2013 20th Asia-Pacific Software Engineering Conference

Mining GitHub: Why Commit Stops

Exploring the Relationship between Developer's Commit Pattern and File Version Evolution

Yang Weicheng, Shen Beijun, Xu Ben
School of Software Engineering
Shanghai Jiao Tong University
Shanghai, China
maxi900201@gmail.com, bjshen@sjtu.edu.cn

***Abstract*—Using the freeware in GitHub, we are often confused by a phenomenon: the new version of GitHub freeware usually was released in an indefinite frequency; and developers often committed nothing for a long time. This evolution phenomenon interferes with our own development plan and architecture design. Why do these updates happen at that time? Can we predict GitHub software version evolution by developers' activities? This paper aims to explore the developer commit patterns in GitHub, and try to mine the relationship between these patterns (if exists) and code evolution. First, we define four metrics to measure commit activity and code evolution: the changes in each commit; the time between two commits; the author of each changes; and the source code dependency. Then, we adopt visualization techniques to explore developers' commit activity and code evolution. Visual techniques are used to describe the progress of the given project and the authors' contributions. To analyze the commit logs in GitHub software repository automatically, Commits Analysis Tool (CAT) is designed and implemented. Finally, eight open source projects in GitHub are analyzed using CAT, and we find that: 1) the file changes in the previous versions may affect the file depend on it in the next version; 2) the average days around "huge commit" is 3 times of that around normal commit. Using these two patterns and developer's commit model, we can predict when his next commit comes and which file may be changed in that commit. Such information is valuable for project planning of both GitHub projects and other projects which use GitHub freeware to develop software.**

Keywords-repository mining; GitHub; commit pattern; version evolution; visualization technology

I. INTRODUCTION AND MOTIVATION

GitHub is a relatively new project hosting site, which, due to its developer-friendly features, has enjoyed widespread acclaim and adoption. It provides freeware for study, research and other uses. In our lab, we usually use these freeware to develop our own software.

However, during the development process, our plan and design are often interfered by sudden updates of files and APIs. New version of freeware usually is released in an indefinite frequency; and developers often do not commit anything for a long time. This is a common phenomenon in GitHub, and therefore we have to change our plan and design frequently to adapt the changes from the freeware.

To find the root cause of this phenomenon, this paper aims at exploring the relation between developer commit patterns and file version evolution by the following three research questions:

RQ1. How do the changes affect the other source files?

When the code changes in the previous version, which source files will be modified in the following versions? The influence of the changes will help to locate the source files which need to be updated in the following versions, and to describe the reason of these updates.

RQ2. What are the developers' commit habits look like?

In GitHub, developer's behavior consist of commit activities and interval between two commits. The developer's commit habit, a pattern that can model the developer's behavior, will show the frequency of the commit activities and the change scales via the time line. Different developers may have different habits. Can there special patterns describe the core developers' commit habits?

RQ3. What's the relationship between core developers' commit habits and the file version evolution?

The core developer dominates the software development. In the file version evolution, what are the effects of the core developers' behaviors? The relationship between core developers' commit habits and the file version evolution will describe when the new version will come and which files will be changed.

The rest of the paper is organized as following. Section II proposes four metrics to explore the developer's commit habit and the file version evolution. Section III describes the design of a software repository mining tool which calculates and visualizes the four metrics using class dependency graphs and barcodes. Section IV conducts case studies. Finally, Section V concludes the paper and outlines future work.

II. MEASURES

According to the three research questions, we design four metrics to measure commit activity and software evolution.

A. Changes in each commit

Changes in each commit describe the change scale evolution of the software. To measure the changes, the concept "sub-version" is introduced. One software version is divided into several sub-versions, and each sub-version consists of several revisions. The introduction of sub-version will reduce the workload of code changes calculating without sacrificing the confidence of the result.

The classis and simple measure of changes in line of code (LOC) modified. In our study, we divide changes into two types, “add” and “delete”, to build the code change metrics. So in one sub-version, each source file has two values, added LOC and deleted LOC. These changed lines can be extracted from the log entries.

B. *Interval between two commits*

Interval between two commits describe the change frequency of the software. In order to measure the interval, we need to extract the time strings from the commit log entries.

We define three interval values: interval between current commit and the latest previous commit, interval between current commit and the latest previous “huge commit” and interval between current commit and the latest previous milestone.

C. *Author of each changes*

The authors of each changes help analyze the developers’ commit pattern. In one commit, there is only one author, but in the sub-version, there are more than one developer. These developers are divided into groups, and then additional information is used to measure the features of these groups.

D. *Source code dependency*

Source code dependency describes the structure of the whole project in the given period of time. The dependency can be described by two parts: static source code dependency and dynamic code change dependency.

There are many ways to measure the static source code dependency, including static source code analysis and dynamic program analysis. Function call metrics are chosen as the key of static source code analysis in order to build a dependent sequence. For example, assume that file A has function f1, and file B has function f2. A function call from f1 to f2 indicates a dependency from A to B (A->B).

The second part, dynamic code change dependency, will be built from the commit logs. The basic idea of dynamic code change dependency is that, changes in file B is caused by changes in file A, then file B depends on file A. But in GitHub, the commits are often discontinues. Using sub-version instead of revision will help find out the potential relationship between two changes.

Using these metrics, barcode visualization is adopt to explore the developers’ commit habits. The barcode consists of several vertical parallel line segments of equal length. These segments are sequentially arranged from left to right in accordance with time. The width of a line segment represents one or more commits on a certain day from one developer, and the change scales of these commits. The gaps between two line segments are the periods with no commits.

We have already defined the inert period in [1] as the maximum gap of developer’s commit history. In this study, we use the distribution of the inert periods and the commits in the barcode visualization to describe the developers’ commit habits. We visualize every developer’s commit history on each package as a barcode graph, and also divide developers into core members and peripheral members to see the different distribution styles.

III. TOOL DESIGN

To analyze the software development data in GitHub automatically, we develop a tool named Commits Analysis Tool (CAT). It contains four main modules: dependent sequence generation module, data simplification module, dependency directed graph visualization module and barcode generation module. Each module will be described in the following sub-sections.

A. *Dependent sequence generation*

ArgoUML, an object oriented reverse engineering tool, is used to generate the static source code dependent sequence from the given sub-version source code. However, some relationships between source files are missing, so PowerDesigner, another UML modeling tool, is used to make a complement to one xml file that describes the dependent sequence like Fig.1.

```
<UML:Class xmi.id = '-64--88-68-1-
66f6e0a6:13a812327e2:-8000:00000000000ED3B2'
name = 'Boolean' visibility = 'public' isSpecification =
'false' isRoot = 'false' isLeaf = 'false' isAbstract = 'false'
isActive = 'false'>
<UML:ModelElement.taggedValue>
<UML:TaggedValue xmi.id = '-64--88-68-1-
66f6e0a6:13a812327e2:-8000:00000000000ED3B3'
isSpecification = 'false'>
<UML:TaggedValue.dataValue>yes</UML:TaggedValu
e.dataValue>
<UML:TaggedValue.type>
<UML:TagDefinition xmi.idref = '-64--88-68-1-
66f6e0a6:13a812327e2:-8000:00000000000E7488'/>
</UML:TaggedValue.type>
</UML:TaggedValue>
</UML:ModelElement.taggedValue>
</UML:Class>
```

Figure 1. Simplified commit detail modifications

$$V_{xy} = \alpha \times \frac{T_{lcc}}{T_{lt}} + \beta \times \frac{T_{cc}}{T_t} \qquad (1)$$

In the dynamic code change dependency analysis, there are two dependency value of each two files, V_{xy} and V_{yx}. The two values may not be the same, which means if file B depends on file A at value 1.0, then file A may depend on file B at value 0.1. The higher the value, the more change effects.

Formula 1 gives the basic calculation process. α, β are the proportion variables. T_{lcc} is the co-change times of file x and file y in the latest sub-versions. T_{lt} is the total change times of file y in the latest sub-versions. T_{cc} is the co-change times of file x and file y in the whole version history. T_t is the total change time of file y in the whole version history.

```
diff --git
a/src/main/java/org/bukkit/craftbukkit/CraftWorld.java
b/src/main/java/org/bukkit/craftbukkit/CraftWorld.java
@@ -1 +1 @@
```

Figure 2. Simplified commit detail modifications

B. Data simplification

The data simplification module takes one commit entry as input, and outputs a simplified line change number. The sample result is shown in Fig.2, where the file path and change scale are output as the key information to describe this given commit entry.

C. Dependency directed graph visualization

Combining with dependent sequence data and code changes, this module uses JUNG to visualize the dependency directed graph.

Fig.3 gives an example of the dependency diagram with added lines from backtracked commit 300 to 350 of CraftBukkit project in GitHub. The commit number is reversed from the latest commit to the first one.

To make the layout of this diagram stable, we choose a suitable layout for the last sub-version, and make it the layout template for all the other sub-versions. So in further study, we can make an animation to visualize the evolution of the source code with dependent sequence.

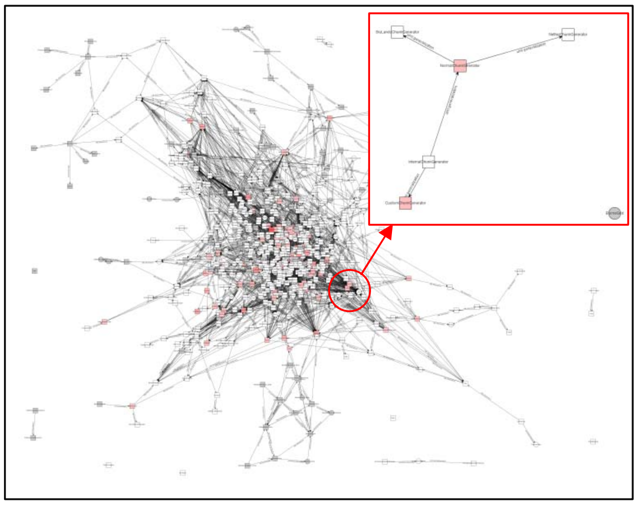

Figure 3. Class dependency diagram with commit information

D. Barcode generation

This module uses parallel line segments to represent a group of commits, and the gaps between two line segments to represent the periods with no commits. The width of each line is considered as:

$$\text{Width} = \alpha \times \text{Commits} + \beta \times \text{Files} + \chi \times \text{LOC} \quad (2)$$

α, β, χ are the variables to decide proportion of commit count, change files and change lines in the visualization. The width of each gap is considered as:

$$\text{Width} = \delta \times \text{Days} + \varepsilon \times \text{OCommits} \quad (3)$$

δ, ε are the variables which decide the proportion of the days between two commits and commits from other developers in that period in the visualization [5].

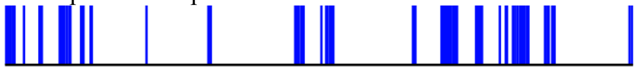

Figure 4. Sample Barcode with commits information

According to the above definitions, this module adopts Scalable Vector Graphics (SVG), an XML-based vector image format for graphics, to visualize barcode, as shown in Fig.4.

IV. Case Study

After the implementation of CAT, we do some case studies to apply the proposed mining method to find developers' behavior patterns.

A. Data

We collect data from GitHub and export detail commit logs using the git command. Fig.5 gives a sample commit log entry. In each commit log entry, there is a cryptographic hash uniquely identification, a developer name, an email address, the commit time and comments, and change details.

Change information can be divided into several parts: the file path, the added and deleted lines of the code with "-" or "+" marks in front of it.

```
commit 05a3daf39c98a187d86754fac850bf948e11098e
Author: feildmaster <admin@feildmaster.com>
Date:   Fri Jul 13 22:57:42 2012 -0500

    Add missing setLastDamageCause. Thanks MonsieurApple

diff --git a/src/main/java/net/minecraft/server/EntityLiving.java
b/src/main/java/net/minecraft/server/EntityLiving.java
index bebac89..cf71f17 100644
--- a/src/main/java/net/minecraft/server/EntityLiving.java
+++ b/src/main/java/net/minecraft/server/EntityLiving.java
@@ -860,7 +860,7 @@
public abstract class EntityLiving extends Entity {
                } else {
                    this.world.makeSound(this, "damage.fallsmall",
1.0F, 1.0F);
                }
-
+               this.getBukkitEntity().setLastDamageCause(event);
               this.damageEntity(DamageSource.FALL, i);
           }
           // CraftBukkit end
```

Figure 5. Sample commit log entry from project CraftBukkit

Eight projects from the GitHub are chosen as data-set in the following studies. Some basic metrics of the data-set are given in Table 1, where "Last(Days)" describes the days between the first commit and the last one.

Due to the length limit, this paper takes example of CraftBukkit project to illustrate the experimental results and discussions. CraftBukkit is an implementation of the Minecraft Server Mod developed using Java language. It's a relatively young project since December 2010. We collected its commits and the developer information from December 2010 to July 2012.

TABLE I. DATA-SET METRICS

Project	Developers	Commits	KLOC	Last(Days)
cassandra	35	7790	224.8	1340
clojure	101	2359	53.4	2405
CraftBukkit	100	1807	60.1	570
git	1171	30614	181.2	2754
jquery	186	4694	35.6	2416
rails	2027	33628	223.6	2899
sparkle	54	727	92.8	1892
voldemort	77	2387	171.2	4322

B. *Results and discussion*

1) RQ1. How do the changes affect the other souce files?

To show the influence of the code changes, we measure the CraftBukkit, and choose 50 sequent commits as a sub-version to build the dependency directed graph. According to the previous studies [2, 3], "expert", the core developer of a source file cluster in a period of time, can be found in package level. We choose the package as the file cluster, and generate the graph for each cluster in every sub-version.

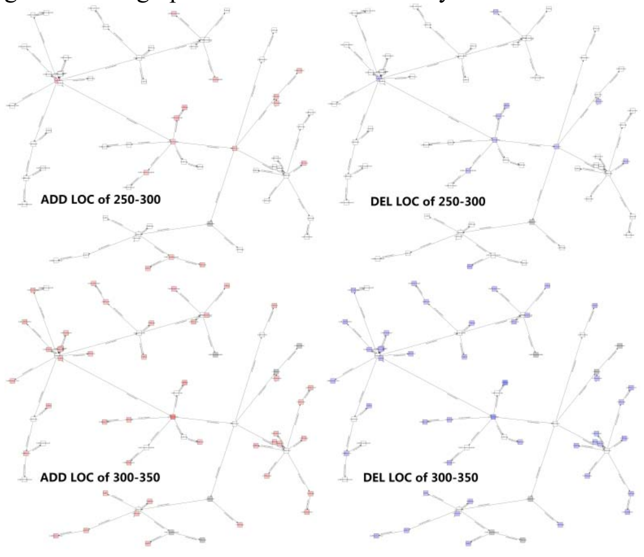

Figure 6. Code changes on package org.bukkit.craftbukkit.entity

Fig.6 shows that in commits 250-300, there are several changes in most files. However some "core" files, which are represented by the vertexes with high degrees in this graph, are changed more frequently. The color depth, which present the change scale of each file, shows that the change scales are lower than 5% of most files. But in commits 300-350 of this package, the change scale increased. Many files changed in this sub-version especially those files with direct relationship with the files that have been changed in previous sub-version. This phenomenon shows that, small changes in previous commit, especially those changes occurred in "core" files, will cause large code changes in the following commits.

Besides CraftBukkit project, this phenomenon can be found in all other seven projects. The influence of the code changes can be well described by the dependency directed graph. The edges in graph that connected to the changed vertexes will help us to identify the future changes of the source files.

2) RQ2. What are the developers' commit habits look like?

Barcode is chosen to visualize the developers' commit habits. For each package, a barcode is generated for each developer that has ever made a commit on it.

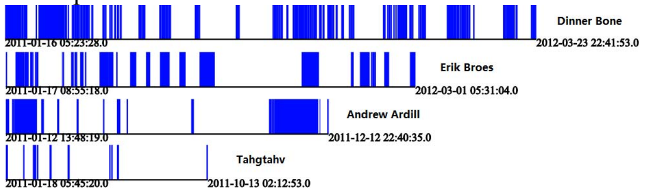

Figure 7. Barcode of package org.bukkit.craftbukkit.entity

Fig.7 shows the barcodes of developers on package org.bukkit.craftbukkit.entity. It is clear that Dinner Bone, who can be figured out as the "expert" of this package, commits the most parts of the code. But Tahgtahv and other developers that are not shown in this figure can be considered as the related developers, they contribute few parts of this package.

If in one commit, there are more than five changed files; and in each changed file, there are more than five changed lines, we call this commit as "huge commit". We calculate the average gap length between the huge commits, and find out the following pattern:

The average gap length between huge commits is 3 times of normal commits.

TABLE II. AVERAGE INTERVAL RESULT

	Avg interval around huge commit	Average Interval around normal commit
cassandra	5.592	1.720
clojure	6.189	2.039
CraftBukkit	6.317	1.895
git	5.582	1.796
jquery	5.336	2.059
rails	4.543	1.893
sparkle	4.562	2.606
voldemort	4.579	1.524

As the average gap length between huge commits is 5.34 days, and the normal commit is 1.94 days. In this calculation, we filter out those gaps larger than 10 days. We consider that the commits with 10 days apart are not in a series and these gaps can be noise data for further study [4].

However, for those developers who commit a little of this package, their commit patterns do not comply with the pattern so well.

3) RQ3. What's the relationship between core developers' commit habits and the file version evolution?

The developers' commit habits tell us a common phenomenon that people need rest, thinking or other preparation to start a series of jobs, or after finishing a series of jobs. The developer will have to do some testing or summary, design for the next part, and take a rest or other kinds of breaks. But in one series of jobs, made up of many commits in sequence over continues time, the developers will have to focus on their jobs, and commit what they have done just in time. Because of the continuity of tasks, the developer does not have to take too long to start a new task after finishing the previous one.

In the iterative development process, developers should make plans at the start of each iteration, and do validation at the end of the iteration. Now we reduce the scale of iterations, and regard one series of jobs as an iteration. The developers' commit habits in one sequence tasks can be expanded to a large scale of tasks, and the commit patterns give us a persuasive proof of the rightness of small iterations in software development. The dependency directed graph of each package over sub-versions shows that the change scale is different in each sub-version. Combining with the developers' commit data, especially the core developer's commit data, we can find the core developer's leading effect clearly in the graph from version to version.

V. CONCLUSION AND FURTHER WORK

In this paper, we explore the developer's commit habits and their relationship with file version evolution using dependency directed graphs and barcodes, in order to find out the root cause of the indefinite software release frequency in GitHub.

To describe the commit habit and file version evolution, four metrics are proposed, namely the changes in each commit, the interval between two commits, the author of each changes, and the source code dependency. Then, we develop a software repository mining tool CAT to generate the metrics, and visualize the results using class dependency graphs and barcodes. Finally, through the case study of eight projects from GitHub, we reveal the following patterns: 1) the later changes can be mostly associated with the previous changes according to the edges in dependency direct graphs; 2) the developers' commits can be grouped into several series and the average gap between huge commits is much longer than other commits.

For the developers in GitHub, knowing the habits of other developers in the group is helpful to make a better work plan and do the important commit at the right time. For the user of the freeware in GitHub, knowing when to release the next version of the freeware will help to make some adjustments early. And for the competitors of the product in GitHub, knowing the software release cycle will help to make a better development plan of their own.

This study is not the first to explore the relationship between the developer's activity and the file version evolution. Richard Wettel and Michele Lanza [6, 7] presented a 3D software visualization approach based on a city metaphor and investigated several factors that concur to the realistic aspect of the city. Peter Weißgerber et al [8] used three visualization techniques to help examine the development behavior of the programmers and the partitioning of the task between them in detail, and recognized when particular developers have been especially active and which developers have worked together or alone on which files. The bulk of the work on the repository mining is working on the cooperation of developers or the evolution of project structure. However these researches did not concern about the relationship between developer's activity and file evolution. This study aims at exploring this relationship, and finally revealing two patterns that can give a sight of the developer's commit habits and the effects to file version evolution.

Moreover, there are still some valuable questions for further studies. Are there any qualities issues that stop the developer's work or do they have to wait for the completion of dependent part? Has the software's quality be affected when the developer make a huge commit? In the near future, we plan to dig the source code, study the comments in each commit and add in some issue tracker to expand our repository, achieve a better understanding of developer's commit habit and try to find out the relationship between the developers' habits and software quality.

REFERENCES

[1] Xu Ben, Shen Beijun, Yang Weicheng, "Mining Developer Contribution in Open Source Software Using Visualization Techniques". ISDEA, 2013, pp. 934-937.

[2] Omar Alonso, Premkumar T. Devanbu, Michael Gertz, "Expertise Identification and Visualization from CVS", In MSR'08: The 5th Working Conference on Mining Software Repositories, pp. 125-128. ACM, 2008.

[3] David Schuler, Thomas Zimmermann, "Mining Usage Expertise from Version Archives", In MSR'08: The 5th Working Conference on Mining Software Repositories, pp. 121-124. ACM, 2008.

[4] Harvey Siy, Parvathi Chundi, Mahadevan Subramaniam, "Summarizing Developer Work History Using Time Series Segmentation", In MSR'08: The 5th Working Conference on Mining Software Repositories, pp. 137-140. ACM, 2008.

[5] Jason R. Casebolt, Jonathan L. Krein, Alexander C. Maclean, Charles D. Knutson, "Author Entropy vs. File Size in the GNOME suit of Applications", In MSR'08: The 5th Working Conference on Mining Software Repositories, pp. 91-94. IEEE, 2008.

[6] Richard Wettel, Michele Lanza, "Visualizing Software Systems as Cities", In Proceeding of VISSOFT 2007: 4th IEEE International Workshop on Visualizing Software For Understanding and Analysis, pp. 92-99. IEEE, 2007.

[7] Richard Wettel, Michele Lanza, "Visual Exploration of Large-Scale System Evolution", In WCRE'15: 2008 15th Working Conference on Reverse Engineering, pp. 219-228. IEEE, 2008.

[8] Peter Weißgerber, Mathias Pohl, Michael Burch, "Visual Data Mining in Software Archives To Detect How Developers Work Together", In MSR'07: The 4th Working Conference on Mining Software Repositories, pp. 9-16. IEEE, 2007.