (i.e., two different textbook versions of quicksort). In all cases we attempted to control for length by beginning the search with

files up to 70 lines long and taking the closest match, in keeping with the distribution, from there. In total we used 45 Java files totaling 3519 lines of code; most of the classes fall between 50 and 80 lines of code (average of 78).

C. Fault Selection and Seeding

Given Java classes from textbook code, we now focus on using them to construct fault localization task instances. Recall that each task consists of a Java class containing at most one bug, and a 20-line window that contains the bug if it is present. We proceed by fault seeding: manual injection allows us to control fault presence and location. The two key questions are thus which faults to select and where to inject them.

1) Fault Selection: Previous work has inspected both the classification of faults for the seeding purposes and the generalizability of seeded faults to actual defects (e.g., [11], [16], [28]). Due to the small scale of the code used for this study, we adapt this previous work to ensure not only the feasibility of a fine-grained analysis but also that we are following a realistic distribution of fault types and frequencies.

Faults can be logically classified using characteristics including size, type, severity, and location. We vary the types of the faults included in our study specifically in an attempt to identify which errors are harder to find. For design simplicity and to allow for reasonable ease of task completion, we restrict attention to bugs that can be traced to a single line.

Existing fault taxonomies typically provide defect categories but not defect frequencies. To obtain a frequency basis, we manually examined one hundred consecutive bug fixes in the Mozilla project bug report repository and version control system, recording the basic classification of each one and its best fit in an extended Knight and Amman taxonomy [16]. Examples of such classifications include "incorrect conditional operator", "incorrect variable used", and "missing assignment statement". This produced relative frequency counts for all eighteen different types of faults under consideration.

2) Fault Seeding: The fault seeding process was performed manually based on random numbers. Given a piece of code, we first randomly selected a type of fault based on the obtained frequency distribution. In some cases the code did not admit the given type of fault (e.g., a conditional "or to and" bug cannot be seeded in code with no "or" operators). In such cases another fault type was chosen at random from the distribution. If the fault was applicable to the code, all possible places in which the fault might be seeded were enumerated. One place was then chosen at random and the fault was seeded there. To further control for the length of code participants would have to examine, we clearly marked a twenty line subspace of each code excerpt containing the bug. The subspaces were chosen by randomly choosing a value between 1 and 20 as the position of the bug within the search space, thus explicitly defining the boundaries. If at all possible, we attempted to model the seeded faults on the corresponding actual bugs described in the Mozilla project bug report repository and version control system. For instance, when seeding a "wrong conditional" bug, if the relevant Mozilla bug involved a less-than that should have been a greater-than, we seeded the fault by replacing a correct greater-than with a buggy less-than.

Automatic fault seeding and mutation operators are well-established in practice (e.g., for measuring test suite efficacy), but were not appropriate for this study. First, automatic fault seeding approaches may not create faults from of the fault categories considered in this study. Second, they may not create faults with the desired frequency distribution. Third, and importantly for a human study, automatically seeded faults may not mimic the surrounding coding style. For example, replacing one statement with another randomly chosen statement might result in contextually or stylistically "glaring" code that could be easily identified. This last concern is difficult to formalize, but our direct experience is that automatic fault seeding often produces code that was clearly not written by a human, especially in pristine textbook code.

Using this approach, we seeded 35 of the 45 textbook files with a single defect each. The 35 chosen included the fifteen classes that showed the same topic in multiple textbooks, as well as twenty of the thirty classes taken at random. Ten of the classes were left with no faults to serve as a baseline and experimental control. To define the twenty line search space for the no-fault cases we again chose a random line in the code and randomly positioned the search window around. Each human was randomly assigned 30 of the 45 instances.

D. Participant Selection

We desire human participants who are relatively indicative of industrial practice, and thus favor programmers who are at least as competent as graduating seniors seeking CS jobs. Notably, "CS100" students are not a good fit for this study, as they may fail to localize faults for reasons that are not reflective of industrial practice (i.e., not understanding what a balanced tree is or otherwise lacking domain knowledge). We initially recruited over 215 potential human subjects for the study. All human subjects were required to have at least some level of self-reported Java programming experience. The subjects came from two broad groups: 14 undergraduates at the University of Virginia, and 201 Internet users participating via Amazon.com's Mechanical Turk website. A complete run-through took 103 minutes on average, or about three minutes per fault localization task.

The Mechanical Turk "crowdsourcing" website allows users to post jobs anonymously and workers to earn money completing them. Previous work has successfully employed this method of gathering human study participants when large and diverse sets of subjects were needed [15], [26]. While such websites allow for many potential users, care must be taken to safeguard participant quality: participants seeking to "game the system" for money must be removed.

We thus removed a number of would-be participants who did not meet various criteria. First, all participants who failed to complete the entire study were removed. Second, all participants with an overall accuracy less than that of random guessing (about 5%) were removed. Third, all participants who completed the study in under 52 minutes (i.e., under half the

average time) were removed. Ultimately, we collected usable data from 65 participants attempting to localize bugs in 30 code segments each, a dataset of 1830 human judgments.

TABLE II
PARTICIPANT SUBSETS AND AVERAGE ACCURACIES. THE COMPLETE HUMAN STUDY INVOLVED $n = 65$ PARTICIPANTS.

Subset	Average Accuracy	Number of Participants
All	46.3%	65
Accuracy $> 40\%$	55.2%	46
Experience > 4 years	51.5%	34
Experience ≥ 4 years	49.9%	51
Experience $= 4$ years	46.7%	17
Experience < 4 years	33.4%	14
Baseline: Guess Longest Line	6.3%	-
Baseline: Guess Randomly	<5.0%	-

Table II presents the average accuracies of several subsets in addition to some relevant baselines. The "Experience" measure is self-reported and includes college years. Given a search space of 20 lines as described in Section IV, guessing a random line for each code excerpt yields a baseline of at most 5% accuracy. A naïve approach of guessing the longest line for each excerpt yields only slightly higher accuracy. The "Experience $= 4$ years" row includes students from a 400-level CS class at the University of Virginia, and represents competent programmers entering the workforce within a year.

V. EXPERIMENTAL RESULTS

This section details the results of the human study in terms of a number of statistical analyses and models. We have identified several subsets of the overall participant pool for the purpose of making distinctions based on experience and quality of data. As Table II shows, the average accuracy between participants with four years of experience and participants with less than four years of experience is 40%. We often restrict attention to those participants with at least 40% accuracy; it is a natural cutoff that intuitively corresponds to entry-level industrial expertise. By contrast, the fourth-year average accuracy of 46.7% actually excludes about half of the forth-year students. While we present results for several subsets of the overall participant set, we focus mainly on this "more accurate" subset in an attempt to generalize the results to at least entry level industrial programmers and to discount participants that did not put forth sufficient effort to mimic the actual fault localization process.

A. Bug Type as Related to Fault Localization

We hypothesize that the type of the seeded fault contributes to the ease with which a human can find it. Figure 3 presents an empirical evaluation of fault localization accuracy as a function of defect type for our human study data. The error bars represent one standard deviation. For example, when presented with textbook code seeded with a "wrong type" error in a 20-line window, humans were able to identify the line containing the error in only 40% of instances.

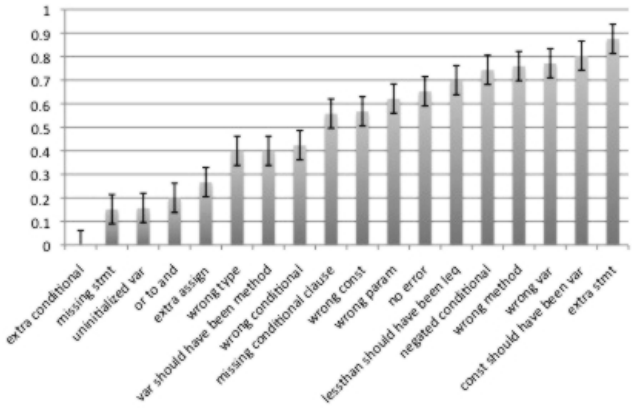

Fig. 3. Human fault localization accuracy as a function of defect type. Data set reflects 46 participants who achieved over 40% accuracy. Error bars represent one standard deviation. A higher bar indicates that human subjects were more accurate at localizing the given type of fault, ignoring the surrounding code context.

The faults listed in Figure 3 represent our expanded interpretation of Knight and Ammann's basic taxonomy. Faults can be characterized as omissions, erroneous inclusion, or incorrect choice of constant, variable, conditional, or method call. Additionally, "no error" served as a control.

We also wish to identify the differences between faults that account for differences in localization accuracy. Figure 3 shows that certain faults were found easily while other faults were more difficult, if not impossible, to find in our study. The fault type that was hardest to localize involved the inclusion of an extra conditional. We found that participants attempting to find this type of bug failed to do so in all cases, but generally reported a line within 3 lines of the actual fault site. In one localization task, such a bug was seeded in an if-statement that was followed immediately by a variable reassignment and the increment of a counter. We hypothesize that participants overlooked the conditional statement and assumed the bug occurred in the imperative statements that explicitly changed the program state. Thus it would appear that programmers are less accurate at debugging strongly imperative code; we return to this issue formally in Section V-C where a high ratio of variable assignments to constants is shown to be a strong predictor of low accuracy. Comparatively, bugs involving an extraneous statement were found most often. We found that even the less-experienced, less-accurate participants often localized bugs of this type.

From these results we conclude that certain types of bugs are intrinsically easier to localize based on their nature and recognizability, even across varying levels of programmer ability and varying code contexts.

B. Modeling Fault Localization Accuracy

Having established a strong relationship between the type of defect and human fault localization accuracy, we now train and evaluate a model of human fault localization accuracy using only contextual features. For this model we include Surface and Syntax, Control Flow, and Abstraction features, but not

Defect Category information. Each of the 1830 human judgments can potentially serve as a testing or a training instance. Once trained, a model can be used to predict human fault localization accuracy for a given piece of code. We use the Pearson product-moment correlation coefficient (i.e., Pearson's r) between the actual and predicted human performance on a testing set to evaluate a model.

In addition to our full model, we also evaluate our model when it is restricted to those features which can be obtained automatically.[2] In particular, determining the values of Abstraction features such as "uses a tree" are the subject of ongoing research and are not included in an automatic model. Instead, only features that can be obtained from a lexer, parser, control flow graph or type checker are included.

We also present three baseline software quality metrics and evaluate their correlation with human accuracy. The first of these is the automated software *Readbility* metric of Buse and Weimer [6]. That metric, based on a human study of 120 participants, measures human readability independent of code complexity. Low automated readability has been shown to correlate with defect density (as reported by automated bug-finding tools) and with code churn (as reported by version control systems). A more thorough discussion of this readability metric is given in Section VI. In our terminology, it uses only Surface and Syntax features. Intuitively, it should be harder to localize faults in less readable code.

The second baseline is McCabe's *Cyclomatic* complexity metric [20]. The stated goal of Cyclomatic complexity is to "provide a quantitative basis for modularization and allow [for identification of] software modules that will be difficult to test or maintain." Identifying code that is difficult to maintain is central to this paper and thus Cyclomatic complexity makes an appropriate comparison. We discuss Cyclomatic complexity in more detail in Section VI. In our terminology, Cyclomatic complexity uses only Control Flow features. Intuitively, it should be harder to localize faults in more complex code.

Our final baseline is a direct measure of the *textbook* presentation order of the underlying code excerpt, as determined by the textbook author. In practice, much textbook code takes the form of "projects" or "integrated running examples" that span entire chapters. As a result, this value was computed with chapter-level granularity: a Java class file associated with chapter seven in a ten chapter textbook is given a value of 0.7. This metric can be viewed as an admittedly-rough approximation to our Abstraction features. Intuitively, it should be harder to localize faults in code from the end of a course.

Figure 4 shows the results. Pearson's correlation ranges from 0.0 (no correlation) to 1.0 (maximal correlation). A typical accepted interpretation is that 0.0 to 0.1 represents no correlation, 0.1 to 0.3 represents low correlation, 0.3 to 0.5 represents medium correlation, and 0.5 to 1.0 represents large correlation [8]. As an upper bound, previous work has shown that 0.5 is "considered to have moderate to strong correlation for a human study" [10, pp.281–282]. Of particular interest are the "Accurate" and "Experienced" groups, which are presumed most indicative of industrial practice. For example, our model's predictions correlate with the performance of "Accurate" study participants with $r = 0.48$: a very high "medium" correlation ("strong" or "large" requires $r \geq 0.5$).

[2]See http://www.cs.virginia.edu/∼zpf5a/textFaultLoc/model.html for further details on both versions of the model

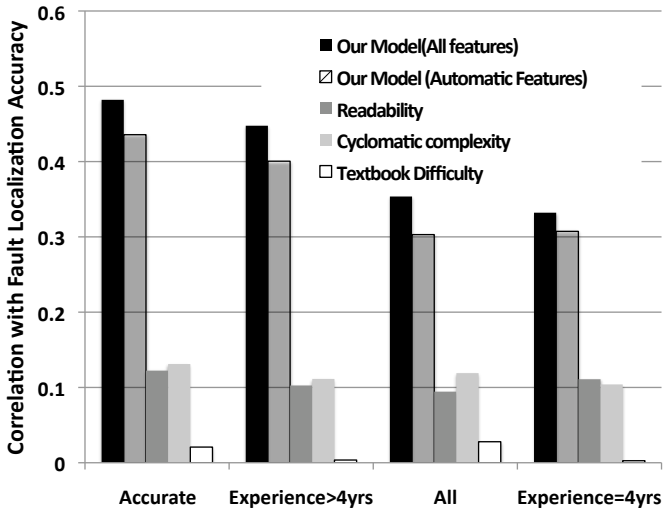

Fig. 4. Pearson correlation of human accuracy when debugging and various software quality metrics. Results are split into different groups of human participants (see Table II). A result at or below 0.1 indicates no correlation, a result above 0.3 is medium, and above 0.5 is strong [8].

Accuracy in fault localization is the primary variable of interest in this study. Our model outperforms the baselines by between 3x and 5x over all relevant subsets of participants. While the existing metrics have been proven useful in specific situations, our results show that they are not particularly effective at predicting fault localization accuracy in this concrete software engineering task. Software quality metrics are ultimately used on production code which will have to be debugged and maintained, and thus we feel that our results are particularly compelling. The comparatively strong performance of our tool suggests that it would be an effective model if used in commercial or industrial practice to measure one aspect of the future maintainability of a code base.

It is also notable that when we restricted our model of software quality to only automatically extractable features, it still performs significantly better than the other baselines. This provides some evidence that our model could be readily adapted into a useful automatic tool.

Figure 5 shows the correlations between human-perceived difficulty and each metric. One of the commonly stated goals of software quality metrics such as readability or Cyclomatic complexity is that of accurately measuring the understandability of code. In our human study we specifically asked humans to rate how "difficult the given piece of code is to understand and debug". While fault localization accuracy is a concrete measurement of a software engineering maintenance task, human-perceived difficulty is what many software quality

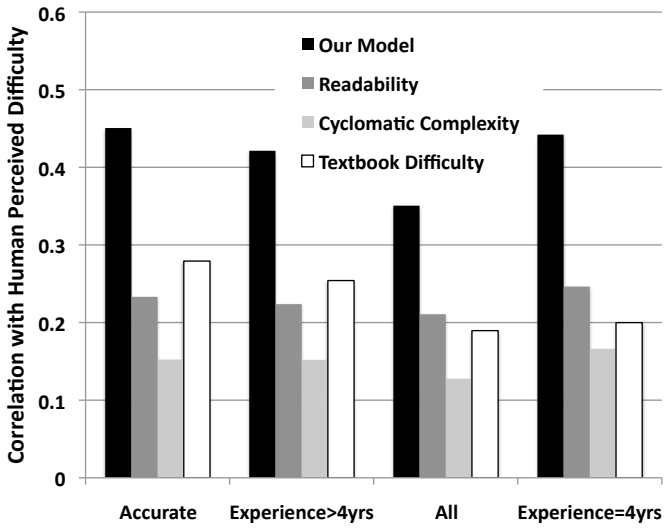

Fig. 5. Correlation of human-perceived difficulty when debugging and various software quality metrics. Results are split into different groups of human participants (see Table II). A result at or below 0.1 indicates no correlation, a result above 0.3 is medium, and above 0.5 is strong [8].

metrics are trying to measure and therefore provides a complementary comparison. As Figure 5 shows, our model correlates with human-perceived difficulty and fault localization accuracy in a similar fashion. Readability, Cyclomatic complexity and textbook positioning (in particular) all correlate somewhat better with perceived difficulty than they do with fault localization accuracy, but all continue to fall below the 0.3 threshhold. We do not include an automatic version of our model for perceived difficulty because we view it as a strictly human judgment that does not require automated tool support.

To detect and mitigate bias resulting from over-fitting by testing and training on the same dataset, we performed 10-fold cross validation [17] for all versions of our model. In 10-fold cross validation, the data is divided randomly into ten subsets (folds) and the model is repeatedly trained on nine and tested on the tenth. In this way the model is never tested and trained on the same data. If the results of cross validation are significantly different from the results of testing and training on the same data, it indicates bias due to over-fitting. Our experiments revealed little to no over-fitting (i.e., $r = 0.482$ without cross validation and $r = 0.474$ with it).

C. Code Features as Predictors of Localization Accuracy

Existing metrics make specific assumptions about which code features are relevant to the notion of software quality. For example, Cyclomatic complexity assumes that only control flow features matter, while Readability uses only textual surface features. We perform an analysis of variance on our formal model of fault localization accuracy to determine the relative predictive power of features for this task.

Table III shows the results. The most powerful feature was "uses abstraction: array", which applies to code that makes use of an implementation of an array datatype (e.g., Java's built-in array type, or the Vector class). While minor uses of arrays may not be difficult, the heavy use of arrays often correlated with the implementation of a data structure such as a hash map or B-tree. By contrast, "provides abstraction: X" is used when a file implements and otherwise adheres to the interface for the given datatype or abstraction. For example, code that provides a priority queue of nodes merits "provides abstraction: queue".

TABLE III
ANALYSIS OF VARIANCE OF OUR FEATURES WITH RESPECT TO HUMAN FAULT LOCALIZATION ACCURACY. A HIGH F-VALUE INDICATES A RELATIVELY PREDICTIVE FEATURE (1.0 MEANS "NOT PREDICTIVE"). THE p VALUE ROUGHLY INDICATES THE STATISTICAL SIGNIFICANCE OF THAT FEATURE'S PREDICTIVE POWER (I.E., $p < 0.05$ IS SIGNIFICANT). THE FINAL COLUMN INDICATES WHETHER THE CORRELATION OF THE GIVEN FEATURE IS POSITIVE OR NEGATIVE WITH RESPECT TO ACCURACY. PREFIXES DENOTE THE CLASSIFICATION OF THE FEATURE: SYNTAX AND SURFACE (SYN), CONTROL FLOW (CFG), ABSTRACTION (ABS).

Feature	F	Pr(F)	Dir
abs – uses abstraction: array	130.9	< 0.001	-
abs – provides abstraction: queue	54.1	< 0.001	+
syn – max complex conditionals	46.5	< 0.001	+
syn – ratio const to var assignments	40.4	< 0.001	+
syn – avg block nesting level	38.9	< 0.001	-
abs – provides abstraction: heap	28.3	< 0.001	+
abs – provides abstraction: stack	27.0	< 0.001	+
syn – max global variables	25.6	< 0.001	+
abs – uses abstraction: linked list	25.6	< 0.001	-
syn – ratio simple to const cond	20.6	< 0.001	-
syn – max local varabiles	19.2	< 0.001	+
syn – ratio primitives to nonprims	10.5	0.001	+
abs – uses list-insert algorithm	10.0	0.002	-
cfg – max CFG out edges per node	10.0	0.002	-
abs – uses abstraction: queue	8.9	0.003	+
syn – max block nesting level	7.4	0.007	+
syn – max for loops	6.0	0.014	+
cfg – avg CFG in edges per node	5.8	0.016	+
syn – avg code line length	4.6	0.031	+

A number of features merit a detailed investigation or explanation. Every feature detailing "provided abstraction" signifies that the code in question is implementing a certain data structure. Comparatively, "uses abstraction" means that the code itself is utilizing a certain data structure. For example, code providing a heap type abstraction might use an array abstraction for ease of implementation. We also explain the three additional potentially ambiguous features in detail.

- "max complex conditionals" — the maximum total number of conditional clauses with at least one method call
- "ratio const to var assignments" — the ratio of the number of assignments made from constant values to the number of assignments where one variable is being assigned from another variable
- "ratio simple to const cond" — the ratio of the number of conditional clauses involving variables to the number of conditional clauses that involve only constant values

All syntax, surface and control flow features were measured with parser- and lexer-style tools. All abstraction-based features were measured from source code or program representations as objectively as possible. Otherwise-difficult judgments such as the presence or absence of recursion or the presence of absence of a heap were straightforward (recall that this is code for which the corresponding textbook is available).

Supporting our hypothesis that a comprehensive model would utilize a diverse set of features, we found that the most predictive features included all three types of features. More specifically, nine of the top twenty most predictive features were from the abstract category. Previous metrics, such as readability and Cyclomatic complexity, do not take this category into account. This further reinforces the results presented in Section V-B by showing that previous metrics overlook crucial code characteristics entirely.

In conclusion, this analysis of variance not only shows that a diverse, aggregate set of features most accurately models concrete software quality but that many of the most important features have not been explored by previous models.

D. Threats to Validity

Although our results are statistically significant and suggest that we can model software quality more accurately than the state of the art, they may not generalize to industrial practice.

Given our intuition that fault localization requires programmers to understand the code and its specification, there are a number of threats related to failing to control for code complexity, code readability, and participant understanding of the specifications involved. We attempted to mitigate all of these threats by selecting our baseline contextual code from textbooks; see Section IV-B for details. While code taken from textbooks may not be directly comparable to industrial code, we assert that the examples used in this study illustrate many of the basic principles of computer science and thus form the building blocks for large real-life systems.

There are several threats to validity associated with the manner in which we seeded the faults into the subject code. Seeding faults manually in code for such a human study inherently introduces bias based on fault type, location, and the choice of which programmatic elements to modify. To mitigate this bias we used a real-life fault type distribution (based on the Mozilla project), used an extended version of a well-established fault taxonomy [16], randomly generated line numbers as potential locations, and changed the closest viable Java constructs based on the associated fault template.

Another potential threat of the experimental results lies in the selection of model features. In particular, bias may be introduced if the features are selected based on the specific subject code and types of faults used in our study. To circumvent this potential threat, we developed and permanently fixed our model's features independently and strictly prior to the design and implementation of the human study — and thus strictly prior to the fault seeding effort. This helps to eliminate bias from a *post hoc* cherry-picking of features based on how the faults were seeded or how the humans performed. An additional safeguard against over-fitting is the use of cross-validation, as described in Section V.

A common pitfall of human studies is a "training effect" wherein the participants do significantly worse at the beginning of the study due to unfamiliarity with the task. This can potentially invalidate results drawn from the affected data. To address this concern we measured the overall accuracy for the first and second halves of all participants' responses. On average, participants were 46.8% accurate for the first half of the code excerpts and 45.6% for the second. This suggests that our results do not suffer from a training effect (or the converse "fatigue effect").

Participant selection can also provide bias in a human study. Results from an exit survey suggest that this bias does not affect our code. Besides the computer science undergraduate students who volunteered, the Mechanical Turk participants self-reported experience indicate that participants had from 1–20 years of programming experience, 1–10 years of computer science academic experience, and 1–20 years of industrial experience. Additionally, the use of the Mechanical Turk interface inherently randomizes the participant selection process which further mitigates this particular threat. However, our strongest guard against such bias is the analysis of subsets of participants based on independent metrics, such as "all participants with $\geq 40\%$ accuracy". Using such subsets prevents us from having to trust self-reported experience levels at all. See Kittur *et al.* [15] and Snow *et al.* [26] for an analysis of the qualities and biases associated with using such participants.

VI. Related Work

Boehm *et al.* pioneered an initial study of several software metrics, in essence defining the concept of software quality [5]. Their study can be viewed as a logical precursor for future work, including ours, in that it laid that framework for the modern notion of software quality. Our work builds upon it by taking human actions and insights into account.

Prabhakararao and Ruthruff *et al.* performed two human studies to gauge the effectiveness of an interactive fault localization tool developed for end users with little to no experience [21], [24]. The goal of their studies was to evaluate the use of feedback when locating faults and to generally study the process of fault localization, especially by users with no expert domain knowledge of the source. By comparison, our human study also examines the fault localization process but for the purpose of evaluating software quality metrics. We are less interested in the specific process and more concerned with the resulting accuracy and the human intuitions about the code in question. Additionally, our human study is of a much broader scope and thus we hope it is more generalizable.

Barkmann *et al.* have presented preliminary work that aims to validate and compare several software quality metrics automatically over many open-source projects [3]. Our work presents a human study that implicitly evaluates software quality metrics, such as readability and complexity, based on data gathered from human fault localization performance and not static code measurements.

McCabe's Cyclomatic complexity metric [20] is well known in the field. The stated goal of Cyclomatic complexity is to "provide a quantitative basis for modularization and allow [for identification of] software modules that will be difficult to test or maintain." Intuitively, this metric identifies non-linear code with high levels of control flow graph interconnectivity as being "more complex". The original authors assert that this

notion of complexity correctly adheres to the previously stated goal. We explicitly compare to Cyclomatic complexity in our experimental section, finding that it correlates at best weakly with human fault localization accuracy. Additionally, in our terminology, it uses only Control Flow features.

Buse and Weimer developed an automated metric of readability, derived from a human study of 120 annotators evaluating 100 pieces of code each [6]. Readability is defined as a human judgment as to how understandable a piece of code is. Thus, it is tied to the maintainability and ease of debugging of code as well. It uses only syntactic properties as features which allows for an overall lightweight and fast approach as well as easy automation. This previous work has shown that readability is positively correlated with several concrete notions of software quality. Specifically, the work compared the constructed metric with that of defects found using the popular debugging tool FindBugs [13] and with future code changes or "code churn". Readability uses only Surface and Syntax features. We explicitly compare against this automated metric in our experimental section, and find that it does not correlate with human fault localization accuracy.

VII. Conclusion

Fault localization is an important software engineering activity. However, not all programs and not all bugs are equally easy to debug. Intuitively, developers must understand a program and its specification to locate defects. We hypothesize that the type of defect, surface features (e.g., readability), control flow features (e.g., the number of program paths), and abstraction features (e.g., the presence of various datatypes) can be used to model human fault localization accuracy.

Software metrics are important, but can be harmful if misapplied. Readability and complexity metrics are widely used, but may not match up with user intuition if they are mistakenly used as proxies for the difficulty of software engineering tasks.

We present formal models using those features, backed by a human study involving 65 participants and 1830 total judgments. To the best of our knowledge, this is the first published human study of developer fault localization accuracy. Our study involves example code from Java textbooks, helping us to control for both readability and complexity. We find that certain types of defects are much harder for humans to locate accurately. For example, humans are over five times more accurate at locating "extra statements" than at "missing statements". We also find that, independent of the type of defect involved, certain code contexts are harder to debug than others. For example, humans are over three times more accurate at finding defects in code that uses array abstractions than in code that uses tree abstractions. Using our features, our formal model has a moderate to strong ($r = 0.48$) correlation with actual human accuracy.

Acknowledgments

The authors would like to thank Raymond P.L. Buse and Pieter Hooimeijer for their help. This research was supported by, but does not reflect the views of, National Science Foundation Grants CCF 0954024, CCF 0916872, CNS 0716478, CNS 0627523 and Air Force Office grant FA9550-07-1-0532, as well as gifts from Microsoft.

References

[1] J. Anvik, L. Hiew, and G. C. Murphy. Coping with an open bug repository. In *OOPSLA workshop on Eclipse technology eXchange*, pages 35–39, 2005.

[2] T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: Localizing errors in counterexample traces. In *In Principles of Programming Languages*, pages 97–105, 2003.

[3] H. Barkmann, R. Lincke, and W. Lowe. Quantitative evaluation of software quality metrics in open-source projects. In *International Conference on Advanced Information Networking and Applications Workshops*, pages 1067–1072, 2009.

[4] B. Boehm and V. R. Basili. Software defect reduction top 10 list. *Computer*, 34(1):135–137, 2001.

[5] B. W. Boehm, J. R. Brown, and M. Lipow. Quantitative evaluation of software quality. In *International conference on Software engineering*, pages 592–605, 1976.

[6] R. P. L. Buse and W. Weimer. A metric for software readability. In *International Symposium on Software Testing and Analysis*, 2008.

[7] F. M. Carrano. *Data Structures and Abstractions with Java, 2nd edition*. Prentice Hall, 2006.

[8] J. Cohen. *Statistical power analysis for the behavioral sciences, 2nd ediditon*. Routledge Academic, 1988.

[9] P. Drake. *Data Structures and Algorithms in Java*. Prentice Hall, 2005.

[10] L. L. Giventer. *Statistical Analysis for Public Administration*. 2007.

[11] M. J. Harrold, A. J. Offutt, and K. Tewary. An approach to fault modeling and fault seeding using the program dependence graph. *Journal of Systems and Software*, (3):273–296, March 1997.

[12] P. Hooimeijer and W. Weimer. Modeling bug report quality. In *Automated software engineering*, pages 34–43, 2007.

[13] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *Object-oriented programming systems, languages, and applications companion*, 2004.

[14] J. A. Jones and M. J. Harrold. Empirical evaluation of the Tarantula automatic fault-localization technique. In *Automated Software Engineering*, pages 273–282, 2005.

[15] A. Kittur, E. H. Chi, and B. Suh. Crowdsourcing user studies with mechanical turk. In *Conference on Human Factors in Computing Systems*, pages 453–456, 2008.

[16] J. C. Knight and P. Ammann. An experimental evaluation of simple methods for seeding program errors. In *International Conference on Software Engineering*, pages 337–342, 1985.

[17] R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. *International Joint Conference on Artificial Intelligence*, 14(2):1137–1145, 1995.

[18] R. Lafore. *Data Structures and Algorithms in Java, 2nd edition*. 2002.

[19] J. Lewis and J. Case. *Java Software Structures: Designing and Using Data Structures, 2nd edition*. Addison-Wesley, 2005.

[20] T. J. McCabe. A complexity measure. *IEEE Trans. Software Eng.*, 2(4):308–320, 1976.

[21] S. Prabhakararao, C. Cook, J. Ruthruff, E. Creswick, M. Main, M. Durham, and M. Burnett. Strategies and behaviors of end-user programmers with interactive fault localization. In *Human Centric Computing Languages and Environments*, pages 15–22, 2003.

[22] D. R. Raymond. Reading source code. In *Conference of the Centre for Advanced Studies on Collaborative Research*, pages 3–16, 1991.

[23] S. Rugaber. The use of domain knowledge in program understanding. *Ann. Softw. Eng.*, 9(1-4):143–192, 2000.

[24] J. R. Ruthruff, M. Burnett, and G. Rothermel. An empirical study of fault localization for end-user programmers. In *International conference on Software engineering*, pages 352–361, 2005.

[25] W. Savitch. *Absolute Java, 3rd edition*. Addison-Wesley, 2007.

[26] R. Snow, B. O'Connor, D. Jurafsky, and A. Y. Ng. Cheap and fast—but is it good?: evaluating non-expert annotations for natural language tasks. In *Empirical Methods in Natural Language Processing*, 2008.

[27] W. Weimer and G. C. Necula. Finding and preventing run-time error handling mistakes. In *Object-oriented programming, systems, languages, and applications*, pages 419–431, 2004.

[28] S. Zeil. Perturbation techniques for detecting domain errors. *IEEE Transactions on Software Engineering*, 15:737–746, 1989.

[29] A. Zeller. Yesterday, my program worked. Today, it does not. Why? In *Foundations of Software Engineering*, pages 253–267, 1999.