Code Change Notification Tool

David E. Carlyn
Computer Science Department
Kent State University

Kent, United States
dcarlyn@kent.edu

***Abstract*—A common issue in software development is the time it takes for program comprehension. It has been observed to take up over half of the time of the developer [6]. Many argue that communication with the developers of the code is the most effective and efficient way to understanding the code. This paper proposes a tool created to use a popular collaboration application, *Slack*, that will notify the developers of a section of code that has been modified. This is achieved by using a *Slack app* that monitors *GitHub* repositories for changes. If any change is made on a section of code that a developer registered to monitor, then a slack message is sent to that developer.**

Keywords—followMe, Slack, GitHub, diff, curl, code comprehension

I. Introduction

Software development as a student or a beginner is very rudimentary and small, so most projects have one developer but occasionally there will be two to four developers. Code comprehension in this case is relatively simple and doesn't require much time. Most time is spent planning, coding, possible testing and debugging. Unfortunately, real software is not this simple to develop. Software development in the industry leads to large and complex software. These programs will age and, if it wants to survive, evolve [4]. Large evolving software has many authors and these developers will come and go. New programmers will be introduced to large and complex code that they have never seen before. With this, a common problem arises: lengthy program comprehension.

It has been found that most of the time spent during programming task is on program comprehension. Much research will claim that program comprehension takes up between 50% - 90% of the total time on a programming task [6]. Many have tried to find ways to decrease the time needed to understand code by changing the process of developing code, creating IDEs, IDE plugins that assist with program comprehension, or analyzing which methods of understanding code is the most effective.

One area of research developed a tool to inform the developer of the author who would be the most knowledgeable of an area of code [2]. Rather than trying to interpret the code yourself in a system that you've probably barely touched, it would be more cost effective, in terms of time, to talk to an expert. In a few conversations, it can be theorized that a developer would understand the code and how to work with it quicker than reading the code by itself.

While this approach of informing the developer of the expert of the code is an effective one, it still leaves room for bugs and long program comprehension time. The developer still must make the time and effort to communicate with the expert. New programmers may tend to ignore going to someone and try to figure out the code on their own. This would lead to a high probability of bugs and long program comprehension. There needs to be an approach where the expert of the code will know who is working on their previous code. Such an approach works to notify the expert when a section of code they tagged has been modified. This will be more of a supervised approach where the expert has control than an information provided approach where the new developer has control.

The tool discussed in this paper, which will be called *followMe*, does just that. Developers can use this tool to track a section of code that will inform them of changes to it. It is worth noting, that in development, time is a valuable resource that we want to use efficiently as much as possible. We don't simply want to inform the developer that their code has been change, but how it has been changed. An ideal tool would inform the user of the types of changes that they would want to see. Simple textual changes may not be important, but syntactical and semantical change could be. The expert developer is looking for meaningful changes [7].

The tool integrates two popular tools for software development: *GitHub* for code repositories and version control, and *Slack* for communication. By using these popular tools, it is more likely that the *followMe* tool will be used in practice and can collect research data in future experiments.

The paper will be organized as follows: section II will talk about the use, workings and the set up of *followMe*, section III will go further and explain the implementation of the tool, section IV will show the working results of the tool, section V will talk about future work of the *followMe* tool, section VI will discuss related works on program comprehension and code change detection, section VII will discuss the current limitations of the tool, and finally the paper will conclude with section VIII.

II. Tool Use & Workflow

A. Prerequisites

As stated before, this tool uses *GitHub* and *Slack,* so an account is needed for both. A server is also needed. The server will need to be able to receive webhooks and allow *curl* commands to be called. With that, the server will need to have the *curl* tool installed. However, the code can be changed to use another tool that can access API data. The code for the *followMe* code can be found at the URL: *https://github.com/dcarlyn/GitHubSlackCodeNotification*. The next three sections will explain the set ups for *GitHub*, *Slack*, and your server.

B. *GitHub Setup*

This section will go over how to set up everything on the *Github* side of this project. The workflow can be referenced in Figure 1. First, find the repository with the desired code to track. Then, access the settings by clicking on the "Settings" section of the page. There will be a set of links on the left side of the page. Click on the "Webhooks" link. Then press the "Add webhook" button and give the required credentials used to access the repository.

Now, *GitHub* sends data by a *JSON* data structure in the form of a payload, settings for how it will be sent, and when to send it need to be set. Under the "Payload URL" put the location (URL) to the server at the location where the downloaded *followMe* code will go (discussed further). Under "Content type" choose "application/json" so the payload is in a *Javascript Object Notation* format. Skip over the "Secret" section and select "Let me select individual events" after the question "Which events would you like to trigger this webhook?" There will be a screen that looks like the one in Figure 3. A list of options will be given to select. All that is needed to be selected for this tool is the "Pushes" option. Finally, click the "Add Webhook" button and the *GitHub* setup is complete.

You will also need your personal access token to allow for 5000 requests to the *GitHub API* per hour. Otherwise, without proper authentication, only 60 requests per hour would be permitted. To do this, click on the profile icon in the upper-right corner of the page and click on "Settings". Then, click on "Developer settings" and then "OAuth Apps." Finally, click on "New OAuth App" and fill out the form giving the URL to your server where the code is to be saved. After you create the app, you will be given a *client id* and a *secret id*. Copy them and save them for later.

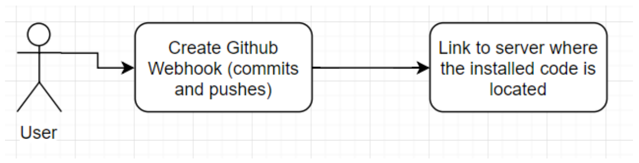
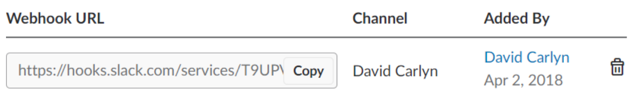
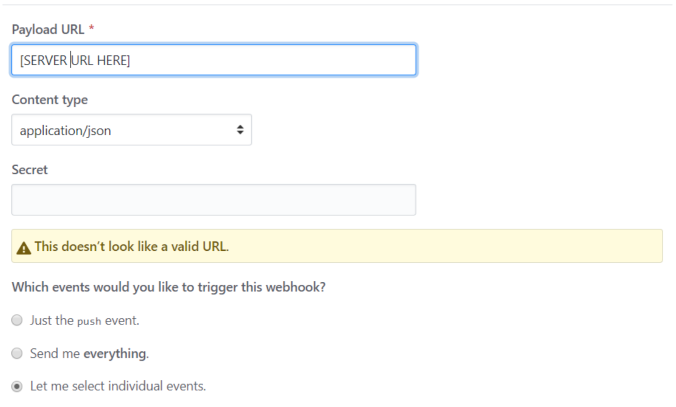

User
Create Github Webhook (commits and pushes)
Link to server where the installed code is located

Figure 1



Webhook URL	Channel	Added By
https://hooks.slack.com/services/T9UP\ Copy	David Carlyn	David Carlyn Apr 2, 2018

Figure 2



Payload URL *
[SERVER URL HERE]

Content type
application/json

Secret

This doesn't look like a valid URL.

Which events would you like to trigger this webhook?

- Just the push event.
- Send me everything.
- [✓ Let me select individual events.]

Figure 3

C. *Slack Setup*

Now, a simple *Slack app* will need to be set up a to receive POST messages from the server. The workflow for this step can be referenced in Figure 4. First, go to https://api.slack.com/apps , and give the proper credentials for the desired *Slack* account the app will live. Click on the "Create New App" button in the top right corner of the page. You will be given a prompt to give your app a name and choose the Development *Slack Workspace* where your *Slack app* will live and notify you about changes to desired sections of code.

After you have created your *Slack app*, you will be brought to a page to configure your app. Under the "Add features and functionality" section, click on "Incoming Webhooks." You will be brought to a page describing the use of webhooks in *Slack*. You will need to turn on this feature by clicking the switch in the top right corner of the middle section of the page. Then, you will scroll down to the bottom of the page and click the button that displays "Add New Webhook to Workspace." You will be prompted where you want the *Slack app* to post messages to. Give your desired location and click "Authorize." Finally, you will be brought back to the page about webhooks. At the bottom of the page under the "Webhook URL" section, you will be given a URL, as shown in Figure 2, where you can post messages to. Copy this URL and save it for the next step.

D. *Server Setup*

The server setup up is simple if you already have a web server set up. If not, then you will need to do so before you proceed to the next step. Also, you will need the latest version of PHP install. Specifically, you will need access to the *curl* command within PHP. This tool works with PHP 7. The workflow for this step can be referenced in Figure 5.

Find a location on your web server that will house the code for the *followMe* tool. Download the code from *Github* to this

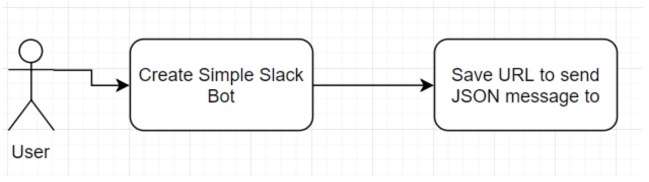


Figure 4

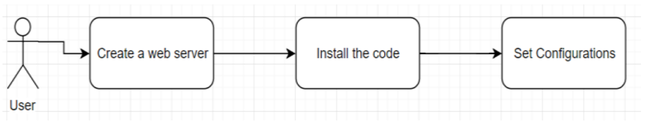


Figure 5

location. Next, go to the *get_github_data.php* file and change the string next to "CURLOPT_USERAGENT" to your *Github* username. You will also have to put your *GitHub* credentials in the *AuthenticateGithub* function. Your *client_id* will be the first parameter and your *secret id* will be the second. Now, you will have to define a tag that will be used to determine if a file is to be tracked or not. To set this tag, go to the file *filter_tagged_files.php* and change the *$tag* variable to your desired tag. Make this tag as unique as possible so that it would not be found anywhere else in your code but in the comments. This next part is optional. Go to the *set_file_pairs_diff.php* file and under the "//RUN" comment you can change the line that gets the difference between the old file and the new to a differencing tool that you prefer. This tool uses the default *diff* tool on most *Linux* machines. Lastly, in the *notify_slack.sh* file you will need to change the URL at the end to the URL you saved from the *Slack Setup* step. Finally, to set a custom tag to be put in the comments of code to indicate that the file needs to be tracked go to the *filter_tagged_files* and set the *$tag* variable to a custom string.

Once all of this is done you will be set up to use the tool. Make sure that all the files on your server that you downloaded are given the proper permissions.

E. *Using the tool*

Using the tool only requires one step. All you have to do is to find the code you want to track changes on and put a comment somewhere in the file with the tag you specified in the file *filter_tagged_files.php*. Once this is done, then you sit back and wait for notifications to be sent to your *Slack app*. Once someone commits changes to your section of code you will receive a *Slack* message containing the differences between the newly committed file and the last time it was commit with the differencing tool that you set up in the *Server Setup* step.

The workflow of using of the tool can be referenced in figure 6.

III. Tool Implementation

In this section, significant parts of the code will be explained so others can not only use this tool, but also expand upon the tool to fit their needs. This will also allow others to understand the capabilities and limitations of the tool as it currently stands. The following files will be discussed: *file_pair.php*, *get_file_pairs.php*, *git_file_paths.php*, *get_github_data.php*, *index.php*, *main.php*, *notify_slack.php*, *notify_slack.sh*, *set_file_pairs_diff.php* and *filter_tagged_files.php*.

A. *index.php*

This is a short file since it is the webhook. It captures the *GitHub* payloads that come from commits pushed to the repository. The payloads are in a *JSON* format. Then that data is written to a file called, "webhook_file.txt." Finally, it executes the command *sudo php main.php*. This will run the tool with root permissions. Make sure your web server will allow this.

B. *main.php*

Here is where all the sub events are managed. The code is shown in Figure 7. First, we must open our file with the *GitHub* payload data (*webhook_file.txt*). Then, that payload data is decoded so we can easily work with it in PHP. From this, the commit and repository data are extracted. Soon after this, the *AuthenticateGithub* function is called to authenticate the use of the *GitHub API*, so the server isn't locked out from using it if there are too many calls to the API.

The repository data gives us the username of the owner of the repository and the name of the repository. These two pieces of data are useful for *GET* calls to *GitHub*. They are needed for determining the URL to make the *GET* call to. The commit data gives us a lot of information, but we use it to get the file paths of the changed files and the file pairs of these changed files. These will be discussed in depth in the *get_file_paths.php* and the *get_file_pairs.php* files respectively.

To get the file paths of the changed files we pass the commit data, username of the owner of the repository, and the name of the repository to the *GetFilePaths* function (defined in *get_file_paths.php*). Next, to will get the file pairs, meaning that we will get the newly committed version of the changed files and the version before the commit was made. To get the file pairs we pass the file path data, commit data along with the

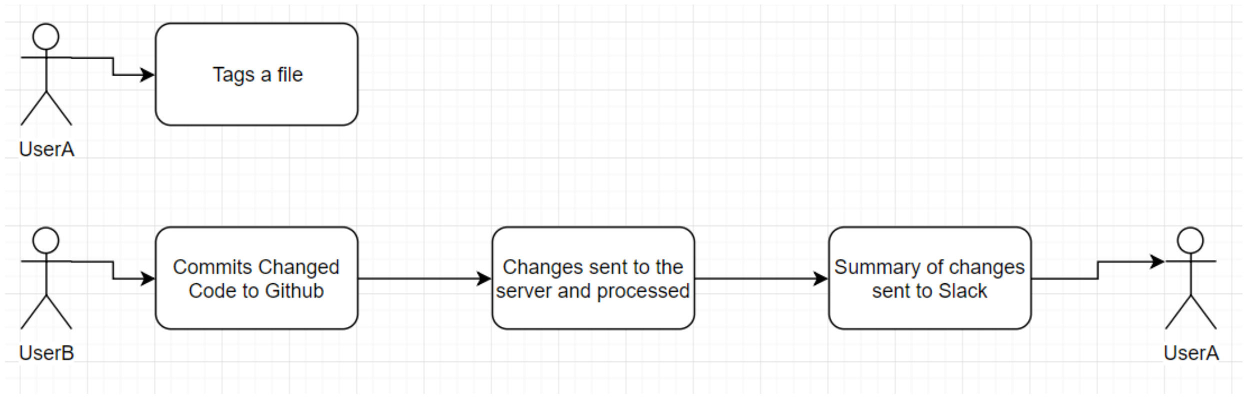
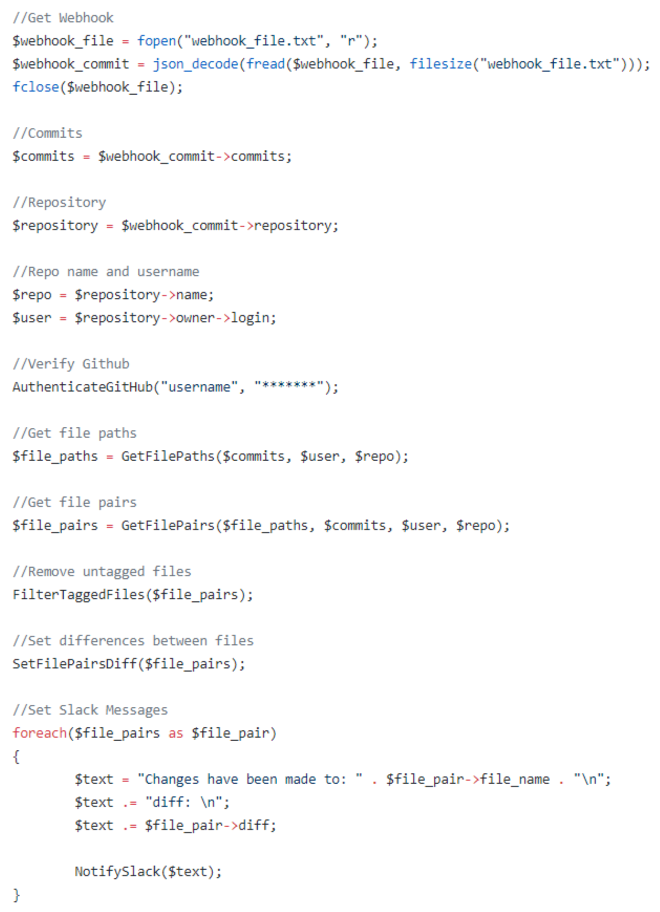

UserA
Tags a file
UserB
Commits Changed Code to Github
Changes sent to the server and processed
Summary of changes sent to Slack
UserA

Figure 6



```
//Get Webhook
$webhook_file = fopen("webhook_file.txt", "r");
$webhook_commit = json_decode(fread($webhook_file, filesize("webhook_file.txt")));
fclose($webhook_file);

//Commits
$commits = $webhook_commit->commits;

//Repository
$repository = $webhook_commit->repository;

//Repo name and username
$repo = $repository->name;
$user = $repository->owner->login;

//Verify Github
AuthenticateGitHub("username", "*******");

//Get file paths
$file_paths = GetFilePaths($commits, $user, $repo);

//Get file pairs
$file_pairs = GetFilePairs($file_paths, $commits, $user, $repo);

//Remove untagged files
FilterTaggedFiles($file_pairs);

//Set differences between files
SetFilePairsDiff($file_pairs);

//Set Slack Messages
foreach($file_pairs as $file_pair)
{
        $text = "Changes have been made to: " . $file_pair->file_name . "\n";
        $text .= "diff: \n";
        $text .= $file_pair->diff;

        NotifySlack($text);
}
```

Figure 7

username and repository name to the *GetFilePairs* function (defined in *get_file_pairs.php*). Next, the file pars are passed into the function *FilterTaggedFiles* where file pairs that do not have a tagged will be removed. With the tagged file pairs of the changed files we can send this data to the *SetFilePairsDiff* function to receive the difference between these files using the differencing algorithm in the *set_file_pairs_diff.php* file. Note that the difference is put in the file pairs structure which is defined in the *file_pair.php* file. Finally, for each file pair we put that data together into a string and send it to the *NotifySlack* function that will *POST* the string to the *Slack app* that was created in the previous section.

C. get_file_paths.php

This file contains the function *GetFilePaths* which can be referenced in Figure 8. Given the commit data, the username of the owner of the repository and the repository name, an array of file paths is returned. For each commit in the commit data the *GetGithubData* function (defined in *get_github_data.php)* is called given the URL *https://api.github.com/repos/[username]/[repo]/commits/[commit_id]*. [username] is the username, [repo] is the repository name, and [commit_id] is the id of the commit obtained from the commit data. From this URL we will obtain specific data from the commit. This includes all the files that were changed. For each file that was changed the file path is obtained and added to an array to return later.

D. get_github_data.php

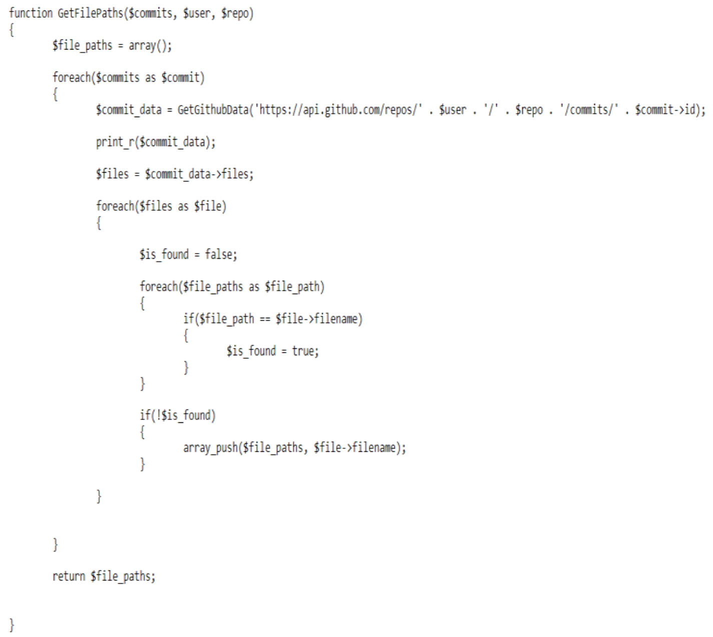
Given the *GitHub* URL to obtain data from, the function *GetGithubData* will return a *JSON* objection received from the *GET* call to *GitHub*. This function simply uses a *PHP curl* to get the data, decodes the *JSON* string, and returns it.

E. get_file_pairs.php

Before explaining this file, it is worth noting that in the *GitHub API*, commits has an id called a *SHA*. This is important in determining the correct URL to make our *GET* call to.

This file contains the function *GetFilePairs*. The code can be referenced in Figure 9. This function will take in an array of file paths, commits, the username of the owner of the repository and the repository name then it will return an array of file pair structures (defined in *file_pair.php*). The function begins by going through each file path to get the newest version of the file and the version of the file before the given payload of commits, now called the old version of the file. To get this data the *GetGitHubData* function is used with the URL *https://api.github.com/[username]/[repo]/commits?path=[file_path]*. This data will give us all the information and the history of the file with the file path: [*file_path*]. This data contains the commit ids or *SHA*s associated with each version of the file. To get the contents of a file the URL *https://api.github.com/[username]/[repo]/contents/[file_path]/?ref=[SHA]* is passed to the *GetGitHubData* function. This content is encoded with base64. Newline characters must be removed from the raw encoded content before using a default *PHP* function to decode the content.

To get the content of the new version of the file, the first element of the file data obtained from the first URL above is



```
function GetFilePaths($commits, $user, $repo)
{
        $file_paths = array();

        foreach($commits as $commit)
        {
                $commit_data = GetGithubData('https://api.github.com/repos/' . $user . '/' . $repo . '/commits/' . $commit->id);

                print_r($commit_data);

                $files = $commit_data->files;

                foreach($files as $file)
                {

                        $is_found = false;

                        foreach($file_paths as $file_path)
                        {
                                if($file_path == $file->filename)
                                {
                                        $is_found = true;
                                }
                        }

                        if(!$is_found)
                        {
                                array_push($file_paths, $file->filename);
                        }

                }


        }

        return $file_paths;


}
```

Figure 8

used to pass the SHA in the second URL above. To get the old version of the file there is more of a process. With the history of the file, all the SHAs can be obtained. These SHAs are then compared to the SHAs from the commit data that was passed to the function. Once a version of the file where the SHA is not contained in the commit data is found, then that SHA is used in the second URL above to obtain its contents and is then decoded appropriately.

The file path (which includes its name), old file contents and the new file contents is put into a file pair structure and push into an array and returned.

F. *filter_tagged_files.php*

This file will remove all the file pairs that don't have the tagged defined in this file. Using the simple *strpos* function to calculate whether the file contains the tag, the position of the file pair is stored to be deleted later. After all the file pairs that do not contain the tag are found, they are then removed from the file pairs array.

G. *set_file_pairs_diff.php*

The function *SetFilePairsDiff* is defined in this file. The function will take in a reference to an array of file pairs and calculated the difference between the old and new versions of the file and store it in the same file pair structure. The differencing algorithm used by default is the *diff* tool. However, this is where any differencing algorithm can be used to replace the current method to fit the needs of any user.

H. *notify_slack.php*

This file contains the *NotifySlack* function and it takes in a string that will be sent to *Slack* as a notification. The string must be preprocessed to set up single quotes and double quotes to be passed through the command line and as an attribute in a *JSON* object. This code can be referenced in Figure 10. After the preprocessing the string is sent as an argument to the *notify_slack.sh* command.

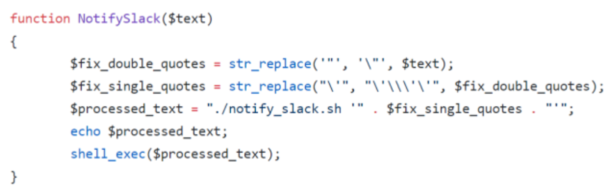
I. *notify_slack.sh*

This is a *SHELL* file that simply takes a string as an argument and sends it to the *Slack* app that is given via URL at the end of the file. This file can be modified greatly to utilize more functionality and features of *Slack* apps.

IV. RESULTS

```
function GetFilePairs($file_names, $commits, $user, $repo)
{
        $file_pairs = array();
        $file_pair;

        //For each file
        foreach ($file_names as $file_name)
        {
                //Store name in file_pair
                $file_pair->file_name = $file_name;

                //Getting file data
                $file_data = GetGithubData('https://api.github.com/repos/' . $user . '/' . $repo . '/commits?path=' . $file_name);

                //Get new file contents
                $new_file_contents = GetGithubData('https://api.github.com/repos/' . $user . '/' . $repo . '/contents/' . $file_name . '?ref=' . $file_data[0]->sha);

                //Grab contents and decode
                $new_file = base64_decode(str_replace("\n", "", $new_file_contents->content));

                //Store in file pair
                $file_pair->new_file = $new_file;

                //Get old file by avoiding ones from the same commit payload
                foreach ($file_data as $file)
                {
                        //print_r($commits);

                        $is_found = false;

                        //Checking commits from payload
                        foreach($commits as $commit)
                        {
                                //Is this file in this commit?
                                if($commit->id == $file->sha)
                                {
                                        $is_found = true;
                                }
                        }

                        if(!$is_found)
                        {
                                //Get new file contents
                                $old_file_contents = GetGithubData('https://api.github.com/repos/' . $user . '/' . $repo . '/contents/' . $file_name . '?ref=' . $file->sha);

                                //Grab contents and decode
                                $old_file = base64_decode(str_replace("\n", "", $old_file_contents->content));

                                //Store in file pair
                                $file_pair->old_file = $old_file;

                                //Insert into file_pairs
                                array_push($file_pairs, $file_pair);

                                break;
                        }
                }
```

Figure 9



```
function NotifySlack($text)
{
        $fix_double_quotes = str_replace('"', '\"', $text);
        $fix_single_quotes = str_replace("\'", "\'\\\'\'", $fix_double_quotes);
        $processed_text = "./notify_slack.sh '" . $fix_single_quotes . "'";
        echo $processed_text;
        shell_exec($processed_text);
}
```

Figure 10

This section will be a step by step example look at the use of the *followMe* tool. The example will be used on a small code repository for a personal PHP website project. The changes made in the example will be small textual changes, as the differencing algorithm being used is *diff*. Syntactical and semantical changes will not be explicitly detect as they will only be seen as textual changes. The example will show that two commits in the same payload with different changes to the same file should be recorded as one larger change, that files not tagged will not be reported to *Slack*, and multiple file changes will be sent to *Slack* separately according to the design of the tool.

A. *Commits and File Changes*

There were three commits to the payload that was send to the tool.

1. *Commit 1*

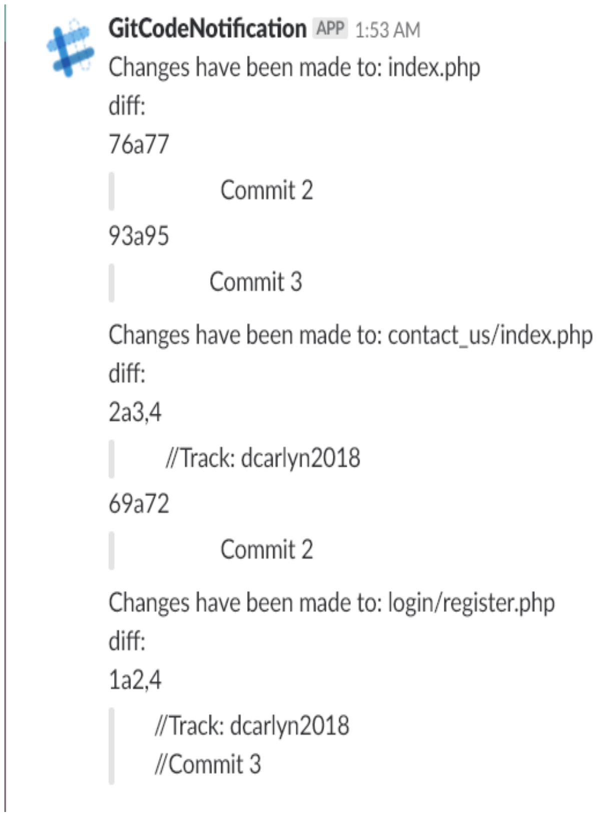
   In this commit, only the *index.php* file was changed in the *PhiPsiKsu* repository. The file already contained a valid tracking tag but was changed with the addition of the comment "Commit 1".

2. *Commit 2*

   In this commit, the following files were modified: *index.php, about/index.php, contact_us.php*. In the *index.php* file, another comment was added that stated "Commit 2". In the *about/index.php* file, a comment of the exact kind as *index.php* was added. However, there was no tag in this file. Lastly, *contact_us/index.php* was tagged and modified with a comment exactly like the one in *index.php*.

3. *Commit 3*

   This commit contained modified versions of the files *index.php* and *login/register.php*. The comment from *Commit 1* was changed to "Commit 3" in the *index.php* file. The *login/register.php* file was tagged and modified with the same comment.



```
GitCodeNotification APP 1:53 AM
Changes have been made to: index.php
diff:
76a77
|            Commit 2
93a95
|          Commit 3
Changes have been made to: contact_us/index.php
diff:
2a3,4
|      //Track: dcarlyn2018
69a72
|            Commit 2
Changes have been made to: login/register.php
diff:
1a2,4
|     //Track: dcarlyn2018
|     //Commit 3
```

Figure 11

B. *Slack Notifications*

After all the commits were made, they were pushed to *GitHub* in one package. The results on *Slack* can be referenced in Figure 11.

The tool ignored that *index.php* was in multiple commits and simply took the difference between the newest version of the code referenced in the latest commit sent and compared it to the latest version of the code not referenced in the commits of the same payload. The tool also ignores the changes made to *about/index.php* because it was not tagged to be tracked. The other files have been tracked because we tagged them, and it didn't matter which commit they were in; they were still reported as changed.

V. FUTURE WORK

This tool has an expandable future with three areas: extra features and functionality, effectiveness evaluation, and research expansion.

A. *Extra Features and Functionality*

There are multiple features and functionality that can be added to the *followMe* tool to make it more powerful. Some possible ones include but are not limited to: enhanced *Slack* functionality, more descriptive *GitHub* changes and finer grain tracking.

1. Some functionality that can be added to the *Slack* end of the project is command tagging, checking untagged code, work for multiple branches, work for multiple repositories and adding UI to make it easier and more appealing. Command tagging would be where in *Slack* the user could type a command to the *Slack app* and it would tag the file that you want to track without having the user manually edit the code and push to the repository. This could be difficult depending on the process of certain code collaboration projects. Checking untagged code could allow the user to type

a command to the *Slack app* and be able to check the code changes to a code file given specific parameters such as the file path and the duration between the most recent change and the older version of the code. Lastly, *Slack apps* allow for creative and easy to use UI to help expand the previous functionalities mentioned to make everything even easier for the user to use.

2. The *followMe* tool only scratches the surface of utilizing the information *GitHub* provides and the functionality it offers. Currently, the tool hasn't been tested for multiple branches. This could be modified to send more information about the code changed to indicated what branch it was changed in. As it stands, every user needs to set up this tool if they want to track separate files independently. This could be solved by posting to different channels per tag given or by using a mapping between tags and *Slack* users.

3. Currently, tracking a section of code can only be done at the file level. The tool could be expanded to allow for tracking at the class and even the function level. This would be difficult, but a possible approach is to allow the distance between the comment tag put in the code and the function or class the user would want to track. If the tag is X number of lines close to the function or class, then track just that function or class. Obviously, there are limitations to this approach, but it is a possibility. There is also the possibility of having an opening tag and a closing tag, but this could lead to code becoming crowded if too many developers are tagging the same section of code.

B. *Effectiveness Evaluation*

Given the short duration of the development of this tool, there has been no evaluation of its effectiveness, or even adequate testing. It would be preferred, once the tool is completed, to do a thorough experiment and evaluation of the effectiveness of the tool, and if this notification style approach help decrease the pending issue of significantly long program comprehension time. A sample representation of an experiment can be referenced in Figure 12.

a) Environment

An environment to track the effectiveness of this tool is essential in determining the type of data that we would want to collect. Some choices would be academia and classrooms, open-source programs and closed-source programs. Closed-source programs would be the best choice considering that data can be easily collected in the industry, the reliability and variability of closed-source processes for software development would prevent the most errors and threats to validity.

b) Data

The data that can be collected include the time spent on program comprehension, the speed that tasks are completed, the number of bugs reported and fixed and quality of the change detection based on user opinion. The data would need to be taken for a time frame before and after the *followMe* tool is introduced to the software development process.

c) Analysis

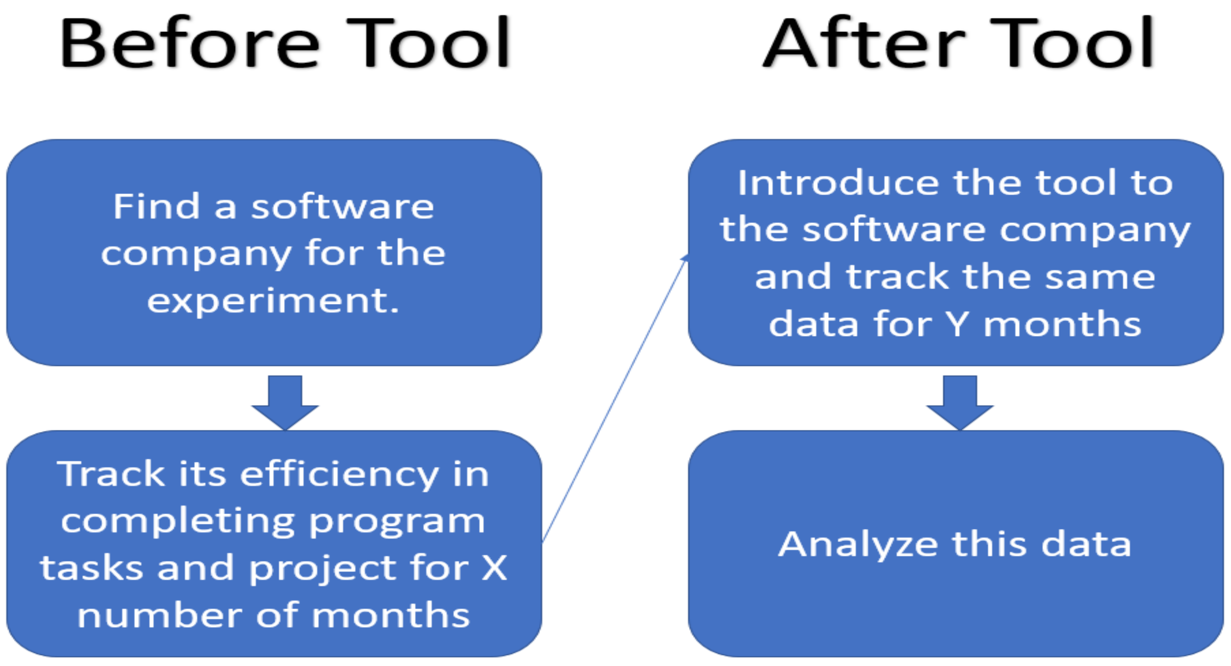

Before Tool
After Tool
Find a software company for the experiment.
Track its efficiency in completing program tasks and project for X number of months
Introduce the tool to the software company and track the same data for Y months
Analyze this data

Figure 12

After taking this data and displaying it in an informative way, the effectiveness of the tool can be analyzed. Questions that could be ask are: Does the tool decrease program comprehension time? Does the tool decrease the introduction of bugs to a program? Does the tool prevent the severity of bug introduced to the program? Do users find the tool to make software development more efficient?

There are multiple ways to approach evaluating the effectiveness of the tool in the software development process, but the above would be a great foundation to begin collection this information.

C. *Research Expansion*

This tool can be expanded for research purposes as well. Currently the tool uses *diff* for its differencing algorithm which is simply a textual-based approach to finding code changes, but semantical and syntactical changes could also be used and repeat in the recommended experiment just mentioned above. One such differencing algorithm that can be added to this tool is *srcDiff* from the *srcML* framework [5]. This would expand the tool to detect more than just mere textual changes, but syntactical and semantical ones as well.

VI. Related Literature

The following literature helped motivated and build the *followMe* tool discussed in this paper

A. *Software is Large*

Bringing it back to Lehman, almost all software can be classified as an E-type software. Based on Lehman's laws of E-type software, we know that they continually grow [4]. It is obvious that continual growth will lead to extremely large programs. These programs become well over 100s of MLOC (million lines of code).

When programs get to be this large many more programmers and software developers are needed to maintain and evolve the system. Unfortunately, programmers come and go and so does their understanding of the code. This eventually leads to the problem of long code comprehension time. Programs become more and more complex and harder to understand as it evolves.

B. *Software Evolves*

It is obvious from papers like [1] and [4] that software evolves. [4] gives us an understanding of E-type programs and that they need to evolve to survive the test of time. [1] gets a little more specific and tells us how APIs evolve and the effects and need of refactoring. [1] shows the need and desire to have automated refactoring tools and changes done to systems that previously depended on the older version of the API. If the developer on the project that worked with API is still around, then switching to the newer version of the API may be easy, but development projects are still faced with new faces dealing with leading this shift to the new API without fully understanding the old one. This can lead to many bugs and may even lead to deviation from the original design. It would be helpful to have programs that either automatically did the changes needed or information about who would be the best person to contact about existing code to understand it more thoroughly [2].

C. *Code Change Detection*

We are often faced with the problem of knowing how to change code. The paper [2] gives a tool and the analysis of the tool on discovering who would be the best developer to talk to given the amount of time one spends on a file of code. This has motivated the use of other developers as a source of program comprehension because they already went through all the hard work of either developing, understand and/or debugging the code themselves.

The paper [7] goes into detail by exactly determining what is a meaningful change and how it is customizable to fit the needs of individual developers. This method is another method, among many, that can be used in the *followMe* tool for the differencing algorithm. Simply sending the difference between file version is not sufficient enough information for developers but allowing them to choose the types of changes they want to see is what motivated the ability to customize the differencing algorithm for the *followMe* tool.

D. *Program Comprehension is Large*

Both [3] and [6] tell us that developers spend well over 50% of their time on almost any development task trying to understand the code. The problem that these papers introduced have been the main motivation behind this tool. [3] tested developers by giving them time sensitive tasks and found that much of the time they spend solving the tasks were spend understanding the code. Most developers could not complete all the tasks given within the time limit. [6] collected data by recording mouse and keyboard events to determine what the developer was doing. The paper ended up concluding that about 70% of the time used to complete a task was program comprehension, but one could argue that another 20% was also program comprehension, making it 90% of the time.

E. *Code Analysis*

Last, but not least, is the paper [5]. It is a simple paper about a tool called *srcML*, but it was another motivation for this tool. Originally, the *followMe* tool was designed to use the *srcDiff* tool of *srcML*, but due to time constraints that couldn't be achieved. *srcML* and its tools allow for a deeper analysis of source code at a fast speed. This tool could even allow for far more advanced functionality to be added to the tool, especially in terms of tagging a function to be tracked considering we could work in another mark up.

VII. Threats To Validaty / Limitations

The *followMe* does have limitations to current functionality that prevent it from being a standard method of code tracking, bug prevention and decreasing program comprehension.

A. *Limitations on Use*

The biggest limitation on use is the amount of *GitHub API* request that are allowed from the server. If authentication is set up correctly, then 5000 requests are allowed per hour, but if it is not set up correctly, then only 60 requests are allowed per hour. Currently, the tool will make 2 + 3n request per payload sent to the server, where n is the number of files changed in the payload. This can become expensive and force this tool to be used on smaller code projects. It would be ideal to have unlimited request to *GitHub's* API, but they must prevent too much traffic to their API.

The small tutorial on setting up credentials for use of the *GitHub API* is also a limitation. The tutorial may be incomplete. Currently, the tool is being authenticated, but will still only accept 60 requests per hours. Due to limited time, this could not be fix.

It would also be hard to do an experiment with a tool that requires a separate server per user which also requires a separate tag and webhook from *GitHub*. This would be hard to implement in a process given the amount of resources required to set it up.

B. *Limitation on Ability*

Currently, the tool has only been tested to work on *GitHub* repositories that have one branch (the master branch). It would be preferred if the tool worked on repositories with multiple branches and to only track a specific branch at a time. Adapting the tool to include this extra feature should not be difficult if it does not work current, but let it be known that it has not been tested.

It is also noted that in Figure 5, there is not much information on the changes made to the files. This is because the formatting the *diff* tool gives in its message is not fully supported to transfer from the server to the *Slack app*. Specific formatting must be done to have the message send correctly and developers may be restricted from their preferred way of formatting change detection messages.

C. *Limitation on Ease of Use*

If there is an API update on either the *Slack* or *GitHub* side, then maintenance of the tool is required. Of course, the user would not want the responsibility of updating the tool every time there is a structure update to either of these APIs.

The user must have access to a web server and must set it up to allow for webhooks to occur.

The user of the tool will also have to allow the server to use root functionality to be used every time a webhook is triggered and the server receives a payload. It is often desired to avoid giving root functionality, especially to a tool that allows any type of message to be sent to it.

VIII. Conclusion

Software is large and complex making it difficult to understand. This difficulty in understand source code is more than just an issue, it is an extreme nuisance. Imagine the amount of productivity that would result if we bring the average program comprehension time spend on a programming task from between 70% - 90% to less than 50%. Productivity would increase over 200% - 400%. This paper brings a new approach to helping developers with program comprehension. As opposed to an informative resource with who a developer (often a new one) should talk to, this tool flips the responsibility to the expert developer to the section of code himself/herself.

Software projects are too large for developers to continually check their existing code to ensure it has not been modified in a way that would break the system or lead to a poor design which would lead to a high probability of bugs being introduced in the future. This problem is similar to the one social media site have solved with social interaction and staying updated with your friends. One such site is *Twitter*. *Twitter* allows a user to "Follow" another user so that they can receive their "Twitter posts" on their timeline. The user is essentially being notified every time a user they follow updates their page. This same structure is found in the *followMe* tool. With this tool, developers are finally able to tag, or follow, a code file and be notified later if it has been modified without having to constantly check to code.

This tool is meant to help resolve the ever pending issue of long program comprehension time. This paper introduces a reactive approach to preventing long program comprehension time by allowing developers to be notified when sections of code are modified so that they may prevent the new developer to the code of the workings it or inform them that the change they made could be dangerous and/or break the project. The purposes of this tool are: to decrease program comprehension time decrease bug comprehension, improve productivity and improve code supervision.

Finally, given the current limitation of the *followMe*, hopefully this tool will be adapted and prove to be helpful toward the solution toward long program comprehension.

The use of this tool is allowed in any regard if it is cited, and credit is given where credit is due.

Acknowledgment

I would like to thank Dr. Jonathan Maletic and Drew Guarnera for helping motivate and bring to light this tool. I would also like to thank the students of the Spring 2018 Software Evolution class at Kent State University for helping inspire the idea of this tool.

References

[1] Dig, Danny, and Ralph Johnson. "How Do APIs Evolve? A Story of Refactoring." *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 18, no. 2, 2006, pp. 83–107.

[2] Kagdi, H., Hammad, M., and Maletic, J. I., "Who Can Help Me with this Source Code Change?", in the Proceedings of the 24th IEEE International Conference on Software Maintenance (ICSM'08), Beijing China, Sept. 28 - Oct. 4, 2008, pp. 157-166.

[3] Ko, Andrew J., et al. "An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks." *IEEE Transactions on Software Engineering*, vol. 32, no. 12, 2006, pp. 971–987.

[4] Lehman, M. M., and J. F. Ramil. “Evolution in Software and Related Areas.” *Proceedings of the 4th International Workshop on Principles of Software Evolution - IWPSE '01*, 2002.

[5] Maletic, Jonathan I., and Michael L. Collard. “Exploration, Analysis, and Manipulation of Source Code Using SrcML.” *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, 2015.

[6] Minelli, Roberto, et al. “I Know What You Did Last Summer - An Investigation of How Developers Spend Their Time.” *2015 IEEE 23rd International Conference on Program Comprehension*, 2015.

[7] Yu, Yijun, et al. “Specifying and Detecting Meaningful Changes in Programs.” *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, 2011.