A Deeper Look into Bug Fixes: Patterns, Replacements, Deletions, and Additions

Mauricio Soto[1], Ferdian Thung[2], Chu-Pan Wong[1], Claire Le Goues[1], and David Lo[2]
[1]School of Computer Science, Carnegie Mellon University, Pittsburgh, PA
[2]Singapore Management University, Singapore
mauriciosoto@cmu.edu, {ferdiant.2013,davidlo}@smu.edu.sg,
{chupanw,clegoues}@cs.cmu.edu

ABSTRACT

Many implementations of research techniques that automatically repair software bugs target programs written in C. Work that targets Java often begins from or compares to direct translations of such techniques to a Java context. However, Java and C are very different languages, and Java should be studied to inform the construction of repair approaches to target it. We conduct a large-scale study of bug-fixing commits in Java projects, focusing on assumptions underlying common search-based repair approaches. We make observations that can be leveraged to guide high quality automatic software repair to target Java specifically, including common and uncommon statement modifications in human patches and the applicability of previously-proposed patch construction operators in the Java context.

Keywords

Automatic error repair; Maintainability; Human-like patches

1. INTRODUCTION

There has been considerable recent attention paid to automatic program repair (e.g., [5, 6, 9, 7]). One broad class of techniques in this space take a generate-and-validate approach: *generating* a large number of candidate patches using a pre-defined set of mutation operators, and then *validating* correctness with respect to a set of test cases. Mutation operators range from simple, coarse granularity statement-level mutations to human-constructed templates learned from a large corpus of previous human bug-fixing commits.

A considerable proportion of these works target programs written in C. Indeed, researchers targeting Java often start from or compare against Java-based implementations of techniques originally implemented for C [3, 5]. This is an oversight because Java and C are very different languages. Consider GenProg [6], which combines statement-level mutations into patches to address a particular defect. Which "Statements" (a semantic unit in C) should be manipulated in Java? Can we learn intelligent rules to inform candidate modifications (cf. Prophet [7], for C)? What types of semantic scoping should limit mutations, given the relative stringency of the Java compiler?

In this paper, we study bug-fixing commits to Java programs, taken from several million human-made bug fixes from Github. We study broad characteristics of these changes; the applicability of previously-proposed Java repair templates [5]; and the nature of additions and replacements of Java statements in bug-fixing commits. We make observations to directly guide future research in automatic repair of Java programs, to increase the success rate of such techniques and the degree to which the patches are human-like and therefore more readable and maintainable by human developers. This is important, because patch quality is an important concern in this area [10].

Related work. Zhong and Su [11] ask some of the same questions we do, on 6 projects. We study a much larger dataset [4] with 380,125 repositories; and we look at more statement types, as well as replacements. Similar to our study in Section 4, researchers have studied AST-level [8] and line-level changes [1] across bug fixing commits. Although the granularity differs, our approach is also novel with respect to scale of analysis. Barr et al. [2] studied changes to Java programs to understand the "Plastic Surgery Hypothesis" underlying certain types of program repair, an orthogonal concern. Kim et al. [5] manually analyze changes to Java programs to inform automated repair, and show that doing so results in higher-quality repairs; we study their templates heuristically on a different dataset. Their results motivate studies of human repairs, as it may result in better patches. Long and Rinard learn probabilistic models from bug fixes to C code [7]. Our analysis is significantly less precise and does not inform a new technique. Rather, it serves as a starting point for research on the repair of Java bugs.

2. DATASET AND CHARACTERISTICS

First, we broadly characterize bug-fixing commits to Java code. In this paper, we study the *September 2015/Github* dataset offered by Boa, including 554,864 Java projects with 23,229,406 revisions. Boa identifies 4,590,679 as bug fixing.

How many files are changed to fix a bug? Most program repair techniques assume that bugs are local to only one or two files. We consider a file *changed* if it is new, modified, or deleted in a commit. In total, 52,052,571 files were changed, an average of 11.3 files per bug-fixing commit. The median number of file changed is 2. Although the me-

MSR 2016 Austin, Texas USA
© 2016 ACM. ISBN 978-1-4503-4186-8/16/05. . . $15.00
DOI: http://dx.doi.org/10.1145/2901739.2903495

File Kind	Total	Average
Binary	752,945	0.16
Java (ERROR)	2,073,558	0.45
Java (JLS2)	2,607,413	0.57
Java (JLS3)	15,748,967	3.43
Java (JLS4)	83,798	0.02
Text	541,023	0.12
XML	6,818,299	1.49
UNKNOWN	23,426,568	5.10

Introducing	Total	Average
Class	729,201	0.16
Methods	3,186,867	0.69
Fields	6,076,646	1.32
Variables	924,259	0.20

Table 1: Top: File Types of Changed Files. Bottom: Introduction of new classes, methods, fields, local variables to fix bugs.

dian conforms to our intuition, the average is surprisingly high, suggesting a long tailed distribution. We hypothesize that other software artifacts (e.g. documentation) may be updated after bug fixing, or fixing commits may contain changes unrelated to fixing the bug (e.g. refactoring, feature addition). To better understand this phenomenon, we ask:

What are the types of those changed files? The top of Table 1 shows the types of the changed files in the Boa dataset. Text and binary files are changed least frequently. This is unsurprising, since such files are often documentation, and binaries should be changed rarely. XML files in Java projects usually represent build files; changing build files is unsurprising. Rather more surprising is how frequently UNKNOWN files are changed. Given that all projects are Java projects, it is unclear whether these are written in other programming languages. We hypothesize they might be test resources. JAVA (ERROR) files are those that Boa failed to parse. JLS2, JLS3, and JLS4 are Java files in different J2SE versions. If we group all Java types, on average, each bug fixing revision changes 4.47 Java files. This does not conform to our assumption of bug locality. To better understand what changes are usually made to those Java files, we next ask:

How often do developers introduce new classes, methods, fields, local variables in bug fixing commits? This question is interesting because most automatic program repair approaches do not create methods or variable declarations, nor do most explicitly consider object-oriented features, like fields or classes. The bottom section of Table 1 shows that introducing new classes or variables is rare. On average, we still have 0.69 new methods per bug fix. These methods include new test methods (annotated by @Test, for example), which is not too alarming. However, we note that 1.32 fields are created per bug fix, suggesting that automatic program repair approaches should pay attention to the state of the class when fixing Java programs.

3. BUG FIXING PATTERNS

One of the best-known program repair techniques to target Java is PAR [5]. PAR modifies Java code according to predefined templates, constructed by humans to cover a large set of bug fixes from an existing corpus. These templates provide important source of possible mutations to study to inform future directions. We therefore search for most of the PAR bug fixing patterns, estimating their prevalence in this dataset.

3.1 Detecting PAR templates using Boa

Boa's capabilities are powerful, but limited in the precision it enables in detection of PAR bug-fixing patterns. For example, it cannot directly `diff` two files. Rather than finding exact counts of bug fixing patterns, we approximate by processing pre- and post-fix files separately. Additionally, two of the 10 patterns cannot be easily detected by Boa, as we describe below. We search for the following patterns:

Altering method parameters (AMP). This template changes input method parameters. To detect this pattern, for both pre- and post-fix versions of a buggy file, we create a custom method call signature that contains the name, literal parameters, and variable parameters, all as string (complex expressions are listed as "OTHER" string). We discard signatures that appear both in pre- and post-fix versions, and then identify methods with the same name and number of parameters, but different signatures between versions.

Calling another method with the same parameters (MSM). This template changes a method name. To detect it, we create a method signature similar to the above for AMP, looking for methods with the exact same parameter signature but different names in pre- and post-fix versions.

Calling another overloaded method with one more parameter (COM). This template adds a parameter to a method. We create a method signature similar to the above, seeking methods with the same name, but with one more parameter in the post-fix version.

Add or remove a branch condition (ABC). This template adds or removes a condition. We count the number of *logical and* and *logical or* inside if conditional expression for both pre- and post-fix version. We assume that the addition/removal of logical operators indicate addition/removal of condition. If there is a difference in count between versions, we consider an instance of the ABC pattern is found.

Initializing an object (IAO). This template adds a initialization to object declaration. We count the number of NEW expressions in variable declarations for both pre- and post-fix versions. If there is a larger count in post-fix version, we consider that an instance of the IAO pattern is found.

Adding a null checker (ANC). This template inserts a condition to check whether an object is null. To detect ANC pattern, we count the number of if conditional expressions that contains *!=null* or *==null* for both pre- and post-fix version. If there is a larger count in post-fix version, we consider that an instance of the ANC pattern found.

Adding an array out of bound checker (AOB). This pattern inserts a condition check to check that an array index is within bound right before the index is used. We count the number of if conditional expression that contains *expr<var.length* or *var.length>expr* for both pre- and post-fix versions. If there is a larger count in post-fix version, we consider that an instance of the AOB pattern is found.

Adding a collection out of bound checker (COB). This pattern inserts a condition to check that a collection index is within bound before it is used. To detect COB pattern, we count the number of if conditional expression that contains *expr<col.size()* or *col.size()>expr* for both pre- and post-fix version. If the is a larger count in post-fix version, we consider that an instance of the COB pattern is found.

These patterns cover 8 of the 10 bug fixing patterns in PAR. We do not investigate (1) Class Cast Checker (cannot be analyzed because Boa does not support the *instanceof* expression); and (2) Expression Changer (requires us to track scope between pre- and post-fix version; not easily supported by Boa). Investigating 8 out of 10 patterns provides an informative approximation of their prevalence.

3.2 Results

	#Appearance	Percentage
AMP Pattern	901,083	1.95%
MSM Pattern	783,073	1.69%
COM Pattern	270,128	0.58%
ABC Pattern	1,959,377	4.23%
IAO Pattern	1,308,006	2.82%
ANC Pattern	1,340,561	2.90%
AOB Pattern	128,016	0.28%
COB Pattern	262,915	0.57%

Table 2: Frequency of Bug Fixing Patterns

Table 2 estimates the prevalence of PAR templates in the Java dataset. The most common pattern we observed is ABC (add or remove a branch condition); and the least common pattern is AOB (adding an array out of bound checker). If we conservatively assume that these patterns never appear together, they cover 14.78% of buggy files. In the dataset studied by Kim et al., the overall 10 patterns cover almost 30% of real patches [5]. Although we only analyze 8 out of 10 bug fixing patterns, it is unlikely the final two will cover the difference. To even get close to 30%, each of the patterns need to cover around 7% of buggy files and we still assume that the patterns never appear together. This suggests that these fixing patterns might not generalize beyond an initial dataset, and that more work remains to expressively characterize human bug-fixing behavior.

4. STATEMENT-LEVEL MUTATIONS

Some automatic repair approaches seek generality by using higher-granularity mutation operators such as statement-level addition, deletion and replacement. To support the generation of high-quality patches, we analyze how developers mutate source code to fix bugs at this granularity level.

Detection Approach Because direct diffs are difficult to identify on this dataset, we heuristically approximate the extent to which one statement type appears to be "replaced" by another. For each modified file, we count the number of appearances of each statement type in the file pre- and post-commit. We then compare the results to see how many of each statement type was removed, and how many inserted, to roughly characterize the types of replacement that happen at a per-file level. Note that this analysis doesn't distinguish the replacement of the same statement kind, since we are counting the amount of appearances of each statement kind. We follow a similar approach to approximately count deletions and insertions. For each bug fixing revision r and each statement kind k, we compare the count of statements of kind k in revision r and $r-1$.

Potential replacements. Table 3 shows the replacement likelihood for our dataset (each cell shows the percent of the time that the statement in the *row* was replaced by a statement of the type in the *column*). For example, the corresponding to the `Do` row (row 5) and `Assert` column (column 2) shows 0.81, indicating that `Do` statements were replaced by `Assert` statements 0.81% of the times.

Additional analysis (raw numbers not shown) show that the most common replacement replaces `Return` statements with `If` statements (in 30,489 files). The second most common replacement replaces an `If` statement with a `Return` (28,536 incidences). By contrast, the least common replacement was an `Assert` statement replacing a `TypeDecl`, which we did not observe. The second least common replacements were replacing `Do` statements or `Label`s for a `TypeDecl`; we observed these once each.

The most common *replacement* statement (the statement kind that most commonly replaces others) is the `If` statement (101,366 appearances). The least common *replacement* is the `TypeDecl` (447 appearances). The most common *replacee* was the `Return` (111,938 appearances); the least common *replacee* was again the `TypeDecl` (399 appearances).

Potential deletions and insertions From Figure 1, we can see that `expression`, `If`, `Return`, `for` and `Try` statements are both added and deleted most often as compared to the other statement times. These findings indicate that most bugs were fixed by changing control flow.

An important study that complements ours is the repair model approach proposed by Martinez and Monperrus [8], which proposes a probability distribution suggesting when to apply which kind of edit. Although their approach can trace more fine-grain AST-level changes, our results are consistent with their findings. For example, their empirical analysis [8] shows that method invocations, if statements, and variable declarations are added/deleted/updated most often, which is also illustrated in Figure 1 (Boa groups method invocations and variable declarations into the `Expression` category). Our study complements there at a much larger scale (we study 380,125 repositories with 23,229,406 revisions as compared to the 14 repositories in the prior work).

5. DISCUSSION

Threats to validity. The correctness of our analysis depends on both our programs and Boa and its DSL. For example, we rely on Boa to identify bug fixing revisions; however, precisely accomplishing this is an open problem. To mitigate the risk of implementation errors, we released our scripts[1]. Because Boa does not provide an easy mechanism to identify precise, statement-level diffs between revisions, our template matching and analysis of code changes (by counting each statement kind) only provide estimates of behavior; we consider our results as informative approximations.

Lessons learned. The findings of our study provide useful insights for automatic program repair tools in Java. It suggests that patterns proposed by the state-of-the-art approaches for Java are insufficient to cover the extent of bug fixes in our dataset. Adding more patterns is one solution, but may not be a general one. Our findings further suggest that mutation based program repair may need to consider field or method insertion to achieve human-comparability in patches. This finding indicates that direct translation of mutation based approaches from C to Java is not likely to succeed due to the different language constructs. A deeper

[1]https://github.com/chupanw/BoaChallenge

	Assert	Break	Continue	Do	For	If	Label	Return	Case	Switch	Synch	Throw	Try	TypeDecl	While
Assert	-	7.48	3.76	0.53	8.30	23.05	0.31	20.04	4.90	4.62	1.30	13.50	7.23	0.03	4.95
Break	1.00	-	4.08	0.60	9.93	26.03	0.13	25.39	2.48	1.57	1.79	8.39	11.73	0.10	6.77
Continue	1.74	9.42	-	1.28	11.39	18.25	0.35	22.60	3.80	2.85	2.17	8.98	9.42	0.11	7.63
Do	0.81	5.26	6.60	-	9.44	14.21	0.18	15.86	3.73	1.67	1.97	5.88	6.39	0.03	27.98
For	0.86	6.28	3.19	0.79	-	22.89	0.09	21.08	5.01	3.34	1.87	10.01	10.71	0.08	13.79
If	1.64	8.43	2.87	0.60	13.49	-	0.24	26.46	7.45	4.80	2.85	9.89	15.11	0.08	6.11
Label	1.30	8.33	7.86	1.11	5.18	22.85	-	15.17	3.05	2.04	14.62	10.45	4.16	0.09	3.79
Return	1.13	9.41	3.11	0.49	13.33	27.24	0.24	-	5.59	3.65	2.55	14.91	12.61	0.12	5.61
Case	0.78	2.84	2.84	0.39	10.27	31.79	0.16	22.40	-	0.46	2.07	7.37	11.69	0.08	6.87
Switch	1.14	2.72	3.80	0.55	11.07	34.14	0.13	21.86	0.75	-	1.53	8.65	9.02	0.05	4.58
Synch	0.80	6.57	2.28	0.43	10.21	24.18	0.05	19.77	6.35	2.07	-	9.16	12.16	0.04	5.93
Throw	2.11	6.57	2.58	0.48	11.87	18.84	0.17	32.28	4.64	3.30	2.74	-	10.08	0.07	4.27
Try	0.71	7.41	3.02	0.66	11.73	27.75	0.11	23.24	5.63	2.65	2.58	8.99	-	0.09	5.42
TypeDecl	0.00	4.51	7.52	1.00	10.28	21.05	0.50	17.79	6.02	1.75	2.01	9.27	11.53	-	6.77
While	0.72	8.02	3.82	1.96	23.16	19.78	0.12	16.48	6.56	3.09	1.64	6.81	7.80	0.04	-

Table 3: Likelihood of replacing a statement type (row) by a statement of another type (column), for Java.

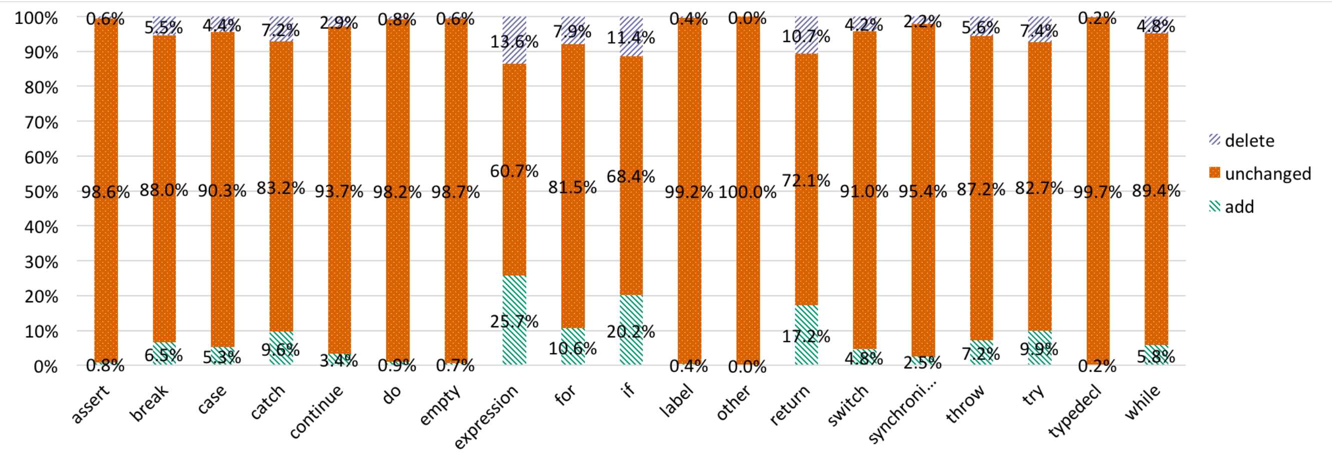

Figure 1: For each kind of statement, how many bug fixing revisions add/delete statements of the same kind?

look suggests that such techniques may benefit from leveraging probabilistic knowledge of what types of statements are commonly inserted, deleted, or replaced in Java bug-fixing commits. Again, the probabilistic knowledge for C is not likely to work for Java program if different coding conventions are considered. Overall, our findings motivate additional study of repair in Java, as assumptions that underlie approaches that target C are unlikely to translate directly.

6. ACKNOWLEDGMENTS

This work is partially supported by the US Department of Defense through the Systems Engineering Research Center (SERC), Contract H98230-08-D-0171.

7. REFERENCES

[1] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and M. Di Penta. LHDiff: A Language-Independent Hybrid Approach for Tracking Source Code Lines. *ICSM*, pages 230–239, 2013.

[2] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro. The plastic surgery hypothesis. In *FSE*, pages 306–317, 2014.

[3] F. DeMarco, J. Xuan, D. Le Berre, and M. Monperrus. Automatic repair of buggy if conditions and missing preconditions with smt. In *CSTVA*, 2014.

[4] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *ICSE*, pages 422–431, 2013.

[5] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *ICSE*, pages 802–811, 2013.

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

[7] F. Long and M. Rinard. Automatic patch generation by learning correct code. In *POPL*, 2016.

[8] M. Martinez and M. Monperrus. Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Software Engineering*, 20(1):176–205, 2015.

[9] S. Mechtaev, J. Yi, and A. Roychoudhury. DirectFix: Looking for simple program repairs. In *ICSE*, 2015.

[10] Z. Qi, F. Long, S. Achour, and M. Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *ISSTA*, pages 24–36, 2015.

[11] H. Zhong and Z. Su. An empirical study on real bug fixes. In *ICSE*, pages 913–923, 2015.