Examining the effects of ownership on software quality

Apache Lucene - A study case

The purpose of this work is to replicate the study (http://dl.acm.org/citation.cfm?doid=2025113.2025119) done by Bird et al., published in FSE'11. To replicate the work, we have selected Lucene (https://lucene.apache.org/core/), a search engine written in java. Our goal is to see the results of a similar investigation on an OSS system.

Data collection

The first step for performing the analysis is to analyse the data and create a **table** with the relevant information (i.e. *data collection*).

In this study, we intend to relate certain ownership metrics and post-release bugs. The project under analysis was mainly developed in Java, which means that, most of the time, classes will be contained in a single file and each file will contain a single class. For this reason, the relation was analysed for each file in the project.

This implies that our table will contain one row per file and a column per each ownership metric for that file, with one additional column containing the number of identified post release bugs.

Choosing what to analyse

For defining a scope for the analysis, we opted to choose a release, compute the ownership metrics for all the files present in that release and determine which ones contained bugs.

In order to choose a release to analyse, we went through a complete list of all Lucene issues until August, 2015. We then determined which of these issues identified bugs that had been fixed and chose to analyse the release version with the most fixed bugs.

In order to analyse the files, we imported several python modules for handling files (and json files in particular) and set a path to our folder containing the issue files.

```
In [56]: import json
         import time
         from os import listdir
         from os.path import isfile, join

         ISSUES_PATH = "issue_LUCENE"
```

If we haven't the issues folder already we are going to download it. For this we will need the sh and os modules.

```
In [57]: import sh
         import os

         if not os.path.exists(ISSUES_PATH):
             sh.wget("-O", "issue_LUCENE.tar.bz2",
                     "https://drive.google.com/uc?id=0BzuWZdqy9QYwUHN6bThDU2
         VFN1k&export=download") # download package
             sh.tar("-jxvf", "issue_LUCENE.tar.bz2") # extract package
             sh.rm("issue_LUCENE.tar.bz2") # remove package
             print "Downloaded the issues successfully."
         else:
             print "Issues folder already exists."
```

```
Downloaded the issues successfully.
```

After this, we retrieve all json files in the specified directory.

```
In [58]: onlyfiles = [ f for f in listdir(ISSUES_PATH) if isfile(join(ISSUES
         _PATH,f)) and f.endswith(".json") ]
```

And we analyse all of them to determine which ones identify closed bugs with a *Fixed* resolution status, storing the bugs in a dictionary that maps from release version to bugs that affect this release. For convenience, we also store the resolution dates in a separate dictonary that maps from bugs to dates in order to later determine the most recent commit of the ones we will need to analyse.

In [59]:
```
version_bugs = {}
bug_fixed_dates = {}

for f in onlyfiles:
    with open(join(ISSUES_PATH,f)) as data_file:
        data = json.load(data_file)
        if (data["fields"]["issuetype"]["name"] == "Bug" and
            data["fields"]["status"]["name"] == "Closed" and
            data["fields"]["resolution"]["name"] == "Fixed"):

            bug_fixed_dates[data["key"]] = data["fields"]["resoluti
ondate"]

            versions = data["fields"]["versions"]
            for v in versions:
                if version_bugs.has_key(v["name"]):
                    version_bugs[v["name"]].append(data["key"])
                else:
                    version_bugs[v["name"]] = [data["key"]]
```

After generating the dictionary, we iterate through it to determine which version has the most bugs and store the list of bugs in another dictonary which maps to an array which will contain the list of files affected by the bug.

In [60]:
```
most_bugs = 0
buggy_version = ""

for k,v in version_bugs.iteritems():
    if len(v) > most_bugs:
        most_bugs = len(v)
        buggy_version = k

bugs = {}

for b in version_bugs[buggy_version]:
    bugs[b] = []

print "Version with most closed bugs:", buggy_version , "with", len
(version_bugs[buggy_version]), "bugs."
```

```
Version with most closed bugs: 4.0-ALPHA with 179 bugs.
```

The next step is finding the date of the most recent bug fix that affects the chosen version.

```
In [61]: latest_date = "1999"   # date for the most recent bug fix

         for b in bugs:
             latest_date = (bug_fixed_dates[b] if bug_fixed_dates[b] > lates
         t_date else latest_date)

         latest_date = latest_date[:10]

         print "The day of the most recent bug fix is", latest_date
```

```
The day of the most recent bug fix is 2014-01-24
```

Extracting the ownership metrics

Now that we know which files and bugs we want to analyse, we can start going through repository data to extract the relevant metrics.

Firstly, we must first clone the repository and checkout the version we chose to analyse (which we previously determined to be version 4.0-ALPHA). We import the re module which will be useful for extracting information from commits through regular expressions.

```
In [62]: import re

         if not os.path.exists("lucene-solr"):
             sh.git.clone("https://github.com/apache/lucene-solr.git")

         git = sh.git.bake(_cwd='lucene-solr') #specify repository directory

         git.checkout("tags/lucene_solr_4_0_0_ALPHA")

         print "Checkout of tag lucene_solr_4_0_0_ALPHA."
```

```
Checkout of tag lucene_solr_4_0_0_ALPHA.
```

We can now extract a list of files in the repository with the git ls-files command. We are only interested in java files so we only want files ending in ".java".

```
In [63]: files = [ f for f in git("ls-files").split("\n") if f.endswith(".ja
         va") ]
         print "We found", len(files), "java files."
```

```
We found 3851 java files.
```

With the list of files, we extract information relative to the contributors of each one using the git log command to retrieve the list of contributors. With the output generated by this command it is then necessary to sort and count the occorrunces of each contributor. To achieve this, the *sort* and *uniq* shell commands were used.

Knowing how many contributions each contributor did, it is then possible to determine the ownership metrics for each file relating to:

- **total**: total number of contributors
- **major**: number of major contributors (5% or more of total contributions)
- **minor**: number of minor contributors (less than 5% of total contributions)
- **ownership**: maximum ownership of a contributor over the file

In [64]:
```
print "Reading repository...",
start = time.time()

table = {}

for f in files:
    contributors_data = filter(None, sh.uniq(sh.sort(git.log("--for
mat=format:%an", f)), "-c").split("\n"))
    contributors = []
    total = 0
    max_ownership = 0
    minor = 0
    major = 0

    for a in contributors_data:
        num = int(re.search("[0-9]+", a).group(0))
        name = re.search("([A-z]+\s*)+", a).group(0)
        total += num
        max_ownership = num if num > max_ownership else max_ownersh
ip
        contributors.append((name,num))

    for a in contributors:

        if a[1] * 1.0 / total >= 0.05:
            major += 1
        else:
            minor += 1

    table[f]= {"minor": minor,
               "major": major,
               "total": minor + major,
               "ownership": (float("{0:.2f}".format(max_ownership *
1.0 / total * 100))),
               "num_of_bugs":0}

print "...finished reading", time.strftime('in %M minutes and %S se
conds.', time.gmtime(time.time() - start))
```

```
Reading repository... ...finished reading in 10 minutes and 50 sec
onds.
```

Finding commits that fix bugs of the chosen release

Now all that is left is to find how many bugs each file has by analysing the commits that fix each bug.

Using the latest date we found when obtaining the bug list from the issue files, we first determine the latest commit that fixes a bug for this version.

```
In [65]: shaLatest = (git("rev-list","-n 1","--before=\"" + latest_date + "2
         3:59\"","trunk")).stdout[:-1]
         print "Hash of the last commit before latest bug fix:", shaLatest
```

```
Hash of the last commit before latest bug fix: ba560c7484c1df260ae
78b414749fa81af998231
```

We now retrieve the full list of commits that we need to analyse for finding bug fixes.

```
In [66]: commit_list = filter(None, git("rev-list", "--topo-order", "HEAD.."
         + shaLatest).split("\n"))
         print "We have", len(commit_list), "commits to analyse."
```

```
We have 7342 commits to analyse.
```

We search for commits that contain references to Lucene issues and check if we have a bug with the same id. If there is a match, we retrieve the list of files modified by that commit and store them in our *bugs-to-affected_files* dictionary.

```
In [67]: start1 = time.time()

         for commit in commit_list:
             message = str(git.log("--format=%B", "-n 1", commit))
             match = re.search("LUCENE-[0-9]+", message)

             if match:
                 key = match.group(0).strip()
                 if key in bugs:
                     files_changed = filter(None, git("diff-tree", "--no-com
         mit-id", "--name-only", "-r", commit).split("\n"))
                     bugs[key] = files_changed

         print "Identified fixes for", len ({k for (k,v) in bugs.iteritems()
         if len(v) > 0}) ,"out of", str(len(bugs)),
         print time.strftime('in %M minutes and %S seconds.', time.gmtime(ti
         me.time() - start1))
```

```
Identified fixes for 32 out of 179 in 03 minutes and 35 seconds.
```

We can now update our table with information relating to the number of bugs by incrementing the variable for each file that was changed in a bugfix commit.

```
In [68]: for key, value in bugs.iteritems():
             if len(value) > 0:
                 for file in value:
                     if file in table:
                         table[file]["num_of_bugs"] += 1
```

Write CSV

Finally, all that is left to finish the data collection phase is generating a csv file with all the relevant information. A *data.csv* file is created and the table we generated is written with each entry of the table corresponding to a line in the file.

In [69]:
```
f = open("data.csv", "w")
f.write("file_name, minor, major, total, ownership, num_of_bugs\n")

for k,v in table.iteritems():
    f.write(k + "," + str(v["minor"]) + ", " + str(v["major"]) + ",
" + str(v["total"]) +
            ", " + str(v["ownership"]) + "%, " + str(v["num_of_bugs
"]) +"\n")

f.close()

print "CSV file saved as data.csv."
```

```
CSV file saved as data.csv.
```