Extracting Structural Information from Bug Reports

Nicolas Bettenburg
Saarland University
nicbet@st.cs.uni-sb.de

Rahul Premraj
Saarland University
premraj@cs.uni-sb.de

Thomas Zimmermann
University of Calgary
tz@acm.org

Sunghun Kim
MIT CSAIL
hunkim@csail.mit.edu

ABSTRACT

In software engineering experiments, the description of bug reports is typically treated as natural language text, although it often contains stack traces, source code, and patches. Neglecting such structural elements is a loss of valuable information; structure usually leads to a better performance of machine learning approaches. In this paper, we present a tool called infoZilla that detects structural elements from bug reports with near perfect accuracy and allows us to extract them. We anticipate that infoZilla can be used to leverage data from bug reports at a different granularity level that can facilitate interesting research in the future.

Categories and Subject Descriptors: D.2.5 [*Software Engineering*]: Testing and Debugging—*diagnostics*; D.2.7 [*Software Engineering*]: Distribution, Maintenance, and Enhancement

General Terms: Management, Measurement

1. INTRODUCTION

Bug reports are vital for any software development. They allow customers to inform developers of the problems encountered while using a software. Bug reports typically contain a detailed description of the problem in natural language text, which is used by researchers to automatically assign developers [1] and locations [4], recognize bug duplicates [9], and predict correction effort [10].

Occasionally bug reports also hint at the location of the defect in form of stack traces, source code fragments, and patches that we together refer to as elements. Because such information is often embedded in the description, they are treated by all approaches mentioned above as natural language, although they should not be.

In this paper, we present infoZilla, a tool that detects and extracts such elements from bug reports (and the following discussions) to enable more prudent use of such information sources in bug reports. Having additional information such as *stack traces* and *source code* separated from natural language yields several advantages for research: it gives access to more and structured data, facilitates better training of machine learners, and allows research based on specific interests, e.g., only stack traces to automatically detect assignment of bug reports.

The paper begins with details about the implementation of infoZilla (Section 2) and then moves on to its evaluation that entailed manual inspection of 800 ECLIPSE bug reports (Section 3). We found that our tool comes with very high accuracy, precision, and recall, all very close to perfect. Thereafter, potential research applications of the tool are discussed (Section 4) and lastly, we conclude the paper with some threats to validity (Section 5) and our plans for future work (Section 6).

To our knowledge there is only little related work: Bird et al. describe how to detect patches in mailing lists [7] and Dekhtyar et al. discussed the opportunities and challenges for text mining [5].

2. METHODOLOGY

We implemented infoZilla, a tool that allows us to detect and extract a number of elements from bug reports and their discussions. We implemented several *filters* to detect and extract these elements. This section summarizes the filters and discusses some of the challenges we encountered while developing them.

2.1 Elements in Bug Reports

Bug reports often diverge in the quality and quantity of information provided on the encountered problem. A report commonly contains a detailed description of a failure. If the information given there is not sufficient, to point to the location of the fault, developers have the possibility to start a discussion on the report to resolve issues. Elements that often provide hints at a problem's cause, come in form of technical entities like stack traces or patches. As we have learned from previous work [2], developers are substantially interested in:

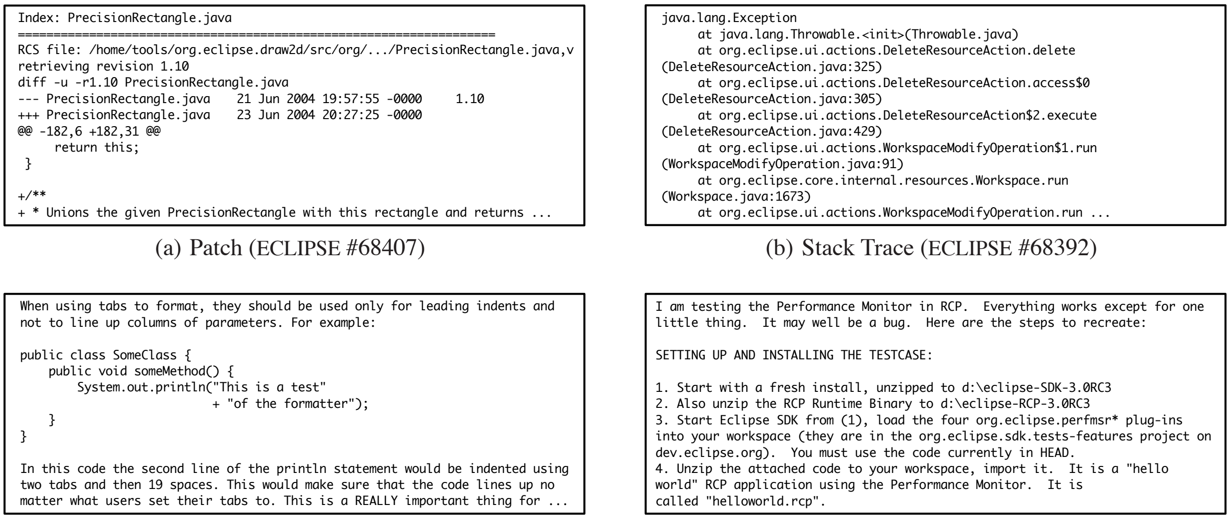
Patches. The common format of a patch is the uniform `diff` format. They represent a small piece of software designed to update or fix problems with a computer program or its supporting data.

Stack Traces. Reports of the active stack frames in the calling stack upon execution of a program are widely used to aid debugging by giving hints to the origin of problem.

Source Code. Small to medium sized code examples are used to illustrate a problem, describe the environment in which a problem occurred or even represent a sample fix to the problem described in the report.

Enumerations. They (and itemizations) are used to list items, describe a causality chain or a sequence of actions to take to reproduce or fix a problem. They add structure to the text of a report and ease reading and understanding it.

MSR'08, May 10-11, 2008, Leipzig, Germany.
Copyright 2008 ACM 978-1-60558-024-1/08/05 ...$5.00.



```
Index: PrecisionRectangle.java
===================================================================
RCS file: /home/tools/org.eclipse.draw2d/src/org/.../PrecisionRectangle.java,v
retrieving revision 1.10
diff -u -r1.10 PrecisionRectangle.java
--- PrecisionRectangle.java    21 Jun 2004 19:57:55 -0000    1.10
+++ PrecisionRectangle.java    23 Jun 2004 20:27:25 -0000
@@ -182,6 +182,31 @@
     return this;
 }

+/**
+ * Unions the given PrecisionRectangle with this rectangle and returns ...
```

(a) Patch (ECLIPSE #68407)

```
java.lang.Exception
     at java.lang.Throwable.<init>(Throwable.java)
     at org.eclipse.ui.actions.DeleteResourceAction.delete
(DeleteResourceAction.java:325)
     at org.eclipse.ui.actions.DeleteResourceAction.access$0
(DeleteResourceAction.java:305)
     at org.eclipse.ui.actions.DeleteResourceAction$2.execute
(DeleteResourceAction.java:429)
     at org.eclipse.ui.actions.WorkspaceModifyOperation$1.run
(WorkspaceModifyOperation.java:91)
     at org.eclipse.core.internal.resources.Workspace.run
(Workspace.java:1673)
     at org.eclipse.ui.actions.WorkspaceModifyOperation.run ...
```

(b) Stack Trace (ECLIPSE #68392)

```
When using tabs to format, they should be used only for leading indents and
not to line up columns of parameters. For example:

public class SomeClass {
    public void someMethod() {
        System.out.println("This is a test"
                           + "of the formatter");
    }
}

In this code the second line of the println statement would be indented using
two tabs and then 19 spaces. This would make sure that the code lines up no
matter what users set their tabs to. This is a REALLY important thing for ...
```

(c) Source Code (ECLIPSE #68369)

```
I am testing the Performance Monitor in RCP.  Everything works except for one
little thing.  It may well be a bug.  Here are the steps to recreate:

SETTING UP AND INSTALLING THE TESTCASE:

1. Start with a fresh install, unzipped to d:\eclipse-SDK-3.0RC3
2. Also unzip the RCP Runtime Binary to d:\eclipse-RCP-3.0RC3
3. Start Eclipse SDK from (1), load the four org.eclipse.perfmsr* plug-ins
into your workspace (they are in the org.eclipse.sdk.tests-features project on
dev.eclipse.org).  You must use the code currently in HEAD.
4. Unzip the attached code to your workspace, import it.  It is a "hello
world" RCP application using the Performance Monitor.  It is
called "helloworld.rcp".
```

(d) Enumeration (ECLIPSE #68361)

Figure 1: Samples of Elements in ECLIPSE reports.

2.2 Challenges

Several challenges pose themselves when detecting and extracting structural elements from bug reports. These elements, when copied into a bug reports, are often surrounded by comments, debug messages, annotations and similar natural language text. Furthermore, copying and pasting these elements into reports leaves room for many undesired consequences like accidental line breaks, premature ends or unanticipated edits. We identified and successfully tackled two most severe problems when trying to extract the elements from the reports:

1. **Line Breaks.** Elements like enumerations, patches or stack traces are closely knit in a line-wise design. Newline characters other than at the end of a line may break their formats and makes detection very difficult. We allowed for accidental line breaks whenever this added to the robustness of our algorithms without generating too many false positives. Without this enhancement, degenerated structural elements would either fail to be recognized by a filter or only be partially detected and extracted, causing a higher amount of false negatives.
2. **Unconfined Ends.** For all elements, it is hard to detect where the element ends and the remaining description starts again. In patches there are context lines, in source code there are comments, stack traces can contain natural language exception messages and enumerations can span across multiple paragraphs. For each filter we hand-tuned heuristics that consider paragraphs, line counts, whitespace and alike, to detect the endings of the elements. Without this enhancement, most filters would still be able detect specific elements, but cause the extraction to be incorrect, at worst extracting the complete remainder of the report.

2.3 Detailed Discussion of Filters

Patches Filter. Patches are an integral part of the software development process. They are used to update or fix issues in the software. Patches can either be binary, or text. Text patches are based on the uniform `diff` format [6]. A typical patch, as shown in Figure 1(a), has several well-defined parts:

- an optional patch *index* that identifies a patch.
- a description of the *original file* including date and time.
- a description of the *modified file* including date and time.
- at least one *patch hunk*, i.e., the section to be patched

Detecting and extracting patches is hence straight forward. We implemented a parser that searches line-by-line for a possible start of a patch, then parses every patch accounting for all parts. We implemented a number of heuristics that handle early line breaks and loose patch ends—two problems commonly observed with copied and pasted patches in bug reports [3].

Stack Traces Filter. A stack trace is a record of the execution of a software, showing the sequence of instructions executed up to an occurred crash. Figure 1(b) shows an example of a stack trace. Typically stack traces have several well-defined parts as well:

- the *exception, error or assertion* that has been violated.
- an optional exception- or error *message*.
- a *calling stack*.

Their well-defined composition can be leveraged for identification by use of regular expressions. We abstract a template for stack traces as follows:

```
( [MODIFIER]? [EXCEPTION] ) ([:]  [MESSAGE])?
( [at] [METHOD] [(] [FILE] [:]  [LINE] [)] )*
```

To successfully extract the stack traces, this template serves as a good starting point to model regular expressions. From this model, we derived a set of regular expressions, that identify parts of the bug report that contain the stack traces. Further expressions are then used to robustly split up the textual representation for extra detail.

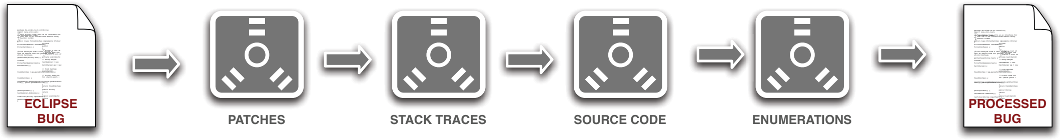


Figure 2: Chain of filters used for detection and extraction of structural elements.

Source Code Filter. Code fragments as shown in Figure 1(c), also called "snippets", are often used in bug reports to support the information given on a problem. These code fragments are often annotated or incomplete. They cannot be easily distinguished from the surrounding text by traditional approaches because neither do they conform to a parser's grammar definition, nor can they be compiled.

We base our filter on an island parsing approach [8]: initially, the complete input is treated as *water*. A fuzzy JAVA code parser then looks for a set of JAVA language constructs like *classes*, *functions*, *conditional statements* or *assignments* in the given input. These serve as starting points, or *islands*. We then explore the surrounding text of the islands and keep expanding the source code regions until no more code regions can be found.

The challenge is to get a good recall in the fuzzy parser, while minimizing false positives. Many JAVA language constructs resemble the English language, which adds to the ambiguity that the parser has to deal with. We overcome these issues by requiring the most ambiguous statements, such as assignments or function calls, to be the only content in the line.

Enumerations Filter. To detect and extract Enumerations, we implemented a parser that looks for occurrences of enumerations (1, 2, 3, . . .), line-by-line, that start with letters or numbers. By allowing additional symbols such as +, –, and $\star$, we easily extended the model to recognize itemizations.

Enumerations typically begin with numbers or symbols at the start of the line. These are often delimited from the content by stops (.) or brackets. Whenever a line confirming to this model is found, we add the content of the corresponding segment to the enumeration. This iteration continues until a symbol violates the requirement of incrementing numbers, or a paragraph ends and no more enumeration symbols are to be found.

Order of Filters. The order in which these filters are executed is important since some structural elements interfere. To cope with such interferences, we use the filter sequence presented in Figure 2: we first extract patches and stack traces from the reports because they are large but well-defined elements that can be extracted with least ambiguity. Then, we detect and extract code fragments and lastly, enumerations. This is because code snippets are often contained in enumerations. The serialization of the filters described, is necessary in our approach. Changing the application order of our filters will result in a large amount of false positives and false negatives. For examples enumerations and patches interfere with each other: + or – mark the start of lines for both, patches and enumerations.

3. EVALUATION OF THE TOOL

Next, we evaluate the performance of infoZilla on ECLIPSE bug reports. The focus is to correctly identify the presence of enumerations, patches, stack traces, and source code in bug reports. In this section, we present the setup for evaluation, followed by results.

Table 1: Description of Terms used in Evaluation.

Term	Short	Description
True Positive	TP	A report correctly classified to P.
True Negative	TN	A report correctly classified to $\overline{P}$.
False Positive	FP	A report wrongly classified to P.
False Negative	FN	A report wrongly classified to $\overline{P}$.

3.1 Setup of Experiments

We parsed 161,500 ECLIPSE bug reports to evaluate infoZilla. For each report, the tool verified the presence of the four elements; for each element, it classified the report in one of two bins: has (P) or has not ($\overline{P}$). To evaluate classification accuracy, from each bin, we randomly selected 100 bug reports to arrive at a total of 800 bug reports. Then, we manually inspected these reports to verify if they have been correctly classified, indicating the presence or absence of the respective elements. This gives the number of true positives, true negatives, false positives, and false negatives for each element (see Table 1), values that we used to calculate *accuracy*, *precision* and *recall* for the tool [11]. Thus, for every element, we calculate the following performance measures:

Accuracy relates the number of correct classifications to the total number of classifications.

$$Accuracy(x) = \frac{(TP_x + TN_x)}{(TP_x + FP_x + TN_x + FN_x)}$$

Precision relates the number of true positives to the total number of instances classified as positives.

$$Precision(x) = \frac{(TP_x)}{(TP_x + FP_x)}$$

Recall relates the number of true positives to the total number of true positives and false negatives.

$$Recall(x) = \frac{(TP_x)}{(TP_x + FN_x)}$$

3.2 Results of Experiments

Table 2 shows the values for TN, FN, FN and FP values we recorded during our evaluation for each element. We observe that all filters performed with remarkable accuracy. We discuss each the results for each element below:

1. **Patches.** infoZilla can detect Patches with perfect accuracy. This is partly because the format of Patches is well defined and hence, their beginnings and ends can be reliably traced. In ECLIPSE, we found a total of 147 patches.
2. **Stack Traces.** Stack Traces were detected with an accuracy of 98.5%. Unlike Patches, their formats are not so well-defined and hence, regular expressions were put to use for

Table 2: Results of Performance Measures.

	Percentage		
Element	Accuracy	Precision	Recall
Patches	100,00	100,00	100,00
Stack Traces	98,50	97,08	100,00
Source Code	98,50	98,00	98,88
Enumerations	97,00	99,00	95,19

their detection. We found a total of 13,997 stack traces in ECLIPSE. We did not find any false positives during the evaluation. False negatives resulted from bad formatting of the stack frames or traces interwoven with natural language text.

3. **Source Code.** The accuracy of detecting Source Code is similar to Stack Traces. We found a total of 19,229 instances of sample code or code snippets. False positives and negatives originated from the ambiguity of source code when trying to be distinguished from natural language text. Our method is a compromise between precision and recall.

4. **Enumerations.** They too were detected with a high accuracy of 97%. While enumerations come in many flavors, most often letters (a,b,c,...) or numbers (1,2,3,..) are used together with delimiter such as stops (.) or braces. We found 29,967 enumerations in ECLIPSE. False positives and false negatives originated from the fact that bug reporters are very creative in enumerating items. In our approach, we tried to cover the most conventional ones.

4. POTENTIAL APPLICATIONS

In the introduction, we claimed that infoZilla opens several avenues for research based on the extracted elements from the bug reports. To demonstrate this, we present three potential applications of the tool—research already near completion by the authors.

First, results from infoZilla can assist directing research towards special interest elements. One example is the patterns in crashing of methods observable in stack traces. An analysis of the patterns reveals the methods that crash most often and to which we must spend more testing efforts on.

Second, we are currently investigating the value of duplicate bug reports by examining the differences in their contents. The results reflect will upon whether duplicate bug reports should be viewed under favorable light by developers and be rather encouraged.

Lastly, we have developed a search facility better suited to bug databases. Developers can use it to find specific information related to bug reports, such as "*Find me all patches and stack traces related to Bug # 12345.*"

5. THREADS TO VALIDITY

We now briefly discuss possible threads to validity of our work. For each element, we randomly sampled 200 reports out of 161,500 in our database. This might not suffice to give a representative picture all possible failures that can occur while detecting elements. Also, we had to inspect a random sample manually leaving some room for human error. However, to reduce the likelihood of such error, we developed and used a graphical user interface for evaluation. Another persistent threat is the difference in opinions on what qualifies as structural element. For example, the presence of references to method calls in the report may or may not qualify as a code snippet.

6. CONCLUSION AND CONSEQUENCES

Bug reports typically comprise a problem description in natural language text and often, structural elements such as patches, stack traces and source code. Research to date using of bug reports have treated all contents as natural language text, but research can potentially benefit from treating such elements differently. We developed a tool, infoZilla, that extracts these elements from the reports with near perfect accuracy, as demonstrated by our evaluation of 800 ECLIPSE bug reports. Access to such piecewise elements from bug reports opens doors to several possibilities for research, for example, assignment of bug reports to developers and detection of duplicates, and more. Future work will focus on exploring several such research topics based on the extracted elements.

7. ACKNOWLEDGMENTS.

Many thanks to Sascha Just, Adrian Schröter, Cathrin Weiss, and Andreas Zeller for their valuable discussions. Also many thanks to the anonymous MSR reviewers for their helpful suggestions on an earlier revision of this paper.

8. REFERENCES

[1] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *ICSE '06: Proceeding of the 28th International Conference on Software Engineering*, pages 361–370, 2006.

[2] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann. Quality of bug reports in Eclipse. In *Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology eXchange (ETX)*, October 2007.

[3] C. Bird, A. Gourley, and P. Devanbu. Detecting patch submission and acceptance in oss projects. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, 2007.

[4] G. Canfora and L. Cerulo. Fine grained indexing of software repositories to support impact analysis. In *MSR '06: Proceedings of the 2006 International Workshop on Mining Software Repositories*, pages 105–111, 2006.

[5] A. Dekhtyar, J. H. Hayes, and T. Menzies. Text is software too. In *Proc. International Workshop on Mining Software Repositories (MSR)*, pages 22–26, Edinburgh, Scotland, UK, May 2004.

[6] Comparing and Merging Files. http://www.gnu.org/software/diffutils/manual/html_node/index.html. Last accessed 2008-01-16.

[7] J. H. Hayes, A. Dekhtyar, and S. Sundaram. Text mining for software engineering: how analyst feedback impacts final results. In *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*, 2005.

[8] L. Moonen. Generating robust parsers using island grammars. In *Proceedings of the 8th Working Conference on Reverse Engineering*, pages 13–22, Oct. 2001.

[9] P. Runeson, M. Alexandersson, and O. Nyholm. Detection of duplicate defect reports using natural language processing. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 499–510, 2007.

[10] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller. How long will it take to fix this bug? In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, 2007.

[11] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations.* Morgan Kaufmann, 2000.