Bug Insertion

Gema Rodríguez Pérez

Abstract

The objective of this paper is to prove that a bug is not caused by the previous commit. Currently, the premise establishes that all bugs were introduced by a previous commit. We have carried out an experiment which has helped us show results pointing out a minor disagreement with the present premise. We have observed three stages totally separable throughout the execution of this experiment. The first stage considers a classification, based on our knowledge of the ticket we have analysed, in three groups: 1.It is an error. 2.It is not an error. 3.Unknown. The second stage involves the analysis of it as an error. In this stage where, with some parameters of interest, we are able to give a percentage of how many prior commits were responsible for the bug. The third stage is the implementation of an analytical tool to help us with the first and second stages. In this paper we focus in two first stages. Explaining clearly and concisely the methodology used in our experiment.

1 The experiment

Our experiment consisted of the analysis of one hundred tickets which were taken randomly from Cinder repository. We will call the first stage 'the filtering' as we will classify them into three categories:

1. It is an error.
2. It is not an error.
3. Unknown

Two different people with knowledge of programming were responsible for classifying these tickets in accordance with their opinion of the three categories mentioned above. These two people worked in parallel during the filtering. Only when all the tickets were classified were both classifications compared. Those whose classification were different, were compared with each other, thus reaching an agreed classification.

After filtering, we continued to the next stage, in which we only kept in mind those tickets belonging to 'It is an error' group. In this phase we were able to conclude whether the bug was added in an earlier commit, or by contrast, if it was not. This stage is still being validated on account of a shortage of tickets analyzed. At this moment, we can design an efficient algorithm that involves relevant parameters to reach the conclusion that we are looking for. So, for the moment, we are using these parameters as a support.

2 Several questions we have to answer

To provide a context, we have to answer some questions referring to the project from which we have taken the information. Likewise, we are going to describe the specific vocabulary.

- What is OpenStack?

  It is a combination of software tools for building and managing cloud computing platforms for public and private clouds. Principally, users deploy it as an infrastructure as a service (IaaS) solution. The technology consists of many different moving parts. In particular, OpenStack has nine key components that can be identified as part of the 'core'. Officially, OpenStack community maintained these system.

  - Compute (Nova) is the primary engine of an IaaS system. It is used for deploying and managing virtual machines.
  - Object Storage (Swift) is a scalable redundant storage system for objects and files.
  - Block Storage (Cinder) manages the creation, attaching and detaching of the block devices to servers, being able to access specific locations on a disk drive.
  - Networking (Neutron) provides the networking capability for OpenStack managing IP addresses.
  - Dashboard (Horizon) is the only graphical interface to OpenStack whereby administrators gain graphical access.
  - Identity (Keystone) provides a central directory of users mapped to the OpenStack services that administrators and users can access.
  - Image Service (Glance) provides virtual copies of services to OpenStack. It allows use of these images as templates when deploying new virtual machine instances.
  - Telemetry Service (Ceilometer)
  - Orchestration (Heat) helps to manage the infrastructure needed for a cloud service to run through both a REST API and a Query API.

  Each one of the above has its own API to achieve integration. Furthermore, all of them have a repository where the community implements improvements; fixing bugs ... etc.

- How can OpenStack help us?

  OpenStack is is of an open nature, anyone can add additional components to OpenStack to help it meet their needs. Because of the high scope of OpenStack we have at our disposal several bugs that we are interested in. The first of which is in Cinder.

- What is Cinder in OpenStack?

  Cinder is a component of OpenStack, it works as a Block Storage manipulating volumes and snapshots. Cinder has a Data Base in which the state of each volume can be found. At the same time, Cinder is composed of:

  - Cinder API: Accepts orders and routes them to cinder volume.
  - Cinder Volume: Reacts to these requests, writing or reading the database. Furthermore, it interacts with other processes such as cinder Scheduler through the message queue. Also acts directly upon storage, providing hardware or software.
  - Cinder Scheduler: Selects the node to storage and volume where created.

- What would we like to achieve by analyzing this repository?

  In analyzing Cinder repository we would like obtain a percentage of how many errors have been inserted by the previous commit owing to the fact that the actual premise assumes that the error was introduced by previous commit. It is important to be able to demonstrate that existing others factors contribute to the appearance of these errors.

- What do we have to analyze in this repository

  We need to be able to analyze a set of true tickets which will help us to reach the desired conclusion. Firstly, we have chosen a repository from Cinder, where we can find a wide group of tickets.These tickets have a state of progress, it is an updating of its status from opening the ticket until closing. We only are interested in a closed ticket and its release patch, that is, whose status is 'Fix released' or 'Fix Committed'.

  Before reaching the goal, we have to answer intermediate questions which provide us with information throughout the analysis. First of all, we must be clear in what we will analyze and where we can find the necessary data for this analysis. So, the first questions are:

- What is a ticket?

  We call a ticket each of the entries that are published in `https://bugs.launchpad.net/cinder`, describing a bug. The purpose is to leave a written a evolution of the bug from its opening until it is fixed.

2.1 First Stage

- Where will we find a ticket?

  The ticket are found in `https://bugs.launchpad.net/cinder`, where we must do an advanced search with a main search parameter called 'status' as:

  - Fix Committed: the changes are pending, and to be uploaded soon.
  - Fix Released: a fix was uploaded to an official Cinder repository.

  The next stage is to choose a random ticket and analyze it. Trying to answer the following questions until arriving at our objective. To answer the main questions, from the first level; we need to ask questions at the second level. The main questions are:

  - Is it a Bug?
  - Can we identify the commit responsible for the bug?
  - Is the previous commit responsible?

  As we have mentioned before, to answer the main questions we need to answer other questions like the following:

- What do the title, the description and the commits of a ticket represent?

  Once determined, we choose a ticket, we must to read carefully the title and the description of the ticket as well as the title of the commit which fixed the patch, because this is where will obtain more information.

  The title and the description of the ticket whose id is 'n_ticket' can be found in `https://bugs.launchpad.net/cinder/+bug/n_ticket`. The title of the commit can, sometimes be found in the same link under the comment 'Fix merged to cinder (master)', where we can find link leads to the review of the ticket `https://review.openstack.org/#/c/n_review`. In the review all the information about the patch that fixed the bug is shown.

  After analyzing these three fields, we should note the keywords of each of them, because in our opinion, they will help us in the future to develop an algorithm. Understanding keywords as those words that show the same pattern or are specific will help us with

the classification. In addition, very often (We must calculate the percentage from the experiment), the description helps us to answer the question: Is it a bug?.

At this stage, we are almost ready to answer the first question, Is it a bug?, Before that, we must be clear about the criteria that we will use.

- What criteria will be used?

  We need to define certain criteria that must be followed in case of doubt about it being a bug or not. We have based the answer on the following criteria:

  - Are there only test files? It is not an error. Our criteria indicate that test files will not be analyzed because we consider these files indispensable when a feature is implemented. With a test file the working order of the program is checked, in a way to give consistency to the program. A bug in the code, implicates there being a bug in the test file, provided that the test are implemented. So, we are not interested in these files.
  - Does the title of the ticket describe a new feature? It is not an error. New features are not considered errors, because there is no failure. The optimization, deletion of a dead code or new characteristics to users are involved here.
  - Does the title of the ticket describe the program as not working as expected? It is an error. Sometimes the error is not in a main function which prevents Cinder from working, because a developer has considered it an important error, although not relatively important. But it is not working as he/she expected.
  - Does the title of the ticket describe that updates are required? It is an error. We consider all tickets that require updating as errors, because if it is not updating cinder is not operating as expected and will cause errors.

  Sometimes we are unable to answer all the questions due to having insufficient data or because of the complexity of the issue.

  At this time, we should be able to classify the tickets according to their nature as:

  1. It is an error.
  2. It is not an error.
  3. Unknown

  In the experiment, we only have focused in the 'It is an error' group, as this is the ticket group that interests us.

  Now the more difficult part begins. Commits will be focused on, having to identify which lines have been removed, modified or added in each of them. Also we should know in which files are the answers to the question about who is responsible.

  As before, to answer these questions we need to answer some questions from lower levels:

  Sometimes, after having read the description of the ticket, we know if the ticket is an upgrade error. In this case, the answer to the question 'Is there a bug responsible?' is negative. That is, according to our criteria nobody is responsible for the evolution of the code. Therefore, if an update is necessary, despite being an error there is no commit responsible for the bug.

  In other cases, after the description we can conclude that we are unable to identify a responsible commit, because the ticket includes more than one component within OpenStack

and although it has been opened in Cinder could have been closed in another repository that belongs to another component of OpenStack. Therefore we do not have a review where we analyze the code and identify the commit responsible.

If the ticket is not in any of the above situations, we must be able to answer the next questions by analyzing the code of the files. These codes can be found in the review page of each ticket. If you click on the file involved, a comparison between the status of the file before commit and the status of the file after the commit appears.

- What kind of code is it? We assume there are three kinds of codes depending on the character of the lines, which are on the involved files.

  1. New Code: When every change is executed, lines were added to each of the files.
  2. Deleted Code: When every change was executed, there were deleted lines in each of the files.
  3. Modified Code: When the previous two codes are combined. This is the most common type of code (Write a percentage).

  Being able to classify the involved code helps us to obtain a first prediction regarding the difficulty in finding the responsible commit. According to our experience analyzing these 100 tickets, the modified code and the deleted code are better than the new code because we have a limited reference as to where the application was failing.

  (The following questions arise; Should I write about the problems of analyzing the modified code? For example: the order of the functions can be changed between the previous commit and the actual commit, when in fact, there is no change. Should I write about the problems of analyzing a new code? For example: Is the new code added, part of a previous function or is it only a new feature?)

- What number of commits are involved? Using 'git blame' command in our local cinder repository, we are told what revision and author last modified each line of a file. We only focus on the involved lines of each implicated file, so after executing this command we get a list with all the previous commits, which could be the responsible for the bug.

  At times, we can discard some commits belonging to the previous list due to the commit only adding a blank line, added or modified commentaries, or maybe, it adds to the actual number of the version file.

- What will happen if the number of folders is greater than one? Ideally a bug is involved in a single file. But unfortunately it is not always true, therefore, for these cases we will analyze all files globally as if they were one. So, we could have more than one responsible commit of the bug.

  We must pay attention when several consecutive lines were modified in a file, it can happen that they were cut and pasted. Even during the previous commits, this would have been happening and therefore the previous commit was not directly responsible for the error. If not that, it could have come from a previous commit and because of cutting and pasting it has been transferred from to commit to commit.

- Is it only the previous commit that is involved? In most cases, if we find a commit that acts along the same lines that have fixed the error, it is a responsible commit. Although there are exceptions, for example, we can find 'forks' commits, i.e. a code that was copied from another repository. These new files will have the same initial commit which is not responsible for the previous errors.

We must keep in mind the initial description, because it has usually explained what is happening and together with the code and the commit of the patch, we can understand how it has been solved. On occasion, as when a feature is required, we may conclude that the commit involved is not responsible for the error. The code evolves, so although we find commits involved in these lines, we must review the description should a case like this occur.

2.2 Second Stage

it is now that we continue with the next stage, where we can obtain a percentage of how many bugs have been introduced by a commit. To understand the second stage, we will explain some fields, important to us, which have been extracted from a review of the ticket. There could be more, but these are the most representative for us:

We use the 'git diff commit1 commit2' command to calculate the next parameters:

1. Number of added lines: Are the number of lines in a file ,which have been added in the actual commit (commit2) with respect to the previous commit (commit1).
2. Number of deleted lines: Are the number of lines in a file, which have been deleted in the actual commit with respect to the previous commit.
3. Number of modified lines: Are the sum between the number of added lines and the number of deleted lines.

We use the 'git blame file' command to calculate, manually at the moment.

1. Number of previous commits: The total number of commits appearing in the lines involved.
2. Is each of the identifiers of each of the above commits involved in the bug.

Now, with all the information that we have extracted, we are be able to answer the last question and our objetive for that investigation.

- Is this the commit responsible? In the same file we can find several commits involved, but not everyone has to be responsible for the commit. Therefore, comparing the status of the file, before commit and after commit, we are be able to identify those commits that are responsible and those that are not. Therefore, we know the degree of involvement of each of the commits involved if we classify them according to their responsibility in:

  1. It is responsible for the bug
  2. It is not responsible for the bug
  3. Unknown responsible for the bug

  Finally, we can start with the analysis of the result obtained in this experiment.

3 The results

3.1 Keywords

3.2 Percentage of the first classification

3.3 Percentage of the fsecond classification

4 Automation

5 Conclusions