Patterns for Extracting High Level Information from Bug Reports

Rodrigo Souza*†, Christina Chavez*‡ and Roberto Bittencourt*§
*Dept. of Computer Science, Federal University of Bahia, Brazil
†Data Processing Center, Federal University of Bahia, Brazil
‡Fraunhofer Project Center, Federal University of Bahia, Brazil
§Dept. of Exact Sciences, State University of Feira de Santana, Brazil
Email: rodrigo@dcc.ufba.br, flach@dcc.ufba.br, roberto@uefs.br

***Abstract*—Bug reports record tasks performed by users and developers while collaborating to resolve bugs. Such data can be transformed into higher level information that helps data scientists understand various aspects of the team's development process. In this paper, we present patterns that show, step by step, how to extract higher level information about software verification from bug report data.**

***Index Terms*—data analysis, mining software repositories, patterns, bugs.**

I. Introduction

Bug tracking systems record in their bug reports the collaboration between final users and developers in order to fix bugs. Such exchange of information can be help data scientists reason about the software development process.

Bug reports help compute many statistics related to quality and software development. For example, how many bugs are reported per day? What proportion of bugs are considered invalid? What is the average bug lifetime?

Even better, the raw data can be transformed into higher level information about developers and the software development process. With such information, one can infer developer roles, developers' workflow, software lifecycle phases, and so on.

In this paper, we present patterns to transform bug data into higher level information about the software verification process. Each pattern contains an *Examples* section with code snippets showing how to apply the pattern on real data. The snippets are written in R, a programming language for data analysis[1].

The next section presents the data set used in this paper. Section III presents the *Not Everyone is a Programmer* pattern, that helps discover quality engineers from bug reports. Section IV, *Testing Phase*, shows how to detect testing phases in the software development life cycle.

II. Data Set

The examples in this paper use bug reports from NetBeans/Platform and Eclipse/Platform, made available for the 2011 edition of the MSR Mining Challenge[2]. Both projects use Bugzilla[3] as their bug tracking system.

Bugzilla stores all modifications users make to bug reports, including changes in priority, status, resolution, or any other field in a bug report. In the examples, such data is available in the `changes` table, in which each row contains the `new value` of a `field` that was modified by a `user`[4] at some point in `time`.

In this paper, two kinds of change are explored: the change of `resolution` to `FIXED` (meaning that the bug was fixed by modifying the source code), and the change of `bug_status` to `VERIFIED` (meaning that the fix was considered appropriate by someone else).

All the data and code used in this paper is available online[5].

III. Fixers and Verifiers

Problem

Find the quality engineering team (if it exists).

Context

Developers tend to assume specific roles in the software development process. While many developers participate by fixing bugs, quality engineers usually take bug fixes and verify if they are appropriate. Making the distinction between quality engineers ("verifiers") and programmers who fix bugs ("fixers") is important when studying the influence of human factors on outcomes of the software development process.

Solution

To find members of the quality team, first analyze each developer's activity in the bug tracking system, such as status and resolution changes. In particular, count how many times each developer has...

- ... changed the status to `VERIFIED` (number of verifications);
- ... changed the resolution to `FIXED` (number of fixes).

[1]http://www.r-project.org/

[2]http://2011.msrconf.org/msr-challenge.html

[3]http://www.bugzilla.org/

[4]In this context, user denotes a user of the bug tracking system, which can be either a developer or a final user.

[5]https://github.com/rodrigorgs/dapse13-analysis

Then, compute the ratio between verifications and fixes for each developer (add 1 to the number of fixes to avoid division by zero). If such ratio is greater than some threshold (e.g., 5 or 10), it suggests that the developer is specialized in verifications. Select all such developers and compute the total number of verifications performed by them, compared to the total number of verifications in the project. If they perform a great part of the verifications in the project (e.g., more than 50%), then the project has a quality team, formed by that developers.

Choosing a suitable threshold for the ratio between verifications and fixes is a hard problem. If the threshold is too high, then only the most active quality engineers are chosen; if it is too low, then sporadic contributors, who have contributed with a few verifications, may also regarded as quality engineers.

One possible criterion is to choose a threshold that results in the smallest quality team that still contributes with large part of the verifications performed in the project. First, compute, for each candidate threshold (e.g., from 1 to 50), the size of the quality team and the number of verifications its members performed in the project. Then create a scatter plot of number of verifications vs. size of quality team (see Figure 1 in the *Examples* section). Finally, look for a point in the plot such as that increasing the size of the quality team does not significantly increase the number of verifications[6]. The value used to compute this point is the chosen threshold.

Discussion

It is a common mistake to use the absolute number of verifications to determine if a developer is a quality engineer. This is a poor indicator because, in some projects, developers that fix bugs also mark them as `VERIFIED`.

Developers can change roles over time. If this is the case, consider using sliding windows, i.e., analyze multiple consecutive short periods.

The solution to this pattern can be adapted to perform similar analyses on different types of change.

Examples

This solution was used by Souza and Chavez [1]. They chose a threshold of 10 for the ratio between verifications and fixes, but did not explain their choice.

In the following source code, we show how to apply this solution to NetBeans/Platform. First, compute the number of verifications and fixes for each user:

```
> resolution <- subset(changes, field=='resolution')
> status <- subset(changes, field == 'bug_status')
> t1 <- table(resolution$user, resolution$new.value)
> t2 <- table(status$user, status$new.value)
> user <- merge(as.data.frame.matrix(t1),
+            as.data.frame.matrix(t2),
+            by="row.names")
> user$ratio <- user$VERIFIED / (1 + user$FIXED)
```

[6]This is similar to the elbow criterion, used to find the optimal number of clusters in a data set.

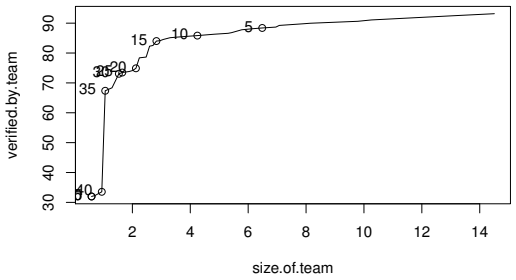

Fig. 1. Plot used to choose a threshold for the ratio between verifications and fixes. Labeled circles represent candidate thresholds.

Next, choose a threshold. To do that, first plot, for each number between 1 and 50, the relative number of verifications (%) vs. the relative size of quality team (%). The source code is omitted from this paper for space reasons, but can be found online (see Section II). The result is the plot in Figure 1, which show percentage values for the variables.

By visually inspecting the plot, we choose 15 as the threshold, because choosing a lower value increases the size of the team without significantly increasing the number of verifications. Choosing this threshold, the discovered quality team is formed by 24 members (2.8%), who contributed with 11310 verifications (84%).

Related Patterns

While this patterns helps identify people who concentrate quality efforts, the pattern *Testing Phase* (Section IV) helps find periods in which such efforts are concentrated.

IV. Testing Phase

Problem

Identify testing phases in the software development life cycle.

Context

Before a new version is shipped to final users, it is common to test new features and bug fixes. In some projects, most of the testing effort is concentrated on a well-defined testing phase, that precedes the release of the next version of the software.

In a bug tracking system, testing efforts are recorded as bug status changes, from `RESOLVED` to `VERIFIED`. Testing phases, therefore, show up as a relatively large number of verifications comprised in a relatively short period.

Failing to recognize testing phases mislead analyses. For example, if most bugs are verified during a testing phase, then measuring the time from `RESOLVED` to `VERIFIED` does not measure verification effort. Instead, it reflects how early a bug was resolved with respect to the next testing phase.

Solution

Solution 1. Select verifications, i.e., changes that set the bug status to `VERIFIED`. Then, plot the accumulated num-

ber of verifications over time using a line chart. If you know the software release dates, highlight them in the chart with vertical lines. Although the chart is monotonically increasing, some portions may exhibit a steeper ascent, that represents a period with high verification activity. Such periods probably are testing phases, particularly if they precede a release date.

Solution 2. Select verifications, i.e., changes that set the bug status to `VERIFIED`. Then, apply Kleinberg's algorithm [2] to verification times in order to detect bursts, i.e., periods of intense verification activity.

The algorithm is based on a Markov model and outputs a hierarchical burst structure. The first level comprises the entire period; the second level contains bursts in the period; the third level, bursts within second-level bursts, and so on. In the data we analyzed, second-level bursts spanned a few days, which seems right for a testing phase, while higher level bursts tended to span a few hours.

Discussion

The first solution is suitable for visual exploration of the data. If the data set is too large, however, it becomes difficult to visualize. The second solution is objective, though computationally expensive.

Be suspicious if the number of verifications per day is too high (e.g., above 50). Such verifications may be the result of a mass verification, when multiple bug reports are simultaneously updated in order to tidy the bug tracking system [3].

Some teams have dedicated quality engineers that are responsible for testing. Well-defined testing phases are less common in such teams, because quality engineers constantly test features and bug fixes, and therefore do not need to switch between programming and testing activities.

Examples

The first solution was used by Souza and Chavez [1] (see Figure 2 in their paper). The following R code shows how to apply the solution to Eclipse/Platform. Only a subset of the data is used, otherwise testing phases would be difficult to visualize. Assume `releases$date` is a vector with release dates.

```
> ver <- subset(changes,
+         field == "bug_status"
+     & new.value == "VERIFIED")
> ver <- ver[order(ver$time), ]
> ver$n.changes <- 1:nrow(ver)
> ver <- subset(ver,
+         time >= as.POSIXct("2009-06-10")
+     & time < as.POSIXct("2010-06-09"))
> with(ver, plot(n.changes ~ time, type="l"))
> abline(v=releases$date, lty=2)
```

The result is shown in Figure 2. Notice how verification activity (steep ascents) is concentrated just before release dates (dashed vertical lines), suggesting there are well-defined testing phases in Eclipse/Platform.

The following R code shows how to apply the second solution, using Kleinberg's algorithm and taking second-level bursts. Then, we count the number of verifications, total and per day, in each burst. The variable `ver` is reused from the previous snippet of code. The first 4 bursts are shown in Table I.

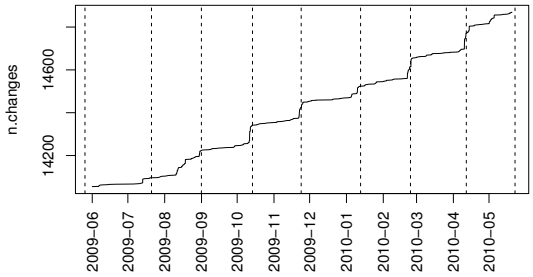

Fig. 2. Accumulated number of verifications over time.

TABLE I
PERIODS WITH INTENSE VERIFICATION ACTIVITY (SAMPLE).

start	end	count	per.day
2009-07-29 09:07:15	2009-07-29 16:21:58	17	17
2009-08-26 11:07:36	2009-09-03 12:24:38	71	7
2009-09-15 02:36:27	2009-09-16 12:54:35	26	13
2009-10-26 09:47:55	2009-10-29 12:00:26	82	20

```
> library(bursts)
> k <- kleinberg(unique(ver$time))
> bursts <- subset(k, level == 2)
> # Num. of verifications (total and per day average)
> bursts$count <- apply(bursts, 1, function(x)
+   sum(ver$time > as.POSIXct(x["start"])
+       & ver$time < as.POSIXct(x["end"])))
> days <- as.Date(bursts$end) - as.Date(bursts$start)
> days <- days + 1
> bursts$per.day <- bursts$count %/% as.numeric(days)
```

Related Patterns

Before applying this pattern, use the *Look Out For Mass Updates* pattern [3] to remove mass verifications from the data. Periods which include mass verifications can be confused with testing phases.

Use the *Fixers and Verifiers* pattern (Section III) to assess if the project has a quality team. The existence of such teams may explain the absence of a testing phase.

REFERENCES

[1] R. Souza and C. Chavez, "Characterizing verification of bug fixes in two open source IDEs." in *Proceedings of the 9th Working Conference on Mining Software Repositories*. IEEE, June 2012.

[2] J. Kleinberg, "Bursty and hierarchical structure in streams," in *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, ser. KDD '02. New York, NY, USA: ACM, 2002, pp. 91–101.

[3] R. Souza, C. Chavez, and R. Bittencourt, "Patterns for cleaning up bug data," in *Proceedings of the First Workshop on Data Analysis Patterns in Software Engineering*. IEEE, May 2013.