A Human Study of Patch Maintainability

Zachary P. Fry Bryan Landau Westley Weimer
University of Virginia
{zpf5a,bal2ag,weimer}@virginia.edu

Abstract

Identifying and fixing defects is a crucial and expensive part of the software lifecycle. Measuring the quality of bug-fixing patches is a difficult task that affects both functional correctness and the future maintainability of the code base. Recent research interest in automatic patch generation makes a systematic understanding of patch maintainability and understandability even more critical.

We present a human study involving over 150 participants, 32 real-world defects, and 40 distinct patches. In the study, humans perform tasks that demonstrate their understanding of the control flow, state, and maintainability aspects of code patches. As a baseline we use both human-written patches that were later reverted and also patches that have stood the test of time to ground our results. To address any potential lack of readability with machine-generated patches, we propose a system wherein such patches are augmented with synthesized, human-readable documentation that summarizes their effects and context. Our results show that machine-generated patches are slightly less maintainable than human-written ones, but that trend reverses when machine patches are augmented with our synthesized documentation. Finally, we examine the relationship between code features (such as the ratio of variable uses to assignments) with participants' abilities to complete the study tasks and thus explain a portion of the broad concept of patch quality.

Categories and Subject Descriptors D.2.3 [*Coding Tools and Techniques*]; D.2.7 [*Distribution, Maintenance, and Enhancement*]: Documentation; D.2.8 [*Software Engineering*]: Metrics

General Terms Human Factors

Keywords Human study, documentation synthesis, patch quality

1. Introduction

Defect repair is an important component of software maintenance, which dominates a system's life cycle [4, 31]. A defect report typically passes through a number of stages, including triage, assignment, and localization, before a developer produces a candidate *patch* to fix the defect in question [15]. Such patches are typically validated (e.g., through testing, inspections and walkthroughs, etc.) before being committed to the code base. Subsequent maintenance, including feature addition or additional bug fixing, requires that engineers are able to reason about and understand the patched code. Reading and understanding code is recognized as a difficult and time-consuming part of the maintenance cycle [27, 30].

The effect of a patch on later software evolution is of particular relevance given the flurry of recent research on automated program repair. Recent work has shown that patches can be automatically generated using evolutionary techniques [19, 20], dynamic program behavior modification [26], enforcement of explicit pre- and post-conditions [35], and program transformation guided by static analysis [16]. While these patches may be functionally correct, little effort has been taken to date to evaluate the understandability of the resulting code. Many developers express natural reluctance about incorporating machine-generated patches into their code bases [19].

Regardless of the provenance of a given patch, the quality of patched code critically affects software maintainability. There are a number of factors that contribute to patch quality. Perhaps most direct is a patch's impact on functional correctness: a high quality patch should bring an implementation more in line with its specification and requirements. Informally, a patch should fix the bug while retaining all other required functionality. However, even functionally correct patches can vary considerably in quality. Consider two patches that fix the same defect. One patch touches many lines, introduces several gotos and unstructured control flow, and contains no comments; the other is short and succinctly commented. Even if both have the same practical effect on program semantics, the first produces code that is likely more difficult to read and understand in the future as compared to the second. In this paper, we focus on patch quality as it relates to code understandability and, more broadly, software maintainability.

Software maintainability is a broad concept; it has been measured from many angles, using many different metrics [14, 38]. Despite ample effort in the area of software quality metrics, it has been noted that there is no adequate *a priori* descriptive metric for maintainability [18, 23, 28]. Sillito *et al.* study and categorize the questions developers actually ask about a code base while performing maintenance tasks [32]. For example, when looking to modify a piece of code, programmers often ask "What is the control flow that leads to this program point?", or "What variables are in scope at this point?" We define maintainability by the ease and accuracy with which these formalized maintenance questions can be answered about a given piece of code. If developers answer such questions less accurately or less rapidly, we say the associated maintainability has decreased. We can ground such a metric by taking advantage of a "natural experiment": many human-written patches are later *reverted* and undone (e.g., [39]), representing an explicit loss of maintenance effort. By examining both a reverted patch and a patch that stood the test of time — for the same bug — we perform a controlled experiment and obtain a lens through which to study patch quality.

We use this framework to quantify patch quality as it relates to maintainability, and to study the relative quality of automatically-generated patches as compared to human patches. To date, we are unaware of any human studies on the maintainability of patches in particular, or any studies that examine the differences between automatically-generated patches and those created by humans. We hypothesize that participants will neither perceive nor expose a difference in maintainability between human-created and machine-generated patches. We additionally propose to augment machine-generated patches with synthesized, human-readable documentation that describes the effect and context of the change. We further hypothesize that adding this supplemental documentation to machine-generated patches increases maintainability on average.

To test these hypotheses, we conduct a human study in which participants are presented with code and asked questions using Sillito *et al.*'s forms. We measure both accuracy and effort as proxies for maintainability; more maintainable code should admit more correct answers in less time. We show that human patches that were later reverted are generally less maintainable than those that were not reverted to establish that the metrics are well-grounded. To test the stated hypotheses, we measure the net changes in both accuracy and effort between the original faulty code and both human-created and machine-generated patches, holding all other factors constant. After establishing the relative maintainability effect of different types of patches, we examine which characteristics of the code relate to maintainability directly, and compare them with participants' opinions of what they thought affected patch quality.

We find that documentation-augmented machine-generated patches are of equal or greater maintainability than human-created patches. This work thus supports the long-term viability and cost-effectiveness of automatic defect repair. Additionally, we identify several code features that correlate with maintainability, which can support better patch generation — both manual and automatic — in the future. The main contributions of this paper are:

- A novel technique for augmenting machine-generated patches with automatically synthesized documentation. We focus on documenting both the context and the effect of a change, with the goal of increased maintainability.
- A human study of patch quality in which over 150 participants are shown patches to historical defects from large programs and asked questions indicative of those posed during real-world maintenance tasks.
- Statistically significant evidence that machine-generated patches, when augmented with synthesized documentation, produce code that can be maintained with equal accuracy and less effort than the code produced by human-written patches. These results provide preliminary evidence that automatically-generated patches may viably reduce long-term maintenance costs.
- A quantitative explanation of differences in patch maintainability in terms of measured code features. We contrast features that are predictive of actual human performance with features participants report as relevant.

2. Motivating Example

This section uses real-world bug fixes as examples to show that the effects of code patches on maintainability merit further study.

There are typically an infinite number of implementations adhering to any consistent specification. As such, there are typically a corresponding infinite number of functionally-correct patches for a given defect. For example, different patches may use different algorithms or data structures, reorder statements, include or remove dead code, or feature different commenting or indenting. Functionally equivalent patches may therefore have different effects on the code's readability or maintainability.

We present two distinct patches for a bug in the `php` scripting language interpreter to illustrate this point concretely. The

```
 1     if (offset >= s1_len) {
 2       php_error_docref(NULL TSRMLS_CC,
 3           E_WARNING, "The start position
 4           cannot exceed string length");
 5       RETURN_FALSE;
 6     }
 7
 8   - if (len > s1_len - offset) {
 9   -   len = s1_len - offset;
10   - }
11
12     cmp_len = (uint) (len ? len :
13         MAX(s2_len, (s1_len - offset)));
```

Figure 1. Patch #1 for `php` bug #54454 and surrounding code context. The patch modifies the `substr_compare` function, removing lines 8, 9, and 10.

`substr_compare` function compares two string parameters, `main_str` and `str`, for equality. The length of the comparison is controlled by a variable `len` — strings are trimmed to `len` characters before being compared. Bug report #54454 describes a defect in `substr_compare` where "if `main_str` is shorter than `str`, `substr_compare` [mistakenly] checks only up to the length of `main_str`." That is, if `main_str="abc"` and `str="abcde"`, `substr_compare` would erroneously return "true".[1] Informally, the bug is that `len` is set too low: checking only up to `len=3` does not reveal the differences between `"abc"` and `"abcde"`.

```
1   + len--;      /* Do set len = len - 1 */
2     if (mode & 2) {
3       for (i = len - 1; i >= 0; i--) {
4   -     if (mask[(unsigned char)c[i]]) {
5   -       len--;
6   -     } else
7           break;
8       }
9     }
```

Figure 2. Patch #2 for php bug #54454 and surrounding code context. The patch modifies function `php_trim`, removing lines 4, 5, and 6 while adding line 1.

Figure 1 shows one candidate patch that changes the `substr_compare` function directly. Lines added as part of the patch are preceded by a + while removed lines are denoted by a -. This patch removes the conditional on lines 8–10, which allowed `len` to be set too low; any extra letters are thus accounted for, and the bug in question is fixed.

By contrast, the patch in Figure 2 alters code in a local helper function related to string trimming. This patch causes `len` to always be decremented once before the loop, instead of once for every iteration of the loop in which a valid letter was found (since the strings are null-terminated, the single decrement is always allowed). `len` is thus, again, left sufficiently high to enable appropriate string comparison.

In terms of functional correctness, the two patches are equivalent: both produce code that passes the test case associated with the bug as well as the other 8,471 regression tests for the `php` interpreter. However, the resulting code may not be equally easy to reason about or maintain. What are the meaningful differences between the two patches, and how do these differences affect future maintenance tasks?

First, the patch contexts differ. The first patch applies directly to the 38-line `substr_compare` function; the second to a local helper function of comparable size (39 lines). In practice, this distinction

[1] `https://bugs.php.net/bug.php?id=54454` as of Feb 2012

means that the first patch changes `len` closer to its definition; the second changes it closer to its use. Both the size and granularity level of the enclosing code may contribute to maintainability. For instance, developers may find shorter functions easier to reason about and thus have no preference between the two patches based on function length. However, they may struggle with low-level, detail-oriented code and thus find the first patch more ultimately maintainable.

With respect to language constructs, the first patch strictly removes control flow and an assignment statement, while the second patch moves a statement outside of two conditionals and a loop, altering rather than removing control flow. While removing control flow often makes reading and understanding code easier, the effect is not universal (e.g., a loop with a constant number of iterations may be easier to grasp than its unrolling, even though the unrolling has fewer tests and branches).

Additionally, the second patch includes a very simple comment describing the effect of the change, which is typically viewed as a benefit [10, 24], but it also leaves in dead code (the loop on lines 3–9 now has no effect), which may confuse later maintainers.

Finally, the patches have different origins. The patch in Figure 1 was created by a developer and has remained untouched since April 3rd, 2011 (we thus deem it "accepted"). The patch in Figure 2 was evolved by the GenProg tool [19, 20] and augmented with machine-generated documentation.

These two patches were both subjects in our human study of patch quality, and clearly differ in several potentially important ways. However, it is not immediately clear how these differences affect maintainability. Perhaps surprisingly, in our study, participants were 0.25% *less* accurate when reasoning about the human-written patch (Figure 1) than about the original, while participants were 6.05% *more* accurate when reasoning about the machine-generated patch with documentation (Figure 2) than about the original (questions asked were common to both patches).

This example demonstrates that multiple patches fixing the same defect can be functionally correct, but result in differently maintainable code. We desire a more formal description of the relationship between various features (e.g., comments, patch context, control flow, etc.) and maintainability. We thus detail our human study and the resulting data in Section 3 and Section 4, designed to directly measure one notion of patch quality.

3. Approach

In this section we describe our proposal to augment machine-generated patches with synthesized documentation, as well as our human study to measure aspects of patch maintainability.

3.1 Synthesizing Documentation for Patches

A number of approaches to automated program repair operate at the source level and produce human-readable patches for defects [16, 19, 20, 35]. Automated program repair approaches hold out the promise of reducing some software maintenance costs, freeing up developers to focus on more important bugs, or allowing developers to address more issues in the same amount of time, since adapting a candidate patch takes less effort than constructing one from nothing [36]. Since bugs are reported faster than they can be addressed [3, p. 363] and commercial developers often take an average of 28 days to address even security-critical repairs [34], automated techniques that take mere hours represent a tempting economic option. However, if machine-generated patches are of poor quality and are harder to maintain than human-generated patches, their economic advantage disappears.

Automated repair techniques typically validate candidate patches against test suites (e.g., [19, 20]), implicit specifications (e.g., [16]) or explicit specifications (e.g,. [35]). The quality of such patches with respect to *functional correctness only* has been evaluated elsewhere and found to be human-competitive (e.g., against large held out test suites [20] or even against DARPA Red Teams [26]). Recall that human developers are not perfect. For example, a recent study of twelve years of patches to multiple free and commercial operating systems found that 15%–24% of human-written fixes for post-release bugs were "incorrect and have made impacts to end users." [39] As seen in Section 2, however, equally-correct patches may be more or less maintainable. In this paper we do not further address the issue of functional correctness and instead restrict attention to aspects of maintainability. All patches we consider, whether human-written or machine-generated, pass all available test cases.

We hypothesize that the maintainability of machine-generated patches can be improved by augmenting them with synthesized, human-readable documentation explicating their effects and contexts. Based on Sillito *et al.*'s set of maintenance questions, we identify state and control flow as critical for many types of maintenance. We hypothesize that developers will find maintenance easier if they understand how a patch changes program state (e.g., alters the values of variables) or alters program control flow (e.g., the conditions under which statements may be executed) at run-time. To that end we desire human-readable documentation that summarizes what a patch does, as opposed to why it was made.

We adapt the DeltaDoc algorithm of Buse *et al.* [6] which synthesizes human-readable version control commit messages for object-oriented programs. Their algorithm is based on a combination of symbolic execution and code summarization. In essence, each symbolically-executed statement is associated with its corresponding path predicate, and differences between the statements and predicates before and after applying a patch are summarized into human-readable documentation. Typical output is of the form "When calling `A()`, If `X`, do `Y` Instead of `Z`", where `X` is a path predicate and `Y` and `Z` are symbolic statement effects. We do not modify this basic output format, but instead widen the scope of program statements for which DeltaDoc generates documentation. The existing approach performs several optimizations with the goal of limiting the size of the output documentation. We make several changes to the technique as published to favor completeness over concision. The following changes were made:

- We alter the algorithm to report changes to all program statements, regardless of the length of the output to favor comprehensive understandability in lieu of brevity. Automatically-generated patches are often short [37]; we claim it is more important in this context to capture and describe all details. In particular, we remove all Summarizing Transformations in DeltaDoc's *Statement Filters* category [6, Sec. 4.3.1]. As a result, documentation is generated for statements, such as assignments to local variables, that are neither method invocations nor field accesses nor `return` or `throw` statements.
- We do not use single-predicate transformations that result in loss of information due to duplication or suspected lack of relatedness to the changed statements. For example, we output "If `a=5` and `b=true`, return `a`" instead of "If `a=5`, return `a`". Formally, this is a removal of DeltaDoc's third Summarizing Transformation in the *Single Predicate* category: "drop conditions that do not have at least one operand in documented statements" [6, Sec. 4.3.2].
- We do not "simplify" output by removing elements such as function call arguments. For example, we output "Always call `str_compare(main_str, str)`." instead of "Always call `str_compare()`." Formally, this corresponds to removing the *Simplification* category of DeltaDoc's Summarizing Transformations [6, Sec. 4.3.4].

- We avoid high-level simplification contingent on the length of the output, to favor a complete explanation over a concise one. The stated motivation of such simplification was that "this information often is sufficient to convey which components of the system were affected by the change when it would be impractical to describe the change precisely" [6, Sec. 4.3.4] — for the purposes of software maintenance we attempt to describe such changes precisely, even at the cost of verbosity.

We do not claim any novel results in the domain of documentation synthesis algorithms. Instead, we focus on the novel application of documentation synthesis to the problem of patch maintainability, and particularly to improve the quality of machine-generated patches.

3.2 Human Study Protocol

Our goal is to measure the maintainability of patched code and understand why some types of patches may be more or less maintainable than others. At a high level, we present human participants with segments of patched code and asking them maintenance questions about those snippets. We measure participant accuracy and effort in answering those questions. The remainder of this subsection formalizes our human study protocol, the procedure for selecting and presenting patches, the formulation and selection of questions, and finally participant selection.

Maintainability is difficult to evaluate *a priori* [18, 23, 28]. In this paper, we avoid predicting maintainability based on indirect correlations with auxiliary code features (cf. [38]) and strive instead to measure it directly. We propose a study to measure both objective measurements and subjective notions related to patch quality. Our general approach was to measure human *effort* and *accuracy* when performing various maintenance-related tasks (i.e., answer questions such as those proposed by Sillito *et al.* [32]). We also measure subjective judgments such as participant evaluations of quality and confidence. Whenever possible, we control for accuracy (e.g., by giving participants unlimited time and/or restricting attention to equally-accurate answers). If participants are equally accurate when reasoning about Patch X and Patch Y (typically two functionally correct patches that both address the same defect), but reasoning about Patch X takes twice as long, then Patch X imposes a higher maintenance burden (i.e., is less easily maintainable).

In our IRB-approved human study protocol, participants were initially presented with a detailed list of instructions and a tutorial detailing the required format for all answers in addition to example questions and answers. This training helps ensure that delays or mistakes can be attributed to the patches and not to initial confusion or training effects (we address such threats explicitly in Section 5). Participants were instructed not to attempt to run the code or use any external resources during the study. The heart of the study consisted of sequentially presenting each participant with 23 partial C code files (sampled from among a total set of 114 files), each with a length of 50 lines. The number 23 was chosen based on initial timing estimates to keep the total task duration manageable. Each code segment had a corresponding code understanding question that focused the user on a single line of code. Participants were asked to complete three tasks for each code segment:

- Answer the code understanding question (in free form text).
- Give a subjective judgment of how confident they were in their answer (using a 1–5 Likert scale).
- Give a subjective judgment of how maintainable they felt the code in question was (using a 1–5 Likert scale). Note that "maintainability" was not defined for participants; they were forced to use their own intuitions.

We recorded participants' accuracy when answering questions as well as the time it took them to reach an answer. As accuracy and effort represent the two major costs of the software maintenance cycle, together they can serve as measurable proxies for some aspects of "maintainability".

Finally, participants were presented with an exit survey containing questions about their computer science experience and personal opinions on the concept of maintainability.

3.3 Code Selection

To allow for a direct, controlled comparison between human-written patches and machine-generates patches, we used the benchmark suite presented by Le Goues *et al.* [19, Tab. 1]. The subject programs used thus come from several large open-source projects under ongoing development that include over 4 million lines of code and 9,000 test cases. Individual statistics for each program are presented in Table 1. We randomly selected 32 defects for which both human-written and machine-generated patches were available. Each defect had a priority/severity rating (where available) of at least three out of five, was sufficiently important for developers to fix manually, and was important enough to merit a checked-in test case. In addition, for each such defect we obtained the original code (i.e., the code for the first version just before the bug appeared) and, if possible, any human-written patches that had previously attempted to fix that bug but were reverted.

There are thus five distinct types of code collected and considered in this study:

- **Original** — defective, un-patched code used as a baseline for measuring relative changes in maintainability.
- **Human-Reverted** — human-created patches that were later reverted during the normal course of software maintenance.
- **Human-Accepted** — human patches that have not been reverted to date (at least six months).
- **Machine** — minimized, machine-generated patches produced by the GenProg tool, taken directly from the dataset of Le Goues *et al.* [19].
- **Machine+Doc** — machine-generated patches as above, but augmented with synthesized, human-readable documentation describing the effect and context of the change (see Section 3.1).

For patches affecting multiple changes, we centered the 50-line context window around the change affecting the largest number of lines, breaking ties randomly. We explicitly include both undocumented and automatically documented machine generated patches to test the effects of synthesized documentation on automatically generated patches. However, we specifically do not test the effect of synthesized documentation on human generated patches, as the goal of this work is to compare fully automatic approaches with a completely manual one. For both Human-Reverted and Human-Accepted patches we add any relevant software versioning commit messages as comments so as to use all available human-created information associated with a given patch.

The types listed have natural overlap with respect to an individual bugs. For instance, a given defect might have corresponding code for the Original, Human-Accepted, Machine, and Machine+Doc categories. Participants were shown a randomly chosen sequence of code types. We ensured that no two code segments from a single bug would ever be presented to a single user to avoid any training bias for the code and bug in question.

Because the Machine patches were created using a C front end [22], they may not correspond exactly to the original code (e.g., they may have different indentation). We manually removed any non-original, unnecessary artifacts left by the tool. All patches

Program	LOC	Tests	Defects	Human-Accepted Patches	Human-Reverted Patches	Machine- Generated Patches	Description
`gzip`	491,083	12	1	1	0	1	Compression utility
`libtiff`	77,258	78	7	7	0	7	Image processing utility
`lighttpd`	61,528	21	3	1	2	1	Webserver
`php`	1,046,421	8471	9	8	1	8	Language interpreter
`python`	407,917	355	1	1	0	1	Language interpreter
`wireshark`	2,812,340	63	11	0	11	0	Packet analyzer
total	4,896,547	9,000	32	18	14	18	

Table 1. A list of the subject programs we used as sources for patches in our human study, including the number of each type of patch used for each code base.

presented to users were functionally identical to those produced by the GenProg tool, while syntactically matching the original code as closely as possible. No changes were made that could not have been applied mechanically.

As mentioned in Section 3.1, we hypothesize that the maintainability of machine-generated patches will be improved by the addition of documentation summarizing effects and contexts. When presenting Machine+Doc code to participants, we inline the descriptive comments on (space permitting) or directly above the first line in the patch. Similarly, for human-written code, we inline the associated version control log message, if any. Thus, in terms of functional correctness and program semantics the Machine and Machine+Doc patches are identical: the only difference is the addition of documentation in the latter.

3.4 Code Understanding Question Selection and Formulation

To measure maintainability, we require subject questions that are indicative of those developers would ask during the maintenance process. Sillito *et al.* identify 44 different types of questions they directly observed real developers asking when performing maintenance tasks [32]. We used the five of these generic question types that focused on line-level granularity, as this was the most appropriate for 50-line code segments we presented to participants. Examples of the question types we selected are as follows:

- What conditions must hold to always reach line X during normal execution?
- What is the value of the variable "`y`" on line X?
- What conditions must be true for the function "`Z()`" to always be called on line X?
- At line X, which variables must be in scope?
- Given the following values for relevant variables, what lines are executed by beginning at line X? `Y=5 && Z=true`.

Many of the questions observed by Sillito *et al.* were more general in nature and would not have been applicable for gauging humans' understanding of the code segments used in the study. An example of a question Sillito *et al.* observed that did not apply is "Does this type have any siblings in the type hierarchy?" [32, Question #9]. In the majority of cases, the code segments shown to participants did not represent an entire class and, as such, there generally would not have been enough context to answer such a question. Question types were randomly selected for each code segment collected. If a question type did not apply to the code in question (e.g. a question about function calls when none appeared in the code), a new question type was randomly selected until a viable option was found. We call the line X in the examples above the *focus line*.

As all questions operated at a line-level granularity, we applied a deterministic algorithm for choosing the focus line. Our goal when selecting focus lines is to direct participant attention to the changes made by the patch; directing them to unchanged statements across different versions of similar code would fail to measure changes in maintainability caused by the patches. The main stipulation for choosing a line on which to focus was that it must occur in all relevant code segments (i.e., it must be associated with all available patches for that bug), to allow for controlled experimentation. For example, if the human-created and machine-generated patch for a given bug share the same context, then only lines that occur in both patched versions of the code as well as the original source are valid choices. We adopted the following process for choosing the focus for a given patch's relevant code segments:

1. Let S be the intersection of all lines for all relevant code segments. If $S = \emptyset$, discard the bug and restart, otherwise proceed to step 2.
2. Let T be the subset lines in S that are *dominated* [2] by any part of the largest change from each of the relevant patches. If $T = \emptyset$, repeat with next largest change from the patch in conflict. If no further changes exist, discard the bug and restart, otherwise proceed to step 3.
3. Choose the line in T that is closest to a line changed by the patch (as reported by `diff`) in question.

Once both a focus line and a question have been selected for a given code segment, the remaining portions of the question were selected uniformly at random, but with the stipulation that the code changed for the patch be highlighted whenever possible. For instance, if the question was "what is the value of variable `Y` on line X?", variable `Y` was chosen randomly out of all the variables with values that were affected in the relevant patches. If no such variable existed or was in scope at line X, a variable was selected at random from those in scope. In the above example, the typical outcome of this process was that X referred to a line dominated by code changed by all of the patches (thus, if control flow were to reach line X, it must first have passed through patched code) and the variable `Y` was one with a value altered by code changed by the patch.

Using this set of guidelines, we selected and crafted maintainability questions for a total of 114 code segments including 40 semantically distinct patches from the 32 unique bugs. When asked if they thought the study questions mimicked the actual maintenance process, the majority of participants responded affirmatively on a Likert scale (1–5).

3.5 Participant Selection

We aim to measure the maintainability of real source code. We thus require study participants with skills at least on par with novice developers, who might perform maintenance tasks on the target systems in the real world. We solicited responses from three groups of people and imposed accuracy standards to ensure that our results are more likely to generalize. All participants were required to have at least some programming experience in the C language. While participants were asked to self-report their experience, we used only objective accuracy measurements as cutoffs.

Participants fell into three categories: 27 fourth-year undergraduate students, 14 graduate students, and 116 Internet users participating via Amazon.com's Mechanical Turk "crowdsourcing" website. A fourth year computer science undergraduate student is indicative of someone who will soon enter the target industry as a novice. This is the lowest acceptable level of experience for our study and would represent "new hires" or developers who may not be familiar with the project — an especially important demographic to consider when studying the future maintainability of a system. Graduate students have generally had more experience and are more akin to a somewhat experienced developer from another project. Finally, Mechanical Turk participants varied widely in both industrial and academic experience. Participants were kept anonymous and were offered a chance to win one of two $50 Amazon gift cards and either class extra credit (via a randomized completion code, for students) or $4.00 (for Mechanical Turk participants).

In all cases, we impose an accuracy cutoff (described below) to ensure the quality and generalizability of the overall participant population. Amazon.com's Mechanical Turk crowdsourcing service deserves a detailed mention; it is effective as a means of obtaining a diverse population, but requires special consideration to ensure the overall quality of the data set. Previous work has shown that the Mechanical Turk participants can be effective when large populations are required [17, 33] — including for software engineering studies [11]. However, when offered a reward for an anonymous service, people may attempt to receive compensation without giving their best effort. We therefore set two criteria for *all* participants. First, participants were required to give answers for all questions and complete the exit survey fully. Second, participants who scored more than one standard deviation below the average student's score were removed from consideration. This accuracy cutoff was imposed because we cannot directly control for experience levels (especially with Mechanical Turk participants) and desire a level of competency consistent with someone who has completed at least some portion of an undergraduate education. Participants were aware of accuracy requirement and that their reward depended on it; since there was no time limit, participants were thus encouraged to take as long as necessary to get the correct answer, rather than to rush through the questions. In practice participants did just that (see details in Section 4), which often allowed us to measurably hold accuracy constant and thus use time taken as a key proxy for maintenance difficulty.

4. Experiments

This section presents statistical analyses of the results of the human study to address the following research questions:

1. How do different types of patches affect maintainability?
2. Which source code characteristics are predictive of our maintainability measurements?
3. Do participants' intuitions about maintainability and its causes agree with measured maintainability?

Recall that we are focusing only on particular aspects of maintainability (i.e., accuracy and effort when answering types of questions observed in real-world maintenance [32]).

As mentioned in Section 3.5, we impose an experimental cutoff to ensure only participants representative of the target "new developer" population are included in our analyses. Out of the 41 students surveyed, the mean accuracy was 53.8%. We thus establish a cutoff one standard deviation below that mean, at 34.7% accuracy. Imposing this cutoff restricted the overall subject pool from 157 to a total of 102 participants and over 2,100 individual data points.[2]

[2] Collected data available at `http://genprog.cs.virginia.edu/`

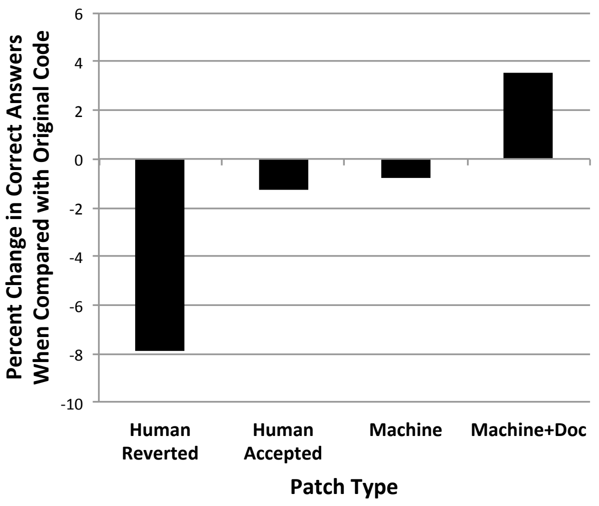

Figure 3. Percent change in *accuracy* of participants' answers as a function of the type of patch. The percent change is measured against the original, buggy code (i.e., before the patch was applied). Of the four types of patches we investigated, the sole type that, when applied, increases the maintainability of the original code on average is Machine+Doc. However, the means of the percent changes presented are not different in a statistically significant manner.

4.1 How do patch types affect maintainability?

We use two metrics to measure patch maintainability, corresponding to major costs throughout the maintenance process: accuracy and effort. *Accuracy* is calculated by manually verifying all collected responses and measuring the percentage of correct answers for all questions of a given patch type. At least two annotators verified each participant's answers to mitigate grading errors or ambiguities due to the use of free-form text. *Effort* is a calculated by averaging the number of minutes it took participants to submit a *correct* answer for all questions related to a given patch type. We omit incorrect answers as part of this statistic because incorrect answers often required only a few seconds if a participant decided to skip a question or simply guess.

Figure 3 shows the average percent change in accuracy for a given patch type. We measure the change in accuracy for a patch type t as follows:

$$\textsf{change\_in\_accuracy}(t) = \frac{\sum_{i \in t} acc(patch_i) - acc(orig_i)}{|t|}$$

where i ranges over all patches of type t, $patch_i$ is a particular patch, and $orig_i$ is the corresponding original code. The acc function calculates the average accuracy for maintenance questions about a particular piece of code over all participants.

If all other variables are held equal (i.e., the code it modifies, any documentation it may include, etc.), any change in accuracy is explained only by the patch. A one-sided Student's t-test shows that none of the means of the percent changes presented can be considered different in a statistically significant manner ($p = 0.066 \not\leq 0.05$). Therefore, we cannot conclude that humans are more or less capable of correctly answering questions typical of those that arise during the maintenance process when examining different types of patches. However, we can conclude that accuracy on machine-generated patches is *not worse* than accuracy on human-written patches (i.e., mean accuracy for machine-generated

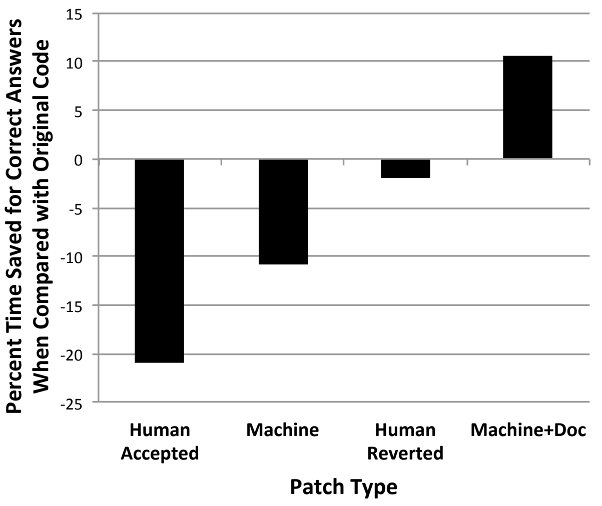

Figure 4. Percent *effort* saved for correct answers as a function of patch type. The percent change is measured against the original, buggy code (i.e., before the patch was applied). Of the four types of patches we investiaged, the only type that, on average, saved humans effort was Machine+Doc. With statistical significance, the mean effort saved when applying Machine+Doc patches is strictly greater than the effort saved when applying Human Accepted patches. In fact, we find that Human Accepted patches actually cause an increase in effort over the original code.

patches is either greater than or equal to that for human-written patches).

Figure 4 shows the percentage of time saved for each patch type when compared with the corresponding original code. The time saved is calculated as follows:

$$\mathsf{time\_saved}(t) = \frac{\sum_{i \in t} time(orig_i) - time(patch_i)}{|t|}$$

where i ranges over patches of type t, $patch_i$ is a particular patch shown to participants, $orig_i$ is the corresponding original code, and the $time$ function returns the average time taken by all participants who answered the question correctly. In this measurement, the mean time required to correctly answer questions of Machine+Doc patches is lower than the mean time required to answer Human-Accepted patches in a statistically significant manner ($p < 0.048$). Specifically, Human-Accepted patches resulted in a 20.9% increase in time-to-correct-answer, compared to the original code. However, Machine+Doc patches actually reduced the average time-to-correct-answer by 10.6%.

As mentioned in Section 3.3, we do not explicitly measure the effect of machine-generated documentation on human patches. In Section 1, we hypothesized that we could create and present machine-generated patches so that they would be at least as maintainable as comparable human-generated patches. As such, investigating the independent effect of synthesized documentation is outside of the scope of this work. We instead aim to provide evidence that fully machine-generated patches may be viable compared to human-generated patches.

To summarize, participants are *at least as accurate* when answering maintenance questions about machine-generated patches augmented with documentation then when they are answering questions about human-written patches. In addition, when accuracy is held constant, participants *take less time* to correctly answer maintenance questions about machine-generated patches augmented with documentation than they do to correctly answer questions about human-written patches (by 31.5%, $p < 0.048$). Perhaps surprisingly, for this aspect of maintainability (i.e., quickly and correctly answering questions that are indicative of the real-world maintenance process), Machine+Doc patches are more maintainable than Human-Accepted patches based on the results of this study: they admit at least as much accuracy and require less maintenance time on average with statistical significance.

4.2 Which code features predict maintainability?

We have established that patches from different sources have different maintainability, as measured by the accuracy obtained and effort required when participants answer indicative software maintenance questions about them. In this section we investigate these changes in maintainability in terms of code features and quantitatively analyze those features in terms of their predictive power.

We tested several types of classifiers (e.g. naïve Bayesian, Bayesian network, multi-layer perceptron, decision tree, etc.) and chose a Logistic regression based on its simplicity and ability to correctly classify patches with respect to human accuracy (Bayesian and perceptron accuracies were within 9% of the chosen classifier). Given the feature values associated with a patch, the logistic model was able to correctly classify whether the participant would answer the maintenance question correctly for 73.16% of the 2109 human judgments. We find this model accurate enough to be useful when investigating the predictive power of individual features. We used the ReliefF technique to measure a given feature's relative predictive power, which computes this statistic without assuming conditional independence [29]. A principle component analysis shows that 17 code features are needed to account for 90% of the variance in the data. This high number of features highlights the complex nature of maintainability: it is not easily explained away by a small combination of notions.

Table 2 ranks the top 17 features by their ReliefF predictive power with respect to the human accuracy data collected from our study. The results show that a mix of both syntactical and semantic features help to predict maintainability. Of note is that "number of comments" is not a particularly predictive feature, while Figure 3 and Figure 4 show a clear difference in human performance between Machine and Machine+Doc — patch types in which the only difference is the presence of summarizing documentation. Heitlager *et al.* echo this intuition, finding that the majority of comments are "simply code that has been commented out" [14]. Since the mere *number* of comments present (i.e., one more in Machine+Doc) is not sufficient to explain the difference in human performance, we conclude that the *content* of the comments is critical.[3] That is, Machine and Machine+Doc can be viewed as a controlled experiment in which only the presence of the documenting comment changes, and thus only a few features change (e.g., number of comments and total number of characters). Those changed features are not sufficient to mathematically predict the differences in accuracy and effort actually observed; we thus conclude that an additional feature, such as the content of the comment, must matter. Together with Figure 3 and Figure 4, this result supports our claim that our proposal to augment machine-generated patches with documentation that summarizes their effects and contexts is useful.

Previous investigations have indicated a lack of consensus on which metrics and concepts adequately explain maintainability [18, 23, 28]. The large number of low-impact features describing human accuracy, as measured by our study, reinforces this conclusion. In the following subsection, we elaborate this claim by showing

[3] We cannot include comment quality as a feature since it does not admit a simple and standard numerical measurement.

Measured Code Feature	Power
Ratio of variable uses per assignment	0.178
Code readability (Buse *et al.* metric [8])	0.157
Ratio of variables declared out of scope / in scope	0.146
Number of total tokens	0.097
Number of non-whitespace characters	0.090
Number of macro uses	0.080
Average token length	0.078
Average line length	0.072
Number of conditionals	0.070
Number of variable declarations or assignments	0.056
Maximum conditional clauses on any path	0.055
Number of blank lines	0.054
Number of variable uses	0.041
Maximum statement nesting depth	0.033
Number of comments	0.022
Curly braces on same line as conditionals	0.014
Majority of lines are longer than half-screen width	0.012

Table 2. Relative Predictive power of code features for modeling human accuracy when answering questions related to maintainability (ReliefF method). A value of 1.0 indicates the feature is a perfect predictor while a value of 0.0 suggests the feature is of no value when predicting the given response variable.

that humans often fail to recognize which features actually predict maintainability.

4.3 Do human maintenance intuitions match reality?

Software maintenance is a complex and demanding task that may be poorly-understood by some practitioners. We hypothesize that, in addition to not being able to identify what makes code more or less maintainable, humans may not be able to recognize maintainable code at all. This section compared subjective data collected during the study against measured human effort and accuracy to assess the validity of participants' intuitions about maintainability.

For each code segment in the study, participants were asked to provide not only an answer to the given question but also their confidence in their answer and a subjective judgment of how maintainable they believe the subject code to be (see Section 3.2). We found that participants' confidence in their answer correlated with their actual accuracy at a level of 0.18 ($p < 0.001$) using Pearson's product-moment statistic and with time to a correct answer at a level of -0.05 ($p < 0.05$). While there is no universal standard for the strength of a correlation, it is generally accepted that values between -0.3 and 0.3 are "not correlated" [9, 12]. We can thus conclude that participant confidence and participant accuracy are largely not linearly related.

Similarly, subjectively-reported values for maintainability exhibited a Pearson correlation of 0.13 ($p < 0.001$) with actual accuracy and of -0.04 ($p < 0.20$) with time to correct answer. We conclude that participant judgments of the task difficulty and their own actual performance are also largely not linearly related.

During the exit survey of the study, participants were asked to list all relevant code features that they felt affected maintainability in any way. The frequencies with which code features were reported are shown in Table 3. Because they are self-reported as free-form text, the descriptions of the features are slightly less formal, but overall participants felt that descriptive variable names, clear whitespace and indentation, and presence of comments affect maintainability the most. By comparing the "Power" column in Table 3 to the top 17 actually predictive features (Table 2), it can be seen that there is limited overlap between the set of features humans believe affect maintainability and those that are good predictors of

Human Reported Feature	Votes	Power
Descriptive variable names	35	0.000*
Clear whitespace and indentation	25	0.003*
Presence of comments	25	0.022
Shorter functions	8	0.000*
Presence of nested conditionals	8	0.033
Presence of compiler directives / macros	7	0.080
Presence of global variables	5	0.146
Use of goto statements	5	0.000*
Lack of conditional complexity	5	0.055
Uniform use and format of curly braces	5	0.014

Table 3. The top ten most-reported features by human participants when asked to list features they felt affected code maintainability. The second column lists the number of human participants who mentioned that feature; the third lists the relative predictive power of that feature when modeling actual human accuracy (ReliefF method; cf. Table 1). Features marked with an asterisk lack significant predictive power in the logistic regression model and are thus cases where humans misjudge the factors that affect maintainability.

it. For example, the feature most commonly mentioned by humans, "descriptive variable names", was manually annotated on the code snippets used in the human study by two separate annotators but was still found to have no predictive power. Similarly, the use of clear whitespace and indentation had a very minimal predictive effect — far below any of the top 17 predictive features. Of the three features most often reported by participants, only one (the presence of comments) has a significant predictive power.

4.4 Qualitative Analysis

We now present two case studies to help illustrate the perhaps-surprising result that the features participants claim are most influential with respect to maintainability do not uniformly result in higher accuracy. The human-created (and later reverted) patch shown in Figure 5 exhibits many of the features participants report make code more maintainable. For instance, comments (the third highest feature) account for almost half of the lines in the patch. Additionally, there is only a single one-letter potentially nondescript variable and, given the juxtaposition of its type, `GtkWidget`, the use of the letter `w` would not seem overly ambiguous in this case. It is clearly indented, uses whitespace well, is a short function, lacks gotos, and does not feature complex conditionals. Despite displaying qualities reported to aid in maintainability, participants correctly answered questions associated with this code only 20.0% of the time — significantly below overall average of 57.2% for all questions in the study. The original, un-patched version of this code contains less commenting and shows a slightly greater use of nondescript identifiers (i.e., `w` and `data` vs. `after_save_no_action`). Despite this, participants exhibit 6.1% greater accuracy, on average, for questions about the original code.

Figure 6 presents a code segment which exhibits relatively few of the features participants claim help to increase maintainability. Notably, the code lacks comments entirely and most of the variable names are terse. While humans subjectively report that these features should make it difficult to answer questions correctly, the average accuracy rate associated with the corresponding maintenance question was 75% — or 17.8% above the average and a 55% increase over the code depicted in Figure 5. While the code in Figure 6 does not match human-reported notions of maintainability, it does have higher-than-average for three out of the top fives features shown to actually predict maintainability in Table 2.

```
 1  - void file_save_cmd_cb(GtkWidget *w,
 2  -                  gpointer data) {
 3  + void file_save_cmd_cb(GtkWidget *w _U_,
 4  +                  gpointer data _U_) {
 5     /* If the file's already been
 6        saved, do nothing.  */
 7     if (cfile.user_saved)
 8       return;
 9  +  /*Properly dissect innerContextToken for
10  +   Kerberos in GSSAPI. Now, all I have to
11  +   do is modularize the Kerberos dissector*/
12     /* Do a "Save As". */
13  -  file_save_as_cmd(w, data);
14  +  file_save_as_cmd(after_save_no_action,
15  +                  NULL, FALSE);
16  }
```

Figure 5. Human-Reverted patch from `wireshark`. The patch modifies function `file_save_cmd_cb`, replacing lines 1–2 with lines 3–4 in addition to replacing line 13 with lines 14–15 and adding the comment on lines 9–11. Despite containing many features subjectively associated with high maintainability, participant accuracy on this snippet was 37% lower than average.

```
static void snmp_users_update_cb(
    void* p _U_, const char** err) {
  snmp_ue_assoc_t* ue = p;
  GString* es = g_string_new("");
  *err = NULL;
  if (! ue->user.userName.len)
    g_string_append(es,"no userName,");
  if (es->len) {
    g_string_truncate(es,es->len-2);
    *err = ep_strdup(es->str);
  }
  g_string_free(es,TRUE);
```

Figure 6. Original un-patched code snippet from `wireshark`. Despite having few features reportedly associated with maintainability, this code was particularly easy for participants to reason about (75% accuracy).

We conclude that there is a significant disconnect between human intuitions and reality regarding which code features affect maintainability. This discrepancy reinforces the need to investigate the root causes of maintainability with respect to guiding future development of both human-created and machine-generated patches. More directly, as automatically-generated patches become more commonplace, it is increasingly critical to know which features actually affect maintainability. Automated repair approaches can often produce multiple patches or target certain code features. Our results suggest, for example, that machine-generated patches should pay more attention to using locally-scoped variables and keeping the total size of the code low than to avoiding nested or complex conditionals.

5. Threats to Validity

Although our experiments show that automatically generated patches augmented with synthesized documentation are at least as maintainable as human written patches, our results may not generalize.

First, the code segments we selected may not be indicative of industrial systems. We attempted to address this threat by including code from a variety of application domains, including web servers, language interpreters, graphics processing, and compression utilities. However, our results may not generalize to commercially developed systems, closed-source programs, or programs with complex graphical user interfaces, for example. Furthermore, there are several threats related to failing to control for factors such as inherent code complexity or readability when measuring maintainability levels for various patch types. We attempt to mitigate these threats by randomizing the selection of both the code segments and the target questions when assigning tasks to patch types.

The participants selected may not accurately reflect industrial developers. We address this bias by soliciting a combination of senior level undergraduates, graduate students, and external participants. Participant self-reported computer science experience ranged from 1–35 years indicating a diverse population. We further restrict the population by removing participants whose skills may not be comparable with that of paid developers by imposing an accuracy cutoff. A related concern is that participants had no *a priori* experience with the code under study. Thus, while our experiments reflect situations in which developers are tasked with examining unfamiliar code, our results cannot generalize to maintenance tasks involving code developers are familiar with. Finally, the questions posed may not be indicative of all maintenance tasks. However, our paper focuses only on measuring maintainability as it relates to the questions developers ask when performing maintenance tasks as described by Sillito *et al.* [32] directly.

Two common threats associated with human studies are *training* or *fatigue* effects. A training effect occurs when participants do poorly at the beginning of a study and increase in accuracy with familiarity. Conversely, a fatigue effect occurs when participants grow tired or apathetic and their performance declines. We explicitly measured for these effects and found none: accuracy changed by only half a percentage point on average between the first half and the second half of the study.

Feature selection admits bias if the particular features measured are chosen based on either the code or the questions being asked. We mitigate this threat by choosing as features the union of those mentioned by participants and those used in previous studies exploring the maintenance process [8, 11].

6. Related Work

There are three general research areas related to this work: automated program repair, investigations of code maintainability, and synthesizing documentation.

6.1 Automated program repair

The recent research area of automated program repair aims to construct or evolve patches or runtime modifications to address defects in programs, but with little or no manual intervention. The GenProg technique used to generate the patches used in this paper leverages evolutionary computation to craft patches from existing code [20]. It has proven effective at fixing a significant fraction of indicative bugs at costs that are comparable to current industry standards [19].

There are a number of promising approaches to automated program repair. The ClearView tool uses dynamic invariant monitoring to recognize errors at runtime and creates patches by satisfying the offending invariants at the binary level [26]. Comparatively, the AutoFixE technique utilizes Eiffel-based contracts to pinpoint pre- and post-condition violations and construct patches to rectify specific failures [35]. Orlov and Sipper have applied evolutionary computation to the domain of unrestricted Java bytecode [25]. The AFix project uses static analysis techniques to expose atomicity-related errors and automatically generate patches for single-variable atomicity violations [16].

While arguments for functional correctness have been made for the aforementioned automated repair techniques, there has been no investigation into what effects such patches will have on the maintainability of systems. Furthermore, there has been little comparison between human-created and machine-generated patches with respect to understandability. By contrast, this paper investigate those two questions directly.

6.2 Code Maintainability

The maintenance process is the dominant cost of the software lifecycle [4, 31]. As such, there has been a significant amount of work aimed at both measuring maintainability aggregately and distinguishing the various subcomponents that affect a system's overall level of maintainability.

Aggarwal *et. al* argue that being able to understand and reason about code is central to the concept of maintainability [1]. Our study measures software understanding by gauging humans' abilities to answer various maintenance-related questions.

Welker *et al.* describe a single metric, the Maintainability Index, to statically determine the maintainability of source code [38]. This metric takes into account a number of other software quality metrics, including Halstead's program volume [13], McCabe's cyclomatic complexity [21], and average lines of code. In subsequent work, Heitlager *et al.* presented several criticisms of the original Maintainability Index and suggest potential improvements such as mapping system-wide characteristics of maintainability to source code properties and determining appropriate measurements for each of these properties [14]. Unlike this line of work, we do not measure maintainability through software metrics designed to model cost and effort related to humans. Instead, we gather an objective measure of maintainability directly from humans and attempt to elucidate maintainability *a posteriori*.

Kozlov *et al.* correlate various software metrics with the Maintainability Index described previously and find that no single analysis can definitely describe the relationships between maintainability and software quality metrics [18]. The idea that there is "no silver bullet" is echoed in the work of Riaz *et al.* [28] and Nishizono *et al.* [23]. We similarly found that no single code feature can predict maintainability, but rather a combination of many syntactical and semantic notions provides a reasonably accurate model.

6.3 Documentation Synthesis

Previous work has shown that documentation aids in program understanding and thus in the overall maintainability of a code base [10, 24]. There have been a number of proposed approaches for automatically documenting particular software aspects (e.g., exceptions [7] or API usage rules [5]). In this paper we propose to augmenting patches with a slightly modified version of the DELTADOC tool [6]. DELTADOC uses symbolic execution to reason about code changes and synthesize natural language explanations of the concrete events associated with the change and the predicates necessary to observe said events. We propose a number of small modifications to DELTADOC, generally designed to favor completeness over conciseness because patches produced by automated program repair tend to be small [37]. This paper presents the first strongly human-evaluated proposal to augment automated patches with synthesized documentation, as well as an evaluation of DELTADOC on newly-created patches without available human documentation — previous work has focused on comparisons against human-written documentation for the same patch [6, Fig. 7].

7. Summary and Conclusion

We have presented a human study of patch maintainability. Our study is large (157 humans participated; the most-accurate 102 produced over 2,100 data points), uses high-priority defects from realistic programs (4.8 million lines of code and 9000 tests), is controlled (we compare human-written to machine-generated patches for the same defects), and is grounded (we use human-reverted patches as a baseline indicative of wasted maintenance effort). The results shed light on the relative accuracy and effort required for participants to answer indicative maintenance questions on patched code. We also contrast the code-level features that humans think influence maintainability with those that are actually predictive of their performance. We acknowledge the research area of automated patch generation and include machine-generated patches in our study, proposing to augment them with human-readable synthesized documentation describing their effects and contexts.

When we control for accuracy, we find that it took participants 30% less time to correctly answer maintenance questions about machine-generated patches with synthesized documentation than to correctly answer questions about human-written patches, in a statistically significant manner. We find that our approach to automatically documenting machine patches is critical to this increase. This result is particularly compelling in light of the general perception that machine-generated patches lower code maintainability. Finally, we investigate code features related to human accuracy and find a strong disparity between what humans think matters to maintainability (e.g., shorter functions, the presence of comments, descriptive variable names) and what is actually predictive (e.g., how often variables are modified, how many referenced variables are locally scoped, etc.).

Understanding the maintainability of patches is crucial to software engineering, especially as automated program repair becomes more common. We believe this work provides a first step toward directly measuring the maintainability of patches, both human-written and machine-generated, as well as proposing particular approaches and treatments (i.e., synthesizing documentation, focusing on particular code features) that repair techniques, developers, and educators can consider for maintainability in the future.

Acknowledgements

We thank Claire Le Goues for insightful discussions. We are indebted to the anonymous reviewers their positive suggestions. We gratefully acknowledge the partial support of NSF (CCF 1116289, CCF-0954024, CCF-0905373), AFOSR (FA9550-07-1-0532, FA9550-10-1-0277), and DARPA (P-1070-113237).

References

[1] K. Aggarwal, Y. Singh, and J. Chhabra. An integrated measure of software maintainability. In *Reliability and Maintainability Symposium*, pages 235 –241, 2002.

[2] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1986. ISBN 0201100886.

[3] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *International Conference on Software Engineering*, pages 361–370, 2006. ISBN 1-59593-375-1.

[4] B. Boehm and V. Basili. Software defect reduction. *IEEE Computer Innovative Technology for Computer Professions*, 34(1):135–137, January 2001.

[5] R. P. L. Buse and W. Weimer. Synthesizing API usage examples. In *International Conference on Software Engineering (to appear)*, 2012.

[6] R. P. L. Buse and W. Weimer. Automatically documenting program changes. In *Automated Software Engineering*, pages 33–42, 2010.

[7] R. P. L. Buse and W. Weimer. Automatic documentation inference for exceptions. In *International Symposium on Software Testing and Analysis*, pages 273–282, 2008.

[8] R. P. L. Buse and W. Weimer. A metric for software readability. In *International Symposium on Software Testing and Analysis*, pages 121–130, 2008.

[9] J. Cohen. *Statistical power analysis for the behavioral sciences, 2nd edidtion*. Routledge Academic, 1988. ISBN 0805802835.

[10] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira. A study of the documentation essential to software maintenance. In *international conference on Design of communication*, pages 68–75, 2005. ISBN 1-59593-175-9.

[11] Z. Fry and W. Weimer. A human study of fault localization accuracy. In *International Conference on Software Maintenance*, pages 1–10, 2010.

[12] L. L. Giventer. *Statistical Analysis in Public Administration*. Jones and Bartlett Publishers, 2007.

[13] M. Halstead. *Elements of Software Science*. Elsevier, New York, 1977.

[14] I. Heitlager, T. Kuipers, and J. Visser. A practical model for measuring maintainability. In *International Conference on Quality of Information and Communications Technology*, pages 30–39, 2007.

[15] P. Hooimeijer and W. Weimer. Modeling bug report quality. In *Automated Software Engineering*, pages 34–43, 2007.

[16] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit. Automated atomicity-violation fixing. In *Programming Language Design and Implementation*, 2011.

[17] 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.

[18] D. Kozlov, J. Koskinen, M. Sakkinen, and J. Markkula. Assessing maintainability change over multiple software releases. *J. Softw. Maint. Evol.*, 20:31–58, January 2008.

[19] 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. In *International Conference on Software Engineering (to appear)*, 2012.

[20] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. GenProg: A generic method for automated software repair. *Transactions on Software Engineering*, 38(1):54–72, 2012.

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

[22] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Conference on Compiler Construction*, volume 2304, pages 213–228, 2002.

[23] 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 *International Conference on Software Maintenance*, pages 473 –481, sept. 2011.

[24] D. G. Novick and K. Ward. What users say they want in documentation. In *International Conference on Design of Communication*, pages 84–91, 2006.

[25] M. Orlov and M. Sipper. Flight of the finch through the java wilderness. *IEEE Transactions on Evolutionary Computation*, 15(2):166–192.

[26] 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. In *Symposium on Operating Systems Principles*, 2009.

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

[28] M. Riaz, E. Mendes, and E. Tempero. A systematic review of software maintainability prediction and metrics. In *International Symposium on Empirical Software Engineering and Measurement*, pages 367–377, 2009.

[29] M. Robnik-Šikonja and I. Kononenko. Theoretical and empirical analysis of ReliefF and RReliefF. *Mach. Learn.*, 53:23–69, October 2003.

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

[31] R. C. Seacord, D. Plakosh, and G. A. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[32] J. Sillito, G. C. Murphy, and K. De Volder. Questions programmers ask during software evolution tasks. In *Foundations of Software Engineering*, pages 23–34, 2006.

[33] 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.

[34] Symantec. Internet security threat report. In `http://eval.symantec.com/mktginfo/enterprise/white_papers/ent-whitepaper_symantec_internet_security_threat_report_x_09_2006.en-us.pdf`, Sept. 2006.

[35] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. In *International Symposium on Software Testing and Analysis*, pages 61–72, 2010.

[36] W. Weimer. Patches as better bug reports. In *Generative Programming and Component Engineering*, pages 181–190, 2006.

[37] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *International Conference on Software Engineering*, pages 364–367, 2009.

[38] K. D. Welker, P. W. Oman, and G. G. Atkinson. Development and application of an automated source code maintainability index. *Journal of Software Maintenance: Research and Practice*, 9(3):127–159, 1997.

[39] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. N. Bairavasundaram. How do fixes become bugs? In *Foundations of Software Engineering*, pages 26–36, 2011.