Tools and Methodology :

Does bug correction add complexity to the code when it isn't done by the former developer?

Phase 1: Identify the bug and the commits concerned

Tools used:
API Github

Detailed methodology:

1) We start by listing all the issues that were created on the repository. This can easily be done with the Github API

https://developer.github.com/v3/issues/#list-issues-for-a-repository
We just need to add a filter that let us retrieve only the issues with the label “bug”.

Result: We now have the list of all the issues that concern bugs and bugs correction for this repository.

2) We now look for who worked on debugging this feature, and which commits of the git repository are concerned by this debug.
In order to do this, we once again use the Github API, which allows to retrieve all kind of events for a given issue.
One of them is particularly interesting in our study: CommitCommentEvent
They give us the information about all the commits that are linked to the debug of this feature, and informations about their authors.
If we look at the first commit of this list, we can easily know the state of the Git when the debug was implemented (we just need to go to the commit N-1).

Result: We now have the commit we have to start our study from.
And also the list of the modifications and their authors made in order to debug this feature.

Phase 2: Identify the author / corrector situation

Tools used:
Git functionnalities and scripts

3) We are now looking to identify the current “author case” we are in, meaning:
- The author of the feature corrected his own bug
- An other developer corrected his feature

We can also extend the “other developer” by newcomer, by completing our methodology. This also leads to other questions like “From when can we consider that our developer is no longer a newcomer?”, that we won’t answer in this study.

In order to do this, we need to use a code comparison tool.
This can be done using different methods:
- We retrieve the concerned files and use gumtree between those files.
- We use git blame that also provide this functionnality, but directly on the concerned git.

We here chose the second solution because it was faster to obtain decent results.

To do that we need to clone the repository in our local storage, and then use the git blame command, but within the commits concerned by our correction.
This can be done with this command:

git rev-parse ^[FIRSTCOMMIT] [LASTCOMMIT] | git blame -S /dev/stdin [FILECONCERNED]

[FIRSTCOMMIT] : Needs to be replaced by the ID of the first commit event that we identified before.
[LASTCOMMIT] : Needs to be replaced by the ID of the last CommitEvent that we found
[FILECONCERNED] : Needs to be replaced with the file concerned by our study (all the files that were edited to correct this bug).

We can also edit the output buffer by editing the /dev/stdin part.

With this we know who edited which line, if the line starts with ^, it’s because it wasn’t edited during our bug corrections, if it was, we have the commit that edited this line, and it’s author.

We need to compare between the author of the initial line and the final line

Result: We now know the case we are in (Self Correction / Someone Else correction).

Phase 3: Analysis of the code quality

Tools used:
Shell scripts
Jacoco
Code-maat

PMD / checkstyle

A) Compare Cyclomatic Complexity

After a lot of research we found out two tools that could possibly match our requirements:
PMD that can be found here : http://pmd.sourceforge.net/pmd-4.3.0/running.html
and checkstyle here: http://checkstyle.sourceforge.net/cmdline.html

These are both command line tools that allows us to retrieve the cyclomatic complexity and other metrics.
Those are tools that are mostly used to detect problems within a project, so they might not be the most appropriate to just detect differences between two versions of a project (but could definitely be a plus if we want to calculate more code quality criterias).
We could for example use them in order to know if we still match our base criterias (Maximum complexity, maximum LoC, code coverage …) after the debug phase.

For checkstyle:

In order to retrieve the cyclomatic complexity with this tool , we need to generate a configuration file, and then add the cyclomatic complexity metric.
More informations about it can be found here:
http://checkstyle.sourceforge.net/config_metrics.html

For PMD:

In order to calculate the cyclomatic complexity, we can refer to this part of the documentation:
http://pmd.sourceforge.net/pmd-4.3.0/rules/codesize.html
We just ned to apply those rules to our configuration file in order to retrieve the results expected.

B) Compare Number of Lines of Code

Under linux, we can easily retrieve the number of lines in a file with the command:
wc -l [FILENAME]

We just need to do this for both files and store it.

C) Compare dependencies

Code-maat helps to calculate the logical coupling within the same application:
https://github.com/adamtornhill/code-maat

java -jar code-maat-0.9.0.jar -l logfile.log -c git -a coupling

We also need to compare coupling with external libraries.
One way to actually do this is to checkout if one of the commit adds an import line in the top lines of the file or edits the packaging system in order to add new dependencies.

D) Code Coverage

The fact that we had a bug that was not detected means that we potentially didn't test this feature enough.
A good point to explore in addition to the previous one is the code coverage for this feature after the debugging happened.

Adding test after the end of the debugging helps us to improve the quality of the software on a known problem.

Those tests can be done with jacoco which can generate reports about the code coverage in both xml and csv format.

The tool can be found here: http://www.eclemma.org/jacoco/

Identified Problems:

We found several problems in our methodology that we need to fix:

1. We currently make the assumption that the only edits done to a file between the initial commit and the last debug commit are only bug correction, which is not the case.

In order to improve this element, we should develop a process that would need to check each commit, save the line edited, and then use one of the 2 solutions:

a) Start from the pre-bug version of the file and apply the corrections to it, and then calculate all the things we wanted (with this solution, all other edits on the file would be rollbacked so we don't consider them when analyzing our code).
b) Find a way to analyse only the part of the code we want to (Calculate lines added/removed only in the bug correction sections / calculate code complexity only for the methods concerned …)

2. We currently Consider that there is only one person correcting a bug, but how should we consider our study when multiple persons are working on a bug correction?

3. We have a large amount of interactions with external API's and command line tools, so we need to synchronize everything through a script that also has to implement a large amount of parser in order to retrieve only the needed informations.
Moreover, we are here explaining the command chain for a single bug, but in order to obtain some results we need to apply this to all the bugs of the repository and then generate some statistics with our results.

Special Mentions:

JArchitect is mentioned in many topics about tools that could help us on the last part, and looks like a great tool that is a fusion of all of those used in the third phase of our study. However this tool isn't free to use, so you may want to use it if you already have a license or if you can afford it.

Author:

Loïc Potages