A deeper look into bug fixes: patterns, replacements, deletions, and additions

Mauricio Soto
Carnegie Mellon University
Pittsburgh, PA
mauriciosoto@cmu.edu

Ferdian Thung
University
City
email@domain.com

Chu Pan Wong
Carnegie Mellon University
Pittsburgh, PA
email@domain.com

Claire Le Goues
Carnegie Mellon University
Pittsburgh, PA
clegoues@cs.cmu.edu

David
University
City
clegoues@cs.cmu.edu

ABSTRACT

With the rise of automated program repair in the last couple of years, there are a lot of questions to remain unanswered. We performed a study on the Github dump of September 2015 in which we analyzed all the fixing revisions of this dataset, and found all the replacements made in these 46,301,429 files from real life projects in Github.

We found very valuable information to guide automatic software repair, such as the most common and least common replacements made by human developers in order to build a successful patch; we analyzed the most and least common statement to replace others, and the most and least likely statement to get replaced by other. We also analyzed, once you have a statement that you want to replace (a faulty statement), what are the most likely statements to replace it for. These information will help automatic program repair tools to be more maintainable and user friendly, taking it one step closer to what humans developers do to repair their code.

Keywords

Automatic error repair; Maintainability; Human-like patches

1. INTRODUCTION

Automatic bug repair is the branch of computer science that deals with automated ways to repair errors in software. There have been several approaches taken towards improving the different methodologies to repair bugs in software automatically [4][7][6] [3], one of the most accepted approaches so far has been GenProg [7][6], an evolutionary program repair tool which applies four different kinds of possible edits to code in order to find a patch for a given bug. These changes are the following: Delete, Replace, Swap and Append.

Another well known approach is PAR [4], which creates 10 different repair templates and applies them to the buggy code in an effort to repair it. In this paper we have taken 7 out of the 10 PAR templates and tested how common they are in the repairs made by programmers in the latest official data dump of Github as of September 2015 provided by [1].

Our research aims to provide guidelines to improve three out of four of these possible changes: Delete, Replace and Append. It will provide the data necessary for this tool and other approaches to have a guide on what is the way in which human programmers change their code when coding a fix for a bug. This way it will make it more likely for automatic error repair approaches to succeed in finding a patch for a particular error, and also, to provide automatic error repair software with heuristics to make the patches more human-like and therefore more readable and maintainable by human developers.

In this article we study the frequency with which human programmers replace, delete and append different statements to their source code in order to fix a bug.

MSR 2016 Austin, Texas USA
© 2016 ACM. ISBN 123-4567-24-567/08/06.
DOI: 10.475/123_4

2. METHODOLOGY

Our methodology is applied to the latest official data dump of Github as of September 2015 provided by [1]. In the following subsections, we first describe data source and objective of our study. We then describe how we detect bug fixing patterns in PAR to find out how common they are the dataset. Last but not least, we describe our approach to detect Delete, Replace, and Append bug-fixing patterns.

2.1 Data Source and Research Objective

We used the Boa platform [1] to be able to query the Github repository looking for patterns that humans use to patch errors in the their code. Our main goal is to provide guidelines for automatic error fixing approaches by providing necessary information regarding historical replace, delete and append statements in human written patches.

2.2 Detecting PAR Bug Fixing Pattern

The bug fixing patterns that we analyze in this work are adopted from Kim et al. paper [4]. The following is the description of each considered pattern.

Table 1: Amount of appearances per replace kind

	Assert	Break	Continue	Do	For	If	Label	Return	Case	Switch	Synchronized	Throw	Try	TypeDecl	While
Assert	-	340	232	27	517	1764	14	1260	211	191	82	1004	352	0	206
Break	455	-	1254	176	3757	9096	90	10538	771	455	672	3123	3660	18	2278
Continue	229	1384	-	221	1909	3094	85	3486	772	635	233	1227	1493	30	1086
Do	32	205	171	-	474	648	12	543	106	92	44	227	325	4	558
For	505	3366	1516	316	-	14547	56	14924	2791	1852	1044	5642	5796	41	6582
If	1402	8820	2430	476	13703	-	247	30489	8639	5710	2472	8960	13714	84	5622
Label	19	44	46	6	56	255	-	270	44	22	5	80	53	2	33
Return	1219	8603	3009	531	12620	28536	164	-	6088	3657	2021	15346	11486	71	4684
Case	298	839	506	125	3002	8039	33	6261	-	126	649	2206	2784	24	1864
Switch	281	533	379	56	1997	5175	22	4090	126	-	212	1571	1311	7	878
Synchronized	79	606	289	66	1122	3070	158	2853	562	256	-	1303	1274	8	466
Throw	821	2844	1195	197	5993	10664	113	16688	2002	1446	936	-	4443	37	1934
Try	440	3976	1254	214	6414	16294	45	14119	3178	1509	1243	4795	-	46	2218
TypeDecl	2	35	15	1	49	83	1	137	21	9	4	35	46	-	11
While	301	2295	1016	937	8257	6587	41	6280	1866	766	606	2028	2677	27	-

1. **Altering method parameters (AMP)**
   Example: obj.method(v1,v2) → obj.method(v1,v3)
   Description: This pattern changes the input for method parameters.

2. **Calling another method with the same parameters (MSM)**
   Example: obj.method1(param) → obj.method2(param)
   Description: This pattern changes the method name.

3. **Calling another overloaded method with one more parameter (COM)**
   Example: obj.method(v1) → obj.method(v1,v2)
   Description: This pattern adds one more parameter to the method.

4. **Changing a branch condition (CBC)**
   Example: if(a != b) → if(a != b && c == 0)
   Description: This pattern adds or removes condition.

5. **Initializing an object (IAO)**
   Example: Type obj → Type obj = new Type()
   Description: This pattern adds a initialization to object declaration.

6. **Adding a null checker (ANC)**
   Example: obj.m1() → if(obj!=null)obj.m1()
   Description: This pattern inserts a condition to check whether an object is null or not in order to prevent unexpected state.

7. **Adding an array out of bound checker (AOB)**
   Example: arr[idx]=0 → if(idx<arr.length)arr[idx]=0
   Description: This pattern inserts a condition to check that an array index is within bound right before the index is used.

To analyze bug fixing patterns, we utilizes Boa's language infrastructure [1]. It contains a large number of dataset and provide a domain specific language to perform analysis on source code. Despite this advantage, Boa still have limited capability that prevents us from performing precise detection of bug fixing patterns. For example, it is unable to perform a diff between two file versions, which is needed to compare the pre-fix and the post-fix version of a buggy file. Thus, rather than finding the exact count of bug fixing patterns, we will find an approximate count of those patterns. The following are descriptions on how we detect each pattern.

1. **AMP Pattern**
   To detect AMP pattern, for both pre-fix and post-fix version of a buggy file, we create a custom method call signature. This signature contains method name, literal parameter, and variable parameter. we consider parameter containing custom expression as OTHER. This is due to inexistence of function that can print Abstract Syntax Tree (AST) back to source code. we discard method signatures that appear both in pre-fix and post-fix version. We then find whether there exists method with the same name and the same number of parameters, but different parameter signatures in pre-fix and post-fix version. If it exists, we consider that AMP pattern is found.

2. **MSM Pattern**
   To detect MSM pattern, we create method signature similar like when detecting AMP pattern. This time, we find whether there exists method with the exact same parameter signature but different name in pre-fix and post-fix version. If it exists, we consider that MSM pattern is found.

3. **COM Pattern**
   To detect COM pattern, we create method signature similar like when detecting AMP and COM pattern. This time, we find whether there are exists method with the same name, but the number of parameters in pre-fix and post-fix version differs by one. If it exists, we consider that COM pattern is found.

4. **CBC Pattern**
   To detect CBC pattern, we count the number of *logical and* and *logical or* inside if conditional expression for both pre-fix and post-fix version. we assume that

the addition/removal of logical operators indicate addition/removal of condition. If there are count differences between pre-fix and post-fix version, we consider that CBC pattern is found.

5. **IAO Pattern**
   To detect IAO pattern, we count the number of NEW expression in variable declaration for both pre-fix and post-fix version. If there are count differences between pre-fix and post-fix version, we consider that IAO pattern is found.

6. **ANC Pattern**
   To detect ANC pattern, we count the number of if conditional expression that contains *!=null* or *==null* for both pre-fix and post-fix version. If there are count differences between pre-fix and post-fix version, we consider that ANC pattern is found.

7. **AOB Pattern**
   To detect AOB pattern, we count the number of if conditional expression that contains *expr<var.length* or *var.length>expr* for both pre-fix and post-fix version. If there are count differences between pre-fix and post-fix version, we consider that AOB pattern is found.

2.3 Replace Query

The replace query is a query in which first we set the project that we are going to use, which is the Github dump of September 2015 provided by [1].

We then create a visitor to traverse all the nodes of this repository. First thing the visitor does is to ask weather the node is a Revision, and if it is, then it asks if it is a fixing revision. Therefore we are analyzing only files that were already committed and are now being reviewed to fix an error. We use Boa's infrastructure to do this by visiting the node they label as "Revision" and then calling the function defined by Boa "isfixingrevision()".

A replacement needs two statements, one that is deleted and another one that is inserted. We then go through all the files that have been changed in this data set asking for two conditions: it must be a fixing revision and there must have been a previous version of this file before. If these conditions are met then we initialize two counters: one for the first statement we are analyzing (the one that was deleted), and another one for the second statement we are analyzing (the one that was inserted).

At this point we visit each statement in the previous state of that changed file, before the change was applied and we count the amount of appearances of the first statement we are analyzing (the one that was deleted), and then we count the amount of appearances of the second statement we are analyzing (the one that was inserted). We save these values, and do the same for the latest version of the file and save these values as well.

We then compare these results to see if either the first and second statements being analyzed increased or decreased. Depending on these we have two conditionals:

If the amount of occurrences of the first statement decreased and the amount of occurrences of the second statement increased on the same file, then we say that the first statement was replaced by the second statement in that file. Likewise, if the amount of occurrences of the first statement increased and the amount of occurrences of the second statement decreased on the same file, we say that the second statement was replaced by the first statement in that file. We count the amount of files in which this happens for each of these two cases.

We need to test this with all the possible combinations of two different statement types since we want to know what is the probability of an specific statement to be replaced by another specific statement. For example, we want to know how often does a For loop gets replaced by a While loop in a successful human written patch, or how often does a Break statement gets replaced by a Continue statement in a successful human written patch.

Because of this we have created Table 1 which records the amount of replaces happening from one kind of statement to another. It is a matrix of fifteen by fifteen slots that describes which statements are being replaced by which other.

The numbers in the slots represent the amount of files in which that replacement took place. In the first column we have the list of statements that are being replaced, and on the top row we have the list of statements for which that statement was replaced for. For example, if we take the intersection of row 5 (DoStatement row) and column 2 (AsserStatement column), we have a 32, which means that from the 46,301,429 files categorized by Boa as fixing revisions in Github in the month of September 2015, only 32 of them replaced a DoStatement for an AssertStatement.

It is important to notice that this analysis doesn't consider replacing an statement kind for a different version of the statement kind, since it is counting the amount of appearances of each statement kind. So, if for example, the developer modifies the condition inside of an if statement, the amount of if statements in the file is going to remain the same

2.4 Delete/Append Query

The Delete/Append Query is similar to the Replace Query with some variations to count for the amount of times that a file was changed and a particular statement kind was either deleted or append, and the rest of the statement kinds remained the same.

We created a Boa file in which first we create an output variable to count all the amount of files in which a statement kind was deleted or appended. Similar to the previous query, we visit all the files labeled as fixing revisions, then visit every node of the previous version of the revision, counting the amount of statement kinds of each one of the possible fifteen different ones. We then do the same for the later version of the revision, counting the different statement kinds.

Finally, we compare the amounts of the previous version, with the new version. For each of the revisions, we ask if the amount of statements stays the same for all the statement kinds, except for X kind of statement. When the amount of statements is different in the previous version than the latest version for X kind of statement. Then we ask if X kinds of statement is greater in the previous version or the latest version. If the amount of appearances of X kind of statement is greater in the previous version than the latest version, and all the remaining fourteen statement kinds have the same amount in the previous than the latest version, then we say that one or more of the statements of kind X were deleted. Otherwise, if the latest version has a greater amount of appearances of X kind of statement and the rest have the same amount of appearances in both the previous and the latest version, then we say that one or more statements of kind X were appended or added to the fixing revision.

2.5 Data Analysis

Our queries were ran on the dump of Github of September 2015. In these real life projects we found 46,301,429 files being fixed. In these files, there is a total of 517,281 statement kinds being replaced, and 511,198 replacing these statements kinds.

We analyzed each of these files looking for statement kinds being either replaced by other kinds, deleted or appended to the file.

We found that the most common Replacer statement kind, meaning the statement kind that most commonly replaces others is the **If Statement** with 101,366 appearances. The least common Replacer, meaning the statement kind which is less likely to replace other statement kinds is the **Type Declaration Statement** with 447 appearances.

The most common Replacee, which is the most likely statement kind to be replaced by another statement kind is the **Return Statement** with 111,938 appearances. The least common Replacee, which is the least common statement to be replaced by another statement kind is the **Type Declaration Statement** with 399 appearances.

2.5.1 Likeliness of replacement

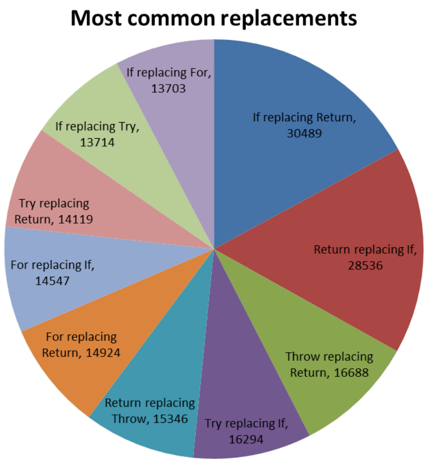
This study is directly applicable to automatic software repair. Most common tools for automatic software repair [4][7][6] first find a faulty statement using several different fault localization techniques [5], and once they have that faulty statement, they apply different kinds of edits to it, usually at random. Which makes it equally likely to replace an statement for another without looking at the kinds of statements being replaced.

With this study, we are able to make a more accurate guess, when having a known faulty statement, to which statement it is more likely to replace it for. By performing simple math with the data of Table 1, we can find the most likely replacements that will end up being a successful patch.

For example, if we have a faulty statement that happens to be a For Statement. Then we can look up in Table 1 the For Statement Column (6th column). If we add all the values in that column, we get 59,870 changes in which a For Statement was replaced by some other kind of statement. Then we calculate the percentage of each of the values in the column to see the likeliness of each of the statements to replace it for, and in this example we see that the top 3 most likely statements to replace a For Statement are an If statement with a 22.88% likeliness, followed by a return statement with a 21.08%, and third is a While statement with a 13.73% of likeliness to replace a For Statement.

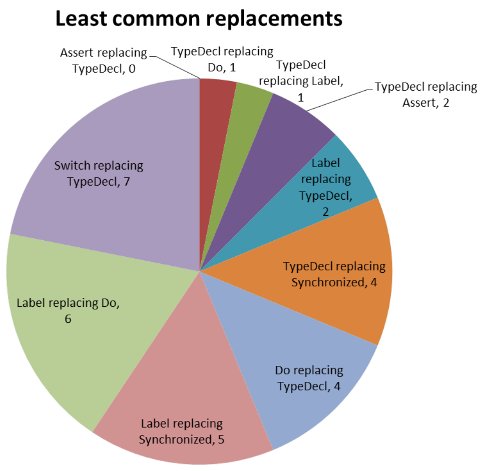
2.5.2 Most common replacements

Our results show that the most common replacement is when developers replace a Return statement for an If statement. This showed up in 30,489 files in the code database. The second most common replacement is an If statement being replaced by a Return statement, which had 28,536 appearances in the code database. Third to that, is a Return statement being replaced by a Throw statement.



2.5.3 Least common replacements

On the opposite side of the data, we can see that the least common replacement was an Assert statement replacing a Type Declaration statement. This was the only replacement to have zero appearances in our search through the 46,301,429 files being fixed. Close to it, as the second least common replacement was replacing a Do statement for a Type Declaration statement, with one appearance in our code database, and on third place, the replacement of a Label being replaced by a Type Declaration Statement had one appearance in our code database.



3. RELATED WORK

A similar study was performed by Dongsun Kim et al. [4] in which they look for the most common ways in which programmers patch bugs in software. The researchers developed a variation of the tool Genprog [7][6] with several different templates resembling patters programmers use to patch bugs.

A good complement for this paper is [2], where they search additions and deletions in a smaller data set. In this paper, we focus on the replacements, which have been left out of the analysis of [2].

4. CONCLUSIONS

File Kind	Total Number	Average
BINARY	752,945	0.16
SOURCE_JAVA_ERROR	2,073,558	0.45
SOURCE_JAVA_JLS2	2,607,413	0.57
SOURCE_JAVA_JLS3	15,748,967	3.43
SOURCE_JAVA_JLS4	83,798	0.02
TEXT	541,023	0.12
XML	6,818,299	1.49
UNKNOWN	23426568	5.10

Table 2: File Types of Changed Files

The findings of our study provide a set of useful guidelines for automatic program repair tools with which their search for a patch can be now improved by having the knowledge of how likely it is to replace a faulty statement by the different kinds of statements available.

5. RESULTS

This section investigates how do developer fix bugs from three different perspectives. To begin with, we show some characteristics of real-world bug fixes. Afterwards, we look deeper to find out what kinds of statements are usually added, modified or deleted by human developers to fix bugs. Lastly, we show how common PAR's fixing templates are in our dataset.

5.1 Characteristics

This subsection shows a bird's-eye view of real-world bug fixes at ultra-large-scale. We did our analysis on the *2015 September/Github* dataset on Boa. This dataset includes $7,830,023$ Java projects with $23,229,406$ revisions. Among all the revisions, $4,590,679$ revisions are identified by Boa as bug fixing revisions. Our analysis in this subsection focuses on these bug fixing revisions. We hope that this population is big enough to reflect the bug fixing practice.

To begin with, we are interested in:

How many files are changed to fix a bug?

To answer this question, we identified all the bug fixing revisions in our dataset and count the number of changed files. In total, $52,052,571$ files got changed. Thus, on average each bug fixing revision changes 11.3 files. This number is surprisingly high because most automatic program repair techniques assume that bugs are local in most cases. There could be several reasons behind this number. It could be the case that other software artifacts (e.g. documentation) are updated after bug fixing. It could also be the case that bug fixing commit also contains changes that are not related to fixing bug (e.g. new features, refactoring). To better understand why so many files are changed, we continue to ask:

What are the file types of those changed files?

In Boa, each changed file is represented by a *ChangedFile* node, and it has a field called *FileKind*, describing the type of each file. Table 2 shows the file types that Boa could identify. To answer this question, we iterate all the changed files in each bug fixing revision and count the total number for each file type. Results are shown in Table 2.

Introducing	Total Number	Average
Class	729201	0.16
Methods	3186867	0.69
Fields	6076646	1.32
Variables	924259	0.2

Table 3: File Types of Changed Files

SOURCE_JAVA_ERROR file kind represents files that Boa failed to parse. *JLS2* refers to Java files that are written in all versions of Java languages up to and including J2SE 1.4. *JLS3* refers to J2SE 1.5, and JLS4 refers to J2SE 1.6 and 1.7. Boa documentation does not explicitly explain what is *TEXT* and what is *UNKNOWN*. Our assumption is that *TEXT* refers to files whose file name ends with ".txt", and *UNKNOWN* refers to all other kinds of files (e.g. .c file, .cpp files).

From Table 2, we can see that *TEXT* files and *BINARY* files are changed least frequently. This is not surprising because *TEXT* files are usually documentation files, and *BINARY* files should only be changed in rare case (e.g. milestone, new release, dependency update). If we consider all 4 kinds of Java files as one kind, then on average each bug fixing revision changes 4.47 Java files. But still, it does not conform to our assumption of bug locality. *XML* files in Java projects usually represents build files, thus changing one build file after each bug fixing is not surprising. What surprises us is that *UNKNOWN* files are changed most frequently. Note that all the projects in our dataset are Java projects, so it is reasonable to assume that these files are not source files that written in other programming languages. One possible explanation is that they are test resources. However, due to the limitations of Boa, we can not look into the content of these *UNKNOWN* files.

Since Boa does not distinguish Java files that related to functionality and Java files that related to testing, it could be the case that among those 4.47 changed Java files, some of them are related to testing, such as modifying the test cases and creating new test classes. To better understand what changes are usually made to Java source code files, we continue to ask:

How often do developers introduce new classes, methods, fields, variables in bug fixing commits?

This research question is interesting because most automatic program repair approaches could not create new classes/methods/fields/variables to fix bugs. Most approaches guess and synthesize the patches based on existing source code.

Table 3 shows our analysis results. This table shows that introducing new classes and new variables are rare. This is good news for automatic program repair approaches. On average, we still have 0.69 new method per bug fix. If we consider the possibility of creating new test method (i.e. methods that annotated by @Test if JUnit is used), this number is not that scary. However, what surprises us is that 1.32 fields are created per bug fix. This indicates that automatic program repair approaches should pay attention to the state of the class when fixing Java programs.

6. REFERENCES

[1] Robert Dyer et al., *Boa: A Language and Infrastructure for Analyzing Ultra-Large-Scale Software Repositories*, 35th International Conference on Software Engineering; ICSE 2013, San Francisco CA, USA

[2] Hao Zhong, Zhendong Su *An Empirical Study on Real Bug Fixes*, In Proceedings of ICSE 2015, Firenze, Italy, May 16-24, 2015.

[3] Kai Pan et al. *Towards an understanding of bug fix patterns*, Journal Empirical Software Engineering Volume 14 Issue 3, June 2009 Pages 286 - 315

[4] Dongsun Kim et al. *Automatic patch generation learned from human-written patches*, Proceeding ICSE '13 Proceedings of the 2013 International Conference on Software Engineering

[5] Zachary P.Fry and Westley Weimer *A human study of fault localization accuracy*, ICSM '10 Proceedings of the 2010 IEEE International Conference on Software Maintenance

[6] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, Westley Weimer *GenProg: A Generic Method for Automated Software Repair.* IEEE Trans. Software Engineering 38(1): 54-72 (January/February 2012)

[7] Westley Weimer, ThanVu Nguyen, Claire Le Goues, Stephanie Forrest *Automatically Finding Patches Using Genetic Programming.* International Conference on Software Engineering (ICSE) 2009: 364-374