Automatically Generated Patches as Debugging Aids: A Human Study

Yida Tao[1], Jindae Kim[1], Sunghun Kim*,[1], and Chang Xu*,[2]

[1]Dept. of Computer Science and Engineering, The Hong Kong University of Science and Technology, Hong Kong, China
[2]State Key Lab for Novel Software Technology and Dept. of Computer Sci. and Tech., Nanjing University, Nanjing, China
{idagoo,jdkim,hunkim}@cse.ust.hk,changxu@nju.edu.cn

ABSTRACT

Recent research has made significant progress in automatic patch generation, an approach to repair programs with less or no manual intervention. However, direct deployment of auto-generated patches remains difficult, for reasons such as patch quality variations and developers' intrinsic resistance.

In this study, we take one step back and investigate a more feasible application scenario of automatic patch generation, that is, using generated patches as debugging aids. We recruited 95 participants for a controlled experiment, in which they performed debugging tasks with the aid of either buggy locations (i.e., the control group), or generated patches of varied qualities. We observe that: a) high-quality patches significantly improve debugging correctness; b) such improvements are more obvious for difficult bugs; c) when using low-quality patches, participants' debugging correctness drops to an even lower point than that of the control group; d) debugging time is significantly affected not by debugging aids, but by participant type and the specific bug to fix. These results highlight that the benefits of using generated patches as debugging aids are contingent upon the quality of the patches. Our qualitative analysis of participants' feedback further sheds light on how generated patches can be improved and better utilized as debugging aids.

Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]: Debugging aids

General Terms

Experimentation, Human Factors

Keywords

Debugging, automatic patch generation, human study

*Corresponding authors

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
FSE '14, November 16–22, 2014, Hong Kong, China
Copyright 2014 ACM 978-1-4503-3056-5/14/11 ...$15.00.

1. INTRODUCTION

Debugging is of paramount importance in the process of software development and maintenance, given tens or even hundreds of bugs reported daily from individual projects [2]. However, debugging is dauntingly challenging, as bugs can take days or even months to resolve [2][3].

To alleviate developers' burden of fixing bugs, *automatic patch generation* is proposed to repair programs with less or no manual intervention. Recent research in this area has made significant progress [6][17][21][22][26][29][33]. Le Goues et al. proposed a technique based on evolutionary computation and used it to successfully repair 55 out of 105 real-world bugs [21]. Other advanced techniques, such as runtime modifications [6][29], program synthesis using code contracts and formal specifications [26][33], and pattern adaptation [17] have also produced promising results in terms of the repair success-rate.

Despite these fruitful research outcomes, direct deployment of auto-generated patches in production code seems unrealistic at this point. Kim et al. suggested that generated patches are sometimes non-sensical and thus less likely to be accepted by developers [17]. Research has also identified other reasons why developers are not comfortable with blindly trusting auto-generated code. For example, generated code might be less readable and maintainable [7][13].

Rather than direct deployment, a more feasible application of auto-generated patches would be to aid debugging, since they not only pinpoint buggy locations, but also suggest candidate fixes. Yet, developers can still judge whether candidate fixes are correct and whether to adapt or discard generated patches.

A natural question thus arises from the above scenario: *are auto-generated patches useful for debugging*? In addition, generated patches vary in quality, which affects both the functional correctness and future maintainability of the program being patched [13]. Research has also found that the quality of automated diagnostic aids affects users' reliance and usages of them [23]. As such, *does a patch's quality affect its usefulness as a debugging aid*?

In this paper, we conduct a large-scale human study to address these two questions. We recruited 95 participants, including 44 graduate students, 28 software engineers, and 23 Amazon Mechanical Turk (MTurk) [1] users to individually fix five real bugs. Each bug was aided by one of the three hints: its *low-quality* or *high-quality* generated patch, or its buggy location (method name) as the control condition. In total, we collected 337 patches submitted by the participants.

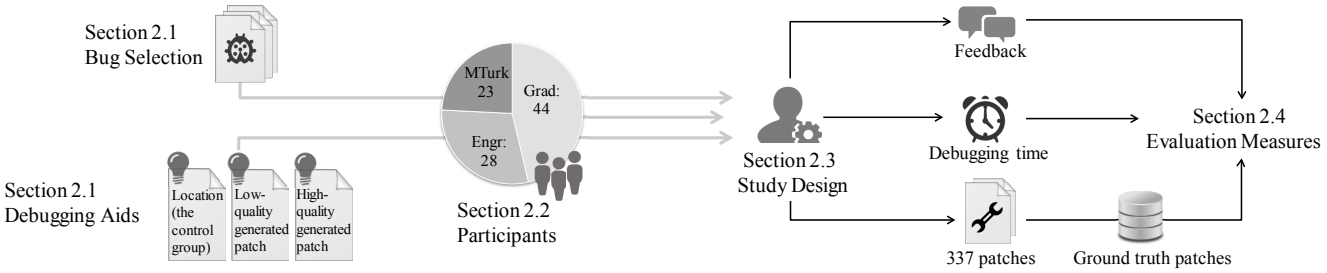


Figure 1: Overview of the experimental process. We create debugging tasks using five real bugs. Each bug is aided by one of the three hints: its generated patch of low or high quality, or its buggy location. Our participants include computer science graduate students, software engineers, and Amazon Mechanical Turk (MTurk) users. Graduate students debug using the Eclipse IDE in onsite sessions while software engineers and MTurk participants debug using our web-based debugging system. We evaluate participants' patch correctness, debugging time, and exit-survey feedback.

We observe a significant increase in the debugging correctness when participants were aided by the high-quality generated patches, and such an improvement was more obvious for difficult bugs. Nevertheless, participants were adversely influenced by the low-quality generated patches, as their debugging correctness dropped surprisingly to an even lower point than the control group. This finding urges a strict quality control for generated patches if we were to use them as debugging aids. Otherwise, incorrect or unmaintainable patches can indeed cloud developers' judgement and deteriorate their debugging correctness.

Interestingly, the types of debugging aid have only marginal influence on participants' debugging time, while the types of bugs and participant populations instead have greater influence. For example, we observe a significant slowdown in debugging time for the most difficult bug, and a significant speed-up for software engineers and MTurk participants compared to graduate students.

We also qualitatively analyze participants' attitudes toward using auto-generated patches. Participants gave generated patches credit for quick problem identification and simplification. However, they remained doubtful about whether generated patches can fix complicated bugs and address root causes. Participants also claimed that they could have been more confident and smooth in using generated patches if they had deeper understanding of the buggy code, its context, and its test cases.

Overall, this paper makes the following contributions.

- A large-scale human study for assessing the usefulness of auto-generated patches as debugging aids.
- An in-depth quantitative evaluation on how debugging is influenced by the quality of generated patches and other factors such as the types of bugs and participants.
- A qualitative analysis of patch usages, which sheds light on potential directions of automatic patch generation research.

The remainder of this paper is organized as follows. Section 2 introduces our experimental process. Section 3 and Section 4 report and discuss the study results. Section 5 presents threats to validity, followed by a survey of related work in Section 6. Section 7 concludes the paper.

2. EXPERIMENTAL PROCESS

This section presents detailed experimental process following the overview of Figure 1.

2.1 Bug Selection and Debugging Aids

To evaluate the usefulness of auto-generated patches in debugging, we select bugs that satisfy the following criteria:

- To simulate real debugging scenarios, real-world bugs are favored over seeded ones [14] and the diversity of bug types is preferred.
- Patches of varied qualities can be automatically generated for the chosen bugs.
- Patches written and accepted by the original developers should be available as the ground truth to evaluate participants' patches.
- A proper number of bugs should be selected to control the length of the human study [16].

Accordingly, we selected five bugs reported by Kim et al. [17] and summarize them in Table 1. Four of the bugs are from Mozilla Rhino[1] and one is from Apache Commons Math[2]. These bugs manifest different symptoms and cover various defect types, and all of them have been fixed by developer-written and verified patches.

Kim et al. reported that all five bugs can be fixed by two state-of-the-art program repair techniques, GenProg [21] and PAR [17], such that their generated patches can pass all the corresponding test cases [17]. Kim et al. launched a survey, in which 85 developers and CS students were asked to rank the acceptability of the generated patches for each bug [17]. Table 2 reports these ten generated patches (two patches for each bug) along with their average acceptability rankings reported by Kim et al. [17].

While the generated patches are equally qualified in terms of passing test cases, humans rank patch acceptability by further judging their fix semantics and understandability, which are the primary concerns of patch quality [13][15]. Hence, we inherited this acceptability ranking reported by Kim et al. [17] as an indicator for patch quality. Specifically, for the two patches generated for each bug, we labeled the one with a higher ranking as *high-quality* and the other one with a lower ranking as *low-quality*.

Since generated patches already reveal buggy locations and suggest candidate fixes, for fair comparison, we provide names of buggy methods to the control group instead of leaving it completely unaided. This design conforms to real debugging scenarios where developers are aware of basic faulty regions, for example, from detailed bug reports [14].

[1]http://www.mozilla.org/rhino/

[2]http://commons.apache.org/proper/commons-math/

Table 1: Description of the five bugs used in the study. We list semantics of developers' original fixes, which are used to evaluate the correctness of participants' patches.

Bug#	Buggy code fragment and symptom	Semantics of original developers' patch

Math-280	`1 public static double[] bracket(...,`
	`2 double lowerBound, double upperBound,...)`
	`3 if (fa * fb >= 0.0 ) {`
	`4 throw new ConvergenceException(`
	`5 ...`
	`ConvergenceException` is thrown at line 4.
		• If `fa*fb == 0.0`, then the method `bracket()` should terminate without `ConvergenceException`, regardless of given `lowerBound` and `upperBound` values.
		• If `fa*fb > 0.0`, then the method `bracket()` should throw `ConvergenceException`.

Rhino-114493	`1 if (lhs == undefined) {`
	`2 lhs = strings[getShort(iCode, pc + 1)];`
	`3 }`
	`ArrayIndexOutOfBoundsException` is thrown at line 2.
		• If `getShort(iCode, pc + 1)` returns a valid index, then `strings[getShort(iCode, pc + 1)]` should be assigned to `lhs`.
		• If `getShort(iCode, pc + 1)` returns -1, the program should return normally without `AIOBE`, and `lhs` should be `undefined`.

Rhino-192226	`1 private void visitRegularCall(Node node, int`
	`2 type, Node child, boolean firstArgDone)`
	`3 ...`
	`4 String simpleCallName = null;`
	`5 if (type != TokenStream.NEW){`
	`6 simpleCallName = getSimpleCallName(node);`
	`7 ...`
	`8 child = child.getNext().getNext();`
	`NullPointerException` is thrown at line 8.
		• Statements in the block of `if` statement is executed only if `firstArgDone` is `false`.
		• `simpleCallName` should be initialized with a return value of `getSimpleCallName(node)`.

Rhino-217379	`1 for (int i = 0; i < parenCount; i++) {`
	`2 SubString sub = (SubString) parens.get(i);`
	`3 args[i+1] = sub.toString();`
	`4 }`
	`NullPointerException` is thrown at line 3.
		• If `parens.get(i)` is not `null`, assign it to `args[i+1]` for all `i = 0...parenCount-1`.
		• The program should terminate normally without `NullPointerException` when `sub` is null.

Rhino-76683	`1 for (int i = num; i < state.parenCount; i++)`
	`2 state.parens[i].length = 0;`
	`3 state.parenCount = num;`
	`NullPointerException` is thrown at line 2.
		• The program should terminate without `NullPointerException` even if `state.parens[i]` is equal to `null`.
		• If `for` statement is processing the array `state.parens`, it should be processed from index `num` to `state.parenCount-1`.

For simplicity, we refer to participants given different aids as the LowQ (low-quality patch aided), HighQ (high-quality patch aided), and Location (buggy location aided) group.

2.2 Participants

Grad: We recruited 44 CS graduate students — 12 from The Hong Kong University of Science and Technology and 32 from Nanjing University. Graduate students typically have some programming experience but in general they are still in an active learning phase. In this sense, they resemble novice developers.

Engr: We also invited industrial software engineers to represent the population of experienced developers. In this study, we recruited 28 software engineers from 11 companies via email invitations.

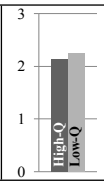
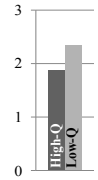
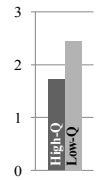
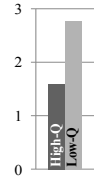
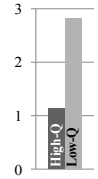
MTurk: We further recruited participants from Amazon Mechanical Turk, a crowdsourcing website for *requesters* to post tasks and for *workers* to earn money by completing tasks [1]. The MTurk platform potentially widens the variety of our participants, since its workers have diverse demographics such as nationality, age, gender, education, and income [32]. To safeguard worker quality, we prepared a buggy code revised from Apache Commons Collection Issue 359 and asked workers to describe how to fix it. Only those who passed this qualifying test could proceed to our debugging tasks. We finally recruited 23 MTurk workers, including 20 developers, two undergraduate students, and one IT manager, with 1-14 (on average 5.7) years of Java programming experience.

2.3 Study Design

We provided detailed bug descriptions from Mozilla and Apache bug reports to participants as their starting point. We also provided developer-written test cases, which included the failed ones that reproduced the bugs. Although participants were encouraged to fix as many bugs as they could, they were free to skip any bug as the completion of all five tasks was not mandatory. To conform to participants' different demographics, we adopted the *onsite* and *online* settings as introduced below.

Onsite setting applies to the graduate students. We adopted a between-group design [20] by dividing them into three groups; each group was exposed to one of the three debugging aids. Prior to the study, we asked students to self-report their Java programming experience and familiarity with Eclipse on a 5-point Likert scale. We used this information to evenly divide them into three groups with members having similar levels of expertise. Finally, the LowQ and

Table 2: The two generated patches for each bug. The middle column shows the patches' acceptability ranking reported by Kim et al. [17]. The patch with a higher ranking (or lower bar) is labeled as *high-quality* and its counterpart for the same bug is labeled as *low-quality*.

Bug	High-quality patch	Rank	Low-quality patch
Math-280	`public static double[] bracket(..., double lowerBound, double upperBound,...) if (fa * fb > 0.0 || fa * fb >= 0.0 ) { throw new ConvergenceException( ...`		`public static double[] bracket(..., double lowerBound, double upperBound,...) if (function == null ) { throw MathRuntimeException. createIllegalArgumentException( ...`
Rhino-114493	`if (lhs == undefined) { if(getShort(iCode, pc + 1) < strings. length && getShort(iCode, pc + 1) >=0) { lhs = strings[getShort(iCode,pc + 1)]; } }`		`if (lhs == undefined) { lhs = ((Scriptable)lhs).getDefaultValue (null); }`
Rhino-192226	`private void visitRegularCall(Node node, int type, Node child, boolean firstArgDone) ... String simpleCallName = null; if (type != TokenStream.NEW){ if (!firstArgDone) simpleCallName = getSimpleCallName(node); ...`		`private void visitRegularCall(Node node, int type, Node child, boolean firstArgDone) ... String simpleCallName = null; visitStatement(node);`
Rhino-217379	`for (int i = 0; i < parenCount; i++) { SubString sub = (SubString) parens.get(i); if (sub != null) { args[i+1] = sub.toString(); } }`		`for (int i = 0; i < parenCount; i++) { SubString sub = (SubString) parens.get(i); args[parenCount + 1] = new Integer(reImpl .leftContext.length); }`
Rhino-76683	`for (int i = num; i < state.parenCount; i++) if( state != null && state.parens[i] != null) { state.parens[i].length = 0; } state.parenCount = num;`		`for (int i = num; i < state.parenCount; i++) { } state.parenCount = num;`

HighQ groups each had 15 participants and the Location group had 14 participants.

The graduate students used the Eclipse IDE (Figure 2a). We piloted the study with 9 CS undergraduate students and recorded their entire Eclipse usage with screen-capturing software. From this pilot study, we determined that two hours should be adequate for participants to complete all five tasks at a reasonably comfortable pace. The formal study session was supervised by one of the authors and two other helpers. At the beginning of the session, we gave a 10-minute tutorial introducing the tasks. Then, within a maximum of two hours, participants completed the five debugging tasks in their preferred order.

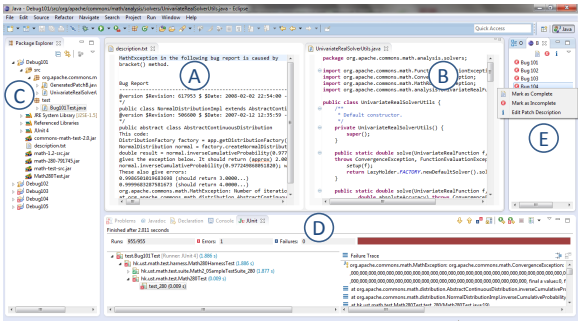
Online setting applies to the software engineers and MTurk participants who are geographically unavailable. We created a web-based online debugging system[3] that provided similar features to the Eclipse workbench (Figure 2b). Unlike Grad, it was unlikely to determine beforehand the total number of these online participants and their expertise. Hence, to minimize individual differences, we adopted a within-group design such that participants can be exposed to different debugging aids [20]. To balance the experimental conditions, we assigned the type of aids to each selected bug in a round-robin fashion such that each aid was equally likely to be given to each bug.

We piloted the study with 4 CS graduate students. They were asked to use our debugging system and report any encountered problems. We resolved these problems, such as browser incompatibility, before formally inviting software engineers and MTurk users.

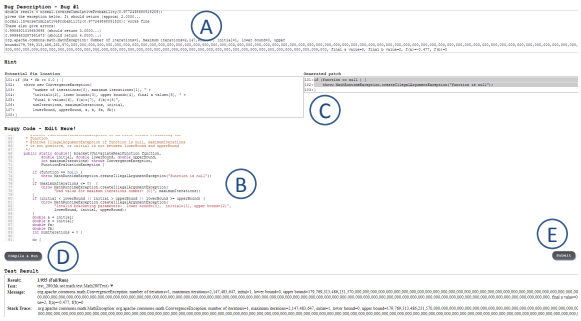
An exit survey is administered to all the participants upon their task completion. In the survey, we asked the participants to rate the difficulty of each bug and the helpfulness of the provided aids on a 5-point Likert scale. Also, we asked them to share their opinions on using auto-generated patches in a free-textual form. We asked Engr and MTurk participants to self-report their Java programming experience and debugging time for each task, since the online system cannot monitor the event when they were away from the keyboard (discussed in Section 5). In addition, we asked MTurk participants to report their occupations.

We summarize our experimental settings in Table 3.

[3]http://pishon.cse.ust.hk/userstudy/



(a) The Eclipse workbench.



(b) The online debugging system.

Figure 2: The onsite and online debugging environments, which allow participants to browse a bug's description (A), edit the buggy source code (B), view the debugging aids (C), run the test cases (D) and submit their patches (E).

2.4 Evaluation Measures

We measure the participants' patch correctness, debugging time, and survey feedback as described below.

Correctness indicates whether a patch submitted by a participant is correct or not. We considered a patch correct only if it passed all the given test cases and also matched the semantics of the original developers' fixes as described in Table 1. Two of the authors and one CS graduate student with 8 years of Java programming experience individually evaluated all patches. Initially, the three evaluators agreed on 283 of 337 patches. The disagreement mainly resulted from misunderstanding of patch semantics. After clarification, the three evaluators carried out another round of individual evaluations and this time agreed on 326 out of 337 patches. This was considered to be near-perfect agreement as its Fleiss kappa value equals to 0.956 [11]. For the remaining 11 patches, the majority's opinion (i.e., two of the three evaluators) was adopted as the final evaluation result.

Debugging time was recorded automatically. For the onsite session, we created an Eclipse plug-in to sum up the elapsed time of all the activities (e.g., opening a file, modifying the code and running the tests) related to each bug. For online sessions, our system computes participants' debugging time as the time elapsed from them entering the website until the patch submission.

Feedback was collected from the exit survey (Section 2.3). We particularly analyzed participants' perception about the helpfulness of generated patches and their free-form answers. These two items allowed us to quantitatively and qualitatively evaluate participants' opinions on using generated patches as debugging aids (Section 3.1 and Section 4).

Table 3: The study settings and designs for different types of participants, along with their population size and number of submitted patches. In cases where participants skipped or gave up some tasks, we excluded patches with no modification to the original buggy code and finally collected 337 patches for analysis.

	Settings	Designs	Size	# Patches
Grad	Onsite (Eclipse)	Between-group	44	216
Engr	Online (website)	Within-group	28	68
MTurk	Online (website)	Within-group	23	53
Total			95	337

3. RESULTS

We report participants' debugging performance with respect to different debugging aids (Section 3.1). We then investigate debugging performance on different participant types (Section 3.2), bug difficulties (Section 3.3), and programming experience (Section 3.4). We use regression models to statistically analyze the relations between these factors and the debugging performance (Section 3.5) and finally summarize our observations (Section 3.6).

3.1 Debugging Aids

Among the 337 patches we collected, 109, 112 and 116 were aided by buggy locations, LowQ and HighQ patches, respectively. Figure 3 shows an overview of the results in terms of the three evaluation measures.

Correctness (Figure 3a): Patches submitted by the Location group is 48% (52/109) correct. The percentage of correct patches dramatically increases to 71% (82/116) for the HighQ group. LowQ patches do not improve debugging correctness. On the contrary, the LowQ group performs even worse than the Location group, with only 33% (37/112) patches being correct.

Debugging Time (Figure 3b): The HighQ group has an average debugging time of 14.7 minutes, which is slightly faster than the 17.3 and 16.3 minutes of the Location and LowQ groups.

Feedback (Figure 3c): HighQ patches are considered much more helpful for debugging compared to LowQ patches. Mann-Whitney U test suggests that this difference is statistically significant (p-value $= 0.0006 < 0.05$). To understand the reasons behind this result, we further analyze participants' textual answers and discuss the details in Section 4.

3.2 Participant Types

Our three types of participants, Engr, Grad, and MTurk, have different demographics and are exposed to different experimental settings. For this reason, we investigate the debugging performance by participant types.

Figure 5a shows a consistent trend across all three participant types: the HighQ group makes the highest percentage of correct patches followed by the Location group, while the LowQ group makes the lowest. Note that Grad has a lower correctness compared to Engr and MTurk, possibly due to their relatively short programming experience.

Figure 5b reveals that Engr debugs faster than Grad and MTurk participants. Yet, among the participants from the same population, their debugging time does not vary too much between the Location, LowQ and HighQ groups.

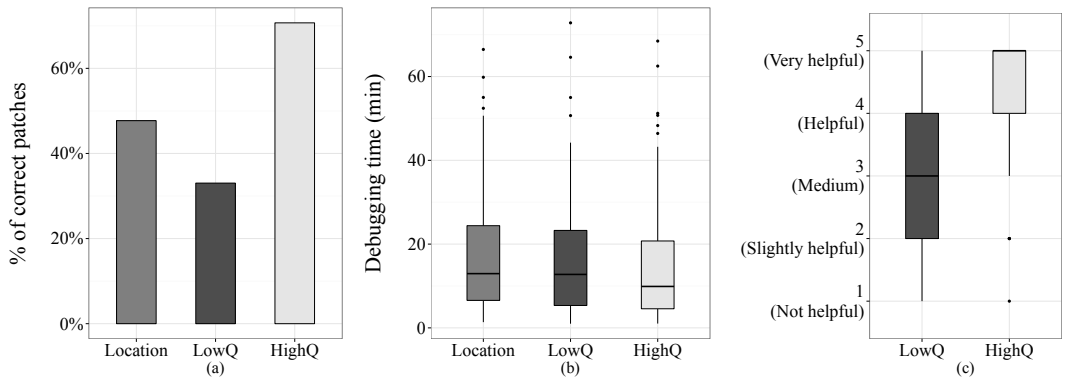


Figure 3: (a) the percentage of correct patches by different debugging aids; (b) debugging time by different aids; (c) participants' perception about the helpfulness of LowQ and HighQ patches.

3.3 Bug Difficulty

Since the five bugs vary in symptoms and defect types, they possibly pose different levels of challenges to the participants. Research also shows that task difficulty potentially correlates to users' perception about automated diagnostic aids [23]. For this reason, we investigate participants' debugging performance with respect to different bugs. We obtained the difficulty ratings of bugs from the exit survey. Note that we adopt the difficulty ratings from only the Location group since other participants' perceptions could have been influenced by the presence of generated patches. As shown in Figure 4, Rhino-192226 is considered to be the most difficult bug while Rhino-217379 is considered to be the easiest. In particular, ANOVA with Tukey HSD post-hoc test [20] shows that the difficulty rating of Rhino-192226[4] is significantly higher than that of the remaining four bugs.

Figure 5c shows that for bug3, the HighQ group made a "landslide victory" with 70% of the submitted patches being correct, while no one from the Location or the LowQ group was able to fix it correctly. One primary reason could be the complex nature of this bug, which is caused by a missing `if` construct (Table 2). This type of bug, also known as the *code omission* fault, can be hard to identify [10][14]. Fry and Weimer also empirically showed that the accuracy of human fault localization is relatively low for missing-conditional faults [14]. In addition, the fix location of this bug is not the statement that throws the exception but five lines above it. The code itself is also complex for involving lexical parsing and dynamic compilation. Hence, both the Location and LowQ groups stumbled on this bug. However, the HighQ group indeed benefited from the generated patch, as one participant explained:

> *"Without deep knowledge of the data structure representing the code being optimized, the problem would have taken quite a long time to determine by manually. The patch precisely points to the problem and provided a good hint, and my debugging time was cut down to about 20 minutes."*

For bug4, interestingly, the Location group performs better than the LowQ and even the HighQ group (Figure 5c). One possible explanation is that the code of this bug requires less project-specific knowledge (Table 2) and its fix by adding a null checker is a common pattern [17]. Therefore, even without the aid of generated patches, participants were likely to resolve this bug solely by themselves, and they indeed considered it to be the easiest bug (Figure 4). For bug5, the LowQ patch deletes the entire buggy block inside the for-loop (Table 2), which might raise suspicion for being too straightforward. This possibly explains why the LowQ and Location groups have the same correctness for this bug.

[4]For simplicity, in the remaining of the paper we refer to each bug using its order appearing in Table 1.

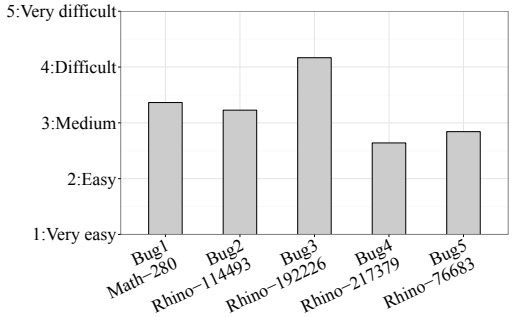


Figure 4: The average difficulty rating for each bug.

In general, the HighQ group has made the highest percentage of correct patches for all bugs except bug4. The LowQ group, on the other hand, has comparable or even lower correctness than the Location group. As for debugging time, participants in general are slower for the first three bugs as shown in Figure 5d. We speculate that this also relates to the difficulty of the bugs, as these three bugs are considered to be more difficult than the other two (Figure 4).

3.4 Programming Experience

Research has found that expert and novice developers are different in terms of debugging strategies and performance [18]. In this study, we also explore whether the usefulness of generated patches as debugging aids is affected by participants' programming experience.

Among all 95 participants, 72 reported their Java programming experience. They have up to 14 and on average 4.4 years of Java programming experience. Accordingly, we divided these 72 participants into two groups — *experts* with above average and *novices* with below average programming experience.

As shown in Figure 5e, experts are 60% correct when aided by Location. This number increases to 76% when they are aided by HighQ patches and decreases to 45% when they

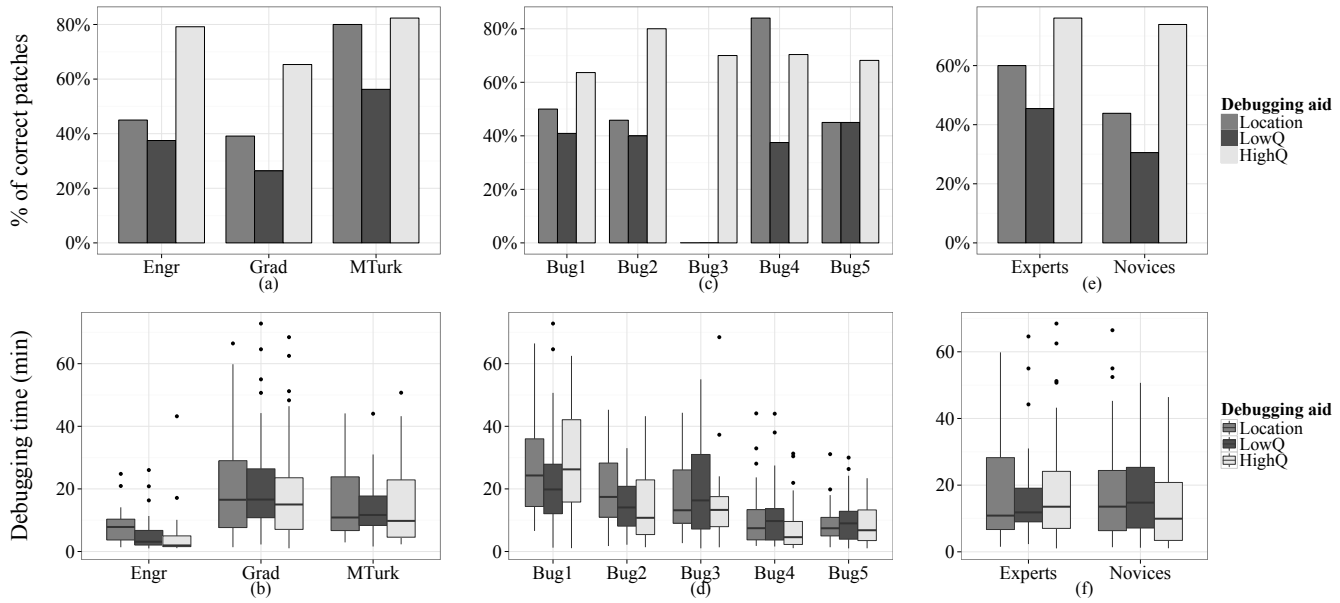


Figure 5: Percentage of correct patches and debugging time by participants, bugs, and programming experience.

are aided by LowQ patches. Novices manifest a similar yet sharper trend: their percentage of correct patches increases substantially from 44% to 74% when they are aided by Location and HighQ patches, respectively. Interestingly, when aided by HighQ patches, the debugging correctness of novices is almost as good as that of the experts. Figure 5f shows that novices also spend less time debugging with HighQ patches than with Location and LowQ patches.

3.5 Regression Analysis

Prior subsections observe that changes in debugging correctness and time are attributable to debugging aids, bugs, participant types, programming experience, and their joint effects. To quantify the effects of these factors on the debugging performance, we perform *multiple regression analysis*. Multiple regression analysis is a statistical process for quantifying relations between a dependent variable and a number of independent variables [12], which fits naturally to our problem. Multiple regression analysis also quantifies the statistical significance of the estimated relations [12]. This type of analysis has been applied in other similar research, for example, to analyze how configurational characteristics of software teams affect project productivity [4][5][31].

Debugging aids, participant types and bugs are all categorical variables, with 3, 3, 5 different values, respectively. Since categorical variables cannot be directly used in regression models, we apply *dummy coding* to transform a categorical variable with k values into $k-1$ binary variables [12]. For example, for the debugging aids, we create two binary variables *HighQ* and *LowQ*, one of them being 1 if the corresponding patch is given. When both variables are 0, the Location aid is given and serves as the basis to which the other two categories are compared. In the same manner, we dummy-code participant types and bugs with "Grad" and "bug4" as the basis categories[5]. Below shows the equations that form our regression models. Since patch correctness is binary data while debugging time is continuous data, we estimate equation 1 using *logistic regression* and equation 2 using *linear regression* [12]. Note that we log-transform debugging time to ensure its normality [12].

$$\begin{aligned}\text{logit(correctness)} = \alpha_0 + \alpha_1 * \text{HighQ} + \alpha_2 * \text{LowQ} \\ +\alpha_3 * \text{Engr} + \alpha_4 * \text{MTurk} + \alpha_5 * \text{Bug1} \\ +\alpha_6 * \text{Bug2} + \alpha_7 * \text{Bug3} \\ +\alpha_8 * \text{Bug5} + \alpha_9 * \text{JavaYears}\end{aligned} \quad (1)$$

$$\begin{aligned}\ln(\text{time}) = \beta_0 + \beta_1 * \text{HighQ} + \beta_2 * \text{LowQ} + \beta_3 * \text{Engr} \\ +\beta_4 * \text{MTurk} + \beta_5 * \text{Bug1} + \beta_6 * \text{Bug2} \\ +\beta_7 * \text{Bug3} + \beta_8 * \text{Bug5} + \beta_9 * \text{JavaYears}\end{aligned} \quad (2)$$

Table 4 shows the regression results, with the first number as the variable coefficient and the second number in the parenthesis as its p-value. In general, a positive coefficient means the variable (row name) improves the debugging performance (column name) compared to the basis category, while a negative coefficient means the opposite. Since the logistic regression computes the log of odds for correctness and we log-transform debugging time in the linear regression, both computed coefficients need to be "anti-logged" to interpret the results. We highlight our findings below.

HighQ patches significantly improve debugging correctness, with coef=1.25>0 and p-value=0.00<0.05. To put this into perspective, holding other variables fixed, the odds of the HighQ group making correct patches is $e^{1.25}$=3.5 times that of the Location group. The LowQ group, on the other hand, is less likely to make correct patches though this trend is not significant (coef= -0.55, p=0.09). The aid of HighQ (coef=-0.19, p=0.08) and LowQ patches (coef=-0.07,p=0.56) both slightly reduce the debugging time but not significantly.

Difficult bugs significantly slow down debugging, as bug1,2,3 all have significant positive coefficients or longer debugging time. For the most difficult bug3, its debugging

[5]The selection of basis category is essentially arbitrary and does not affect the regression results [12].

Table 4: Regression results. Arrows indicate the direction of impact on debugging performance. Double arrows indicate that the coefficient is statistically significant at 5% level.

	Correctness coef (p-value)	Debugging Time coef (p-value)
HighQ	1.25 (0.00): ⇑	-0.19 (0.08): ↓
LowQ	-0.55 (0.09): ↓	-0.07 (0.56): ↓
Bug1	-0.57 (0.18): ↓	0.96 (0.00): ⇑
Bug2	-0.50 (0.22): ↓	0.56 (0.00): ⇑
Bug3	-2.09 (0.00): ⇓	0.59 (0.00): ⇑
Bug5	-0.59 (0.16): ↓	-0.05 (0.75): ↓
Engr	1.03 (0.02): ⇑	-1.25 (0.00): ⇓
MTurk	0.90 (0.02) ⇑	-0.16 (0.20): ↓
Java years	0.12 (0.02): ⇑	-0.03 (0.11): ↓

time is $e^{0.59}$=1.8 times that of bug4, yet the odds of correctly fixing it is only $e^{-2.09}$≈1/8 of the odds of correctly fixing bug4.

Participant type and experience significantly affect debugging performance: compared to Grad, Engr and MTurk are much more likely to make correct patches (coef=1.03 and 0.90 with p=0.02). Engr is also significantly faster, spending approximately $e^{-1.25}$≈1/3 time debugging. Also, more Java programming experience significantly improves debugging correctness (coef=0.12, p=0.02).

Our regression analysis thus far investigates the individual impact of debugging aids and other variables on the debugging performance. However, the impact of debugging aids also possibly depends on the participants who use the aids or the bugs to be fixed. To explore such joint effects, we add *interaction variables* to the regression model and use ANOVA to analyze their explanatory powers [12]. Below are our findings.

HighQ patches are more useful for difficult bugs: as shown in Table 5, in addition to the four single variables, the debugging aids and bugs also jointly explain large variances in the patch correctness (deviance reduction=25.54, p=0.00). Hence, we add this interaction variable Aid:Bug to the logistic regression for estimating correctness. Table 6 shows that HighQ:Bug3 has a relatively large positive coefficient (coef=18.69), indicating that the HighQ patch is particularly useful for fixing this bug.

The type of aid does not affect debugging time: none of the interaction variables significantly contributes to the model (Table 5). Interestingly, neither does the debugging aid itself, which explains only 1% (R^2=0.01) of the variance in the debugging time. Instead, debugging time is much more susceptible to participant types and bugs, which further explain 20% and 16% of its variance, respectively.

These regression results are consistent with those observed from Figures 3, 4 and 5 in the preceding subsections. Due to space limitation, we elaborate more details of the regression analysis at http://www.cse.ust.hk/~idagoo/autofix/regression_analysis.html, which reports the descriptive statistics and correlation matrix for the independent variables, the hierarchical regression results, and how Table 4, 5, and 6 are derived.

3.6 Summary

Based on our results, we draw the following conclusions:

- High-quality generated patches significantly improve debugging correctness and are particularly beneficial for difficult bugs. Low-quality generated patches slightly undermine debugging correctness.
- Participants' debugging time is not affected by the debugging aids they use. However, their debugging takes a significantly longer time for difficult bugs.
- Participant type and experience also significantly affect debugging performance. Engr, MTurk, or the participants with more Java programming experience are more likely to make correct patches.

Table 5: ANOVA for the regression models with interaction variables. It shows how much variance in correctness/debugging time is explained as each (interaction) variable is added to the model. The p-value in parenthesis indicates whether adding the variable significantly improves the model.

Variable	Correctness (resid. deviance reductions)	Debugging Time R^2 (variance explained)
DebuggingAid	32.80 (**0.00**)	0.01 (0.06)
Participant	16.85 (**0.00**)	0.20 (**0.00**)
Bug	21.91 (**0.00**)	0.16 (**0.00**)
JavaYears	6.13 (**0.01**)	0.01 (0.11)
Aid:Participant	2.95 (0.57)	0.02 (0.06)
Aid:Bug	25.54 (**0.00**)	0.02 (0.26)
Aid:JavaYears	1.18 (0.55)	0.00 (0.82)

Table 6: Coefficients of the interaction variable DebugAid:Bug for estimating patch correctness. The bold value is statistically significant at 5% level.

	Bug1	Bug2	Bug3	Bug4	Bug5
HighQ	0.42	1.86	18.69	-0.45	0.99
LowQ	-0.39	-0.24	0.12	**-1.76**	-0.09

4. QUALITATIVE ANALYSIS

We used the survey feedback to qualitatively investigate participants' opinions on using auto-generated patches in debugging. In total, we received 71 textual answers. After an open coding phase [20], we identified twelve common reasons why participants are positive or negative about using auto-generated patches as debugging aids. Table 7 summarizes these reasons along with the participants' original answers.

Participants from both the HighQ and LowQ groups acknowledge generated patches to provide quick starting points by pinpointing the general buggy area (P1-P3). HighQ patches in particular simplify debugging and speed up the debugging process (P4-P6). However, a quick starting point does not guarantee that the debugging is going down the right path, since generated patches can be confusing and misleading (N1-N2). Even HighQ patches may not completely solve the problem and thus need further human perfection (N3).

Participants, regardless of using HighQ or LowQ patches, have a general concern that generated patches may overcomplicate debugging or over-simplify it by not addressing root causes (N4-N5). In other words, participants are concerned about whether machines are making random or educated guesses when generating patches (N6). Another shared concern is that the usage of generated patches should be based on a good understanding to the target programs (N7-N8).

Table 7: Participants' positive and negative opinions on using auto-generated patches as debugging aids. All of these sentences are the participants' original feedback. We organize them into different categories with the summary written in *italic*.

Positive	Negative
It provides a quick starting point.	***It can be confusing, misleading or incomplete.***
P1: It usually provides a good starting point. (HighQ) **P2**: It did point to the general area of code. (LowQ) **P3**: I have followed the patch to identify the exact place where the issue occurred. (LowQ)	**N1**: The generated patch was confusing. (LowQ) **N2**: For developers with less experience it can give them the wrong lead. (LowQ) **N3**: It's not a complete solution. The patch tested for two null references, however, one of the two variables was dereferenced about 1-2 lines above the faulting line. Therefore, there should have been two null reference tests. (HighQ)
It simplifies the problem.	***It may over-complicate the problem or over-simplify it by not addressing the root cause.***
P4: I feel that the hints were quite good and simplified the problem. (HighQ) **P5**: The hint provided the solution to the problem that would have taken quite a long time to determine by manually. (HighQ) **P6**: This saves time when it comes to bugs and memory monitoring. (HighQ)	**N4**: Sometimes they might over-complicate the problem or not address the root of the issue. (HighQ) **N5**: The patch did not appear to catch the issue and instead looked like it deleted the block of code. (LowQ) **N6**:Generated patches are only useful when the machine does not do it randomly, e.g., it cannot just guess that something may fix it, and if it does, that that is good. (LowQ)
It helps brainstorming.	***It may not be helpful for unfamiliar code.***
P7: They would seem to be useful in helping find various ideas around fixing the issue, even if the patch isn't always correct on its own. (LowQ)	**N7**: I prefer to fix the bug manually with a deeper understanding of the program. (HighQ) **N8**: A context to the problem may have helped here. (LowQ)
It recognizes easy problems.	***It may not recognize complicated problems.***
P8: They would be good at recognizing obvious problems (e.g., potential NPE). (HighQ)	**N9**: They would be difficult to recognize more involved defects. (HighQ)
It makes tests pass.	***It may not work if the test suite itself is insufficient.***
P9: The patch made the test case pass. (LowQ)	**N10**: Assuming the test cases are sufficient, the patch will work. (HighQ) **N11**: I'm worried that other test cases will fail. (LowQ)
It provides extra diagnostic information.	***It cannot replace standard debugging tools.***
P10: Any additional information regarding a bug is almost always helpful. (HighQ)	**N12**: I would use them as debugging aids, along with access to standard debugging tools. (LowQ)

Interestingly, we also observe several opposite opinions. Participants think that generated patches may be good at recognizing easy bugs (P8), but may not work well for complicated ones (N9). While generated patches can quickly pass test cases (P9), they work only if test cases are sufficient (N10-N11). Finally, participants use generated patches to brainstorm (P7) and acquire additional diagnostic information (P10). However, they still consider generated patches dispensable, especially when powerful debugging tools are available (N12).

In general, participants are positive about auto-generated patches as they provide quick hints about either buggy areas or even fix solutions. On the other hand, participants are less confident about the capability of generated patches for handling complicated bugs and addressing root causes, especially when they are unfamiliar with code and test cases.

5. THREATS TO VALIDITY

The five bugs and ten generated patches we used to construct debugging tasks may not be representative of all bugs and generated patches. We mitigate this threat by selecting real bugs with varied symptoms and defect types and using patches that were generated by two state-of-the-art program repair techniques [17][34]. Nonetheless, further studies with wider coverage of bugs and auto-generated patches are required to strengthen the generalizability of our findings.

We measured patch quality using human-perceived acceptability, which may not generalize to other measurements such as metric-based ones [27]. However, patch quality is a rather broad concept, and research has suggested that there is no easy way to definitively describe patch quality [13][24]. Another threat regarding patch quality is that we labeled a generated patch as "LowQ" relative to its competing "HighQ" patch in this study. We plan to work on a more general solution by establishing a quality baseline, for example the buggy location as in the control group, and distinguishing the quality of auto-generated patches according to this baseline.

To avoid *repeated testing* [20], we did not invite domain experts who were familiar with the buggy code, since it would be hard to judge whether their good performance is attributable to debugging aids or their a priori knowledge of the corresponding buggy code. Instead, we recruited participants who had little or no domain knowledge of the code. This is not an unrealistic setting since in practice, devel-

opers are often required to work with unfamiliar code due to limited human resources or deadline pressure [9][19][25]. However, our findings may not generalize to project developers who are sufficiently knowledgeable about the code. Future studies are required to explore how developers make use of auto-generated patches to debug their familiar code.

Our manual evaluation of the participants' patches might be biased. We mitigate this threat by considering patches verified by the original project developers as the ground truth. Three human evaluators individually evaluated the patches for two iterations and reached a consensus with the Fleiss kappa equals to 0.956 (Section 2.4).

Our measuring of debugging time might be inaccurate for Engr and MTurk participants, since the online system cannot remotely monitor participants when they were away from keyboard during debugging. To mitigate this threat, we asked these participants to self-report the time they had spent on each task and found no significant difference between their estimated time and our recorded time.

One may argue that participants might have blindly reused the auto-generated patches instead of truly fixing bugs on their own. We took several preventive measures to discourage such behaviors. First, we emphasized in the instructions that the provided patches may or may not really fix the bugs and participants should make their own judgement. Second, we claimed to preserve additional test cases by ourselves so that participants may not relax even if they passed all the tests by reusing the generated patches. We also required participants to justify their patches during submissions. As discussed in Section 3, the participants aided by generated patches spent a similar amount of debugging time to the Location group. This indicates that participants were less likely to reuse generated patches directly, which would otherwise take only seconds to complete.

6. RELATED WORK

Automatic patch generation techniques have been actively studied. Most studies evaluated the quantitative aspects of proposed techniques, such as the number of candidates generated before a valid patch is found and the number of subjects for which patches can be generated [8][21][26][30][33][34]. Our study complements prior studies by qualitatively evaluating the usefulness of auto-generated patches as debugging aids.

Several studies have discussed the quality of generated patches. Kim et al. evaluated generated patches using human acceptability [17], which is adopted in our study to indicate patch quality. Fry et al. considered patch quality as its understandability and maintainability [13]. Their human study revealed that generated patches alone are less maintainable than human-written patches. However, when augmented with synthesized documentation, generated patches have comparable or even better maintainability than human-written patches [13]. Fry et al. also investigated code features (e.g., number of conditionals) that correlate to patch maintainability [13].

While patch quality can be measured from various perspectives, research has suggested that instead of using a universal definition, the measurement of patch quality should depend on the application [13][24][27]. Monperrus suggested that generated patches should be understandable by humans when used in recommendation systems [24]. We used essentially the same experimental setting, in which participants were recommended but not forced to use generated patches for debugging. Our results consistently show that patch understandability affects the recommendation outcome.

Automated support for debugging is another line of related work. Parnin and Orso conducted a controlled experiment, requiring human subjects to debug with or without the support of an automatic fault localization tool [28]. Contrary to their finding that the tool did not help perform difficult tasks, we found that generated patches tend to be more beneficial for fixing difficult bugs.

Ceccato et al. conducted controlled experiments, in which humans performed debugging tasks using manually written or automatically generated test cases [7]. They discovered that despite the lack of readability, human debugging performance improved significantly with auto-generated test cases. However, generated test cases are essentially different from the generated patches used in our study. For instance, generated test cases include sequences of method calls that may expose buggy program behaviors, while generated patches directly suggest bug fixes. Such differences might cause profound deviations in human debugging behaviors. In fact, contrary to the findings of Ceccato et al., we observed that the participants' debugging performance dropped when given low-quality generated patches.

7. CONCLUSION

We conducted a large-scale human study to investigate the usefulness of automatically generated patches as debugging aids. We discover that the participants do fix bugs more quickly and correctly using high-quality generated patches. However, when aided by low-quality generated patches, the participants' debugging correctness is compromised.

Based on the study results, we have learned important aspects to consider when using auto-generated patches in debugging. First of all, strict quality control of generated patches is required even if they are used as debugging aids instead of direct integration into the production code. Otherwise, low-quality generated patches may negatively affect developers' debugging capabilities, which might in turn jeopardize the overall software quality. Second, to maximize the usefulness of generated patches, proper tasks should be selected. For instance, generated patches can come in handy when fixing difficult bugs. For easy bugs, however, simple information such as buggy locations might suffice. Also, users' development experience matters, as our result shows that novices with less programming experience tend to benefit more from using generated patches.

To corroborate these findings, we need to extend our study to cover more diverse bug types and generated patches. We also need to investigate the usefulness of generated patches in debugging with a wider population, such as domain experts who are sufficiently knowledgeable of the buggy code.

8. ACKNOWLEDGMENTS

We thank all the participants for their help with the human study. We thank Fuxiang Chen and Gyehyung Jeon for their help with the result analysis. This work was funded by Research Grants Council (General Research Fund 613911) of Hong Kong, National High-tech R&D 863 Program (2012AA011205) and National Natural Science Foundation (61100038, 91318301, 61321491, 61361120097) of China.

9. REFERENCES

[1] Amazon mechanical turk. https://www.mturk.com/mturk/.

[2] J. Anvik, L. Hiew, and G. C. Murphy. Coping with an open bug repository. In *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*.

[3] J. Aranda and G. Venolia. The secret life of bugs: Going past the errors and omissions in software repositories. ICSE'09.

[4] C. Bird, N. Nagappan, P. Devanbu, H. Gall, and B. Murphy. Does distributed development affect software quality? an empirical case study of windows vista. ICSE '09.

[5] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu. Don't touch my code!: Examining the effects of ownership on software quality. ESEC/FSE '11, 2011.

[6] A. Carzaniga, A. Gorla, A. Mattavelli, N. Perino, and M. Pezzè. Automatic recovery from runtime failures. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13.

[7] M. Ceccato, A. Marchetto, L. Mariani, C. Nguyen, and P. Tonella. An empirical study about the effectiveness of debugging when random test cases are used. In *ICSE'12*, pages 452–462.

[8] V. Dallmeier, A. Zeller, and B. Meyer. Generating fixes from object behavior anomalies. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE'09.

[9] E. Duala-Ekoko and M. P. Robillard. Asking and answering questions about unfamiliar APIs: an exploratory study. ICSE'12, pages 266–276.

[10] J. Duraes and H. Madeira. Emulation of software faults: A field data study and a practical approach. *Software Engineering, IEEE Transactions on*, 32(11):849–867, 2006.

[11] J. L. Fleiss. Measuring nominal scale agreement among many raters. *Psychological bulletin*, 76(5):378, 1971.

[12] J. Fox. *Applied regression analysis, linear models, and related methods*. Sage, 1997.

[13] Z. P. Fry, B. Landau, and W. Weimer. A human study of patch maintainability. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA'12.

[14] Z. P. Fry and W. Weimer. A human study of fault localization accuracy. ICSM'10.

[15] Z. Gu, E. T. Barr, D. J. Hamilton, and Z. Su. Has the bug really been fixed? ICSE'10.

[16] J.Nielsen. Time budgets for usability sessions. http://www.useit.com/alertbox/usability_sessions.html.

[17] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE'13.

[18] A. Ko and B. Uttl. Individual differences in program comprehension strategies in unfamiliar programming systems. In *IWPC'03*.

[19] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. *IEEE Transactions on Software Engineering*, 32:971–987, 2006.

[20] J. Lazar, J. H. Feng, and H. Hochheiser. *Research Methods in Human-Computer Interaction*. John Wiley Sons, 2010.

[21] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: fixing 55 out of 105 bugs for $8 each. ICSE'12.

[22] C. Le Goues, S. Forrest, and W. Weimer. Current challenges in automatic software repair. *Software Quality Journal*, pages 1–23, 2013.

[23] P. Madhavan, D. A. Wiegmann, and F. C. Lacson. Automation failures on tasks easily performed by operators undermine trust in automated aids. *Human Factors: The Journal of the Human Factors and Ergonomics Society*, 48(2):241–256, 2006.

[24] M. Monperrus. A critical review of Şautomatic patch generation learned from human-written patchesŤ: An essay on the problem statement and the evaluation of automatic software repair. ICSE'14.

[25] T. H. Ng, S. C. Cheung, W. K. Chan, and Y. T. Yu. Work experience versus refactoring to design patterns: a controlled experiment. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, SIGSOFT '06/FSE-14, pages 12–22, 2006.

[26] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. SemFix: program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE'13.

[27] K. Nishizono, S. Morisaki, R. Vivanco, and K. Matsumoto. Source code comprehension strategies and metrics to predict comprehension effort in software maintenance and evolution tasks - an empirical study with industry practitioners. In *ICSM'11*.

[28] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA'11.

[29] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard. Automatically patching errors in deployed software. SOSP'09.

[30] Y. Qi, X. Mao, Y. Lei, and C. Wang. Using automated program repair for evaluating the effectiveness of fault localization techniques. ISSTA '13, 2013.

[31] N. Ramasubbu, M. Cataldo, R. K. Balan, and J. D. Herbsleb. Configuring global software teams: A multi-company analysis of project productivity, quality, and profits. ICSE '11, pages 261–270, 2011.

[32] J. Ross, L. Irani, M. S. Silberman, A. Zaldivar, and B. Tomlinson. Who are the crowdworkers?: shifting demographics in mechanical turk. CHI EA '10.

[33] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. ISSTA '10.

[34] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. ICSE '09.