THE SOFTWARE DEVELOPMENT PROCESS AND GIT

by

RAHKEEM D. GEORGE

Advisor

DR. DANIEL PLANTE
A senior research [proposal/paper] submitted in partial fulfillment of the requirements for the degree of Bachelor of Science in the Department of Mathematics and Computer Science in the College of Arts and Science at Stetson University at Stetson University

DeLand, Florida

Spring Term

2016
ACKNOWLEDGMENTS

Thanks to everyone who has helped and guided me to this point of my life and education. Special thanks to my family, friends, loved ones, and every teacher and professor who patiently guided and worked with me. Thanks Dr. Plante and Jessica, for believing in me and bearing with me. Deepest thanks to my Father and my mother.
TABLE OF CONTENTS

ACKNOWLEDGMENTS 2
TABLE OF CONTENTS 4
LIST OF FIGURES 5
ABSTRACT 6
  1. INTRODUCTION 7
  2. Related Work 11
  3. Implementation 17
  4. Results 27
  5. Conclusion 35
  6. Future Works 37
REFERENCES 39
LIST OF FIGURES

Figure 1. Number of files per commit for *Hystrix* 30

Figure 2. Number of files per commit for *SlidingMenu* 31

Figure 3. Number of files per commit for *Picasso* 32

Figure 4. Number of files per commit for *Json-simple* 33

Figure 5. Number of files per commit for *Catacomb-Snatch* 34

Figure 6. Number of files per commit for *Bubble* 35

Figure 7. All methods of comparison per commit for *Bubble* 35
ABSTRACT

In theory, software engineering can be described by clear and systematic processes akin to those defined by the hardware engineering community. In practice, however, software development is a messy and error-prone process that may differ greatly from those described in texts. How code is actually developed in practice is not very well understood, varying a great deal from one development team and project to the next. Understanding how code development evolves in practice may aid in understanding how processes can be improved not just in theory but in the field. Using a revision control system to track the evolution of code for projects considered to be very well developed may shed light on this problem. In this work, we use the distributed version system, git, to track the evolution through time of well-developed projects on Github to see how the actual development process compares to software development theories. By checking out successive commitments, parsing the source code, and tracking various metrics for each commitment, quantitative measures will be obtained and analyzed.
1. INTRODUCTION

Textbooks on software engineering often described clean and sometimes formal processes for how software is or should be developed. In reality, it is well known that the practice of actual code development is not nearly as clear and well defined as such texts would suggest. Not only can the real development process be messy, it is also widely unknown just how it progresses in practice. Fortunately, we are now able to get a glimpse into the development process with the means of Source Version Control Software, such as git.

Having a means to keep track of code evolution has become an essential part of the development process, just in case errors occur and the need for a previous working version arises. Source Management Systems are a very powerful means of doing so. Git, the tool I am using in this project, is one of them. The wealth of information Git provides and how Git treats the information allows developers to keep well organized versions of their source code from start to finish. Not only does it allow developers to keep record of the software process through time, it also allows us to observe the development of a software project and to see what code smells appear and how they are resolved.

Code smells are an example of a code development product that makes it very messy. They are very problematic and can cause widespread issues in software if not managed. These issues are not as noticeable as syntax errors in languages, making them even more problematic if not detected. This concept was first coined by Martin Fowler
and is defined as a surface indication that usually corresponds to a deeper problem in the system [6][11]. These smells can occur anywhere in software projects.

In order to detect a code smell, one must be aware of common issues that occur when programs are being developed and changed. Research into code smells have resulted in some very interesting means of detecting and fixing such issues. However, it needs to be noted, that although there are automated tools designed to detect code smells, it is still best practice to be done by hand[6]. Now with the huge amount of code that is generated, it will be very difficult for a person to inspect and detect smells, so the automated tools are used to aid in the detection of code smells. Some products of this research can be seen in Jdeoderant, HIST, and DECOR.

Git is a Decentralized Version Control System (DVCS) that stores information as snapshots of the file through time. Git is designed to be fast, efficient, simple, and light in comparison to its relative VCS. It is open source and free to the public under the GNU General public License. Since its conception in 2005, git has become increasingly popular in the version control market. Many noteworthy companies and projects are using git for version control, such as Linux, Facebook, Android, Google, and Microsoft. It is designed for speed, lightweight, efficiency, simplicity, and non-linear development. Git retains these attributes among large projects by the way that it handles information. It is the methodology behind git and it’s information handling that separates it from its competitors.

Git handles information as snapshots. This is a crucial concept that sets Git apart. Other VCS look at files as changes overtime. Instead of looking at information as change
in the file overtime, Git takes a picture of the file system each time and stores the picture, or snapshot, at that particular time and stores references to the snapshots. If there aren't any changes in the file, git will store a reference to the snapshot. Other VCS look at data as changes in files and if there are no changes, they will store the same copy again, an unnecessary redundancy. Git maintains integrity among the many snapshots by hashing the identifier that is attached to each commit. The identifier associated to each is unique.

Git operations are local. Git makes complete backups of its repositories and stores them locally, providing for operations can still be performed when not connected to the main repository server. This allows for quality work to continue, even if there is not a wireless connection unlike other VCS. This allows unlimited access to the repository and its content. One can even place a repository on a portable drive and carry it wherever they desire. For other VCS, a connection to the main server is required, even if the repository is stored locally. If a connection is not established, or if there is a poor connection, the amount of work that can be accomplished is severely limited, if not halted.

Code metrics is an example of an automated tool used to detect code smells. Although code metrics can give a programmer a feel for the type of code they are working with, they are not really effective at detecting smells in code. They are also not an effective means of determining software quality. With this in mind, we still intend to use metrics to provide a quantifiable means of comparison in software projects as they progress from earlier states to latter ones. We intend to analyze some notable open source projects to view how code development occurs overtime. Some of the metrics we
intend to use include Physical Lines of Code, Halstead Metrics, Cyclomatic Complexity metric. In addition, we also intend to tally the amount of classes, methods, if statements, switch statements, and comments, so as obtain a measure of a repository and its different files from commit to commit. We are using Antlr4 and git commands to get the information we desire out of the git repositories we will be looking at.

    Git has many other features that separate it from alternatives for SMC, but there is a reason we selected git over its competition. Git was selected over other SCM because of its increasing popularity. Since its conception in 2005, git has continued to become widely used. Many other teams are also migrating from other SCMs to git. Despite its growing popularity, there is a lack of research into mining git for information. We selected git mainly because of this lack of research into mining information from git. Also, git is unique in how it treats information and the way its users are able to interact and obtain information from it. Part of this research will involve mining information from git, then sifting through that information and analyzing it.

    To accomplish this, we will be using git commands, such as “git diff” and “git log,” to mine user contributions from open-source project that utilize git for version control. We will also be using parsing programs and code metrics to further analyze and quantify how the code evolves overtime. Some popular metrics include single lines of code (SLOC), Halstead and Complexity metrics. These metrics will provide us with a quantifiable means of measuring code size, complexity, and length as well as a means of comparison in and amongst projects.
2. RELATED WORK

As mentioned earlier, there is a great amount of information contained in repositories. We have found several research papers that have attempted to mine and organize this information in a variety of ways. Anderson's [5] group uses commit messages as the means of obtaining this information, while Allamanis [12] approaches it via language models. They both are accessing repositories, but different parts of the information contained in them. Anderson [5] focuses more on the messages and creating a means of identifying the type of information contained by keywords in the message, while Allamanis [12] focuses more on the information obtained through analyzing source code, via natural language models. There is also research models that can be described as hybrids of their approaches.

Carlsson [3] presents a new tool of obtaining all of the information from git repositories, via their tool, DORIS. There are also others that focus mainly on tools to generate a means of detecting code smells through the information contained in the repositories by analyzing files from commit to commit [9][10][4]. These project aim more at helping developers detect code smells within their project, so that they can be resolved before they cause bigger problems. All these research papers provide great insight and novel ways of obtaining information from git, but we noticed they all shared a similar characteristic.

In most of the research we have looked at, the focus was getting that information
from git. But we have not noticed any that seek to explain how the information obtained
describes the actual process of development that went into that project, or how the
process usually looks amongst many projects, besides Anderson [5]. We are more
interested in learning about what the development process looks like and how the project
changes through time.

Purpose

The goal is for this project to show us what does the development process look
like in the field? In this research, we will look at several noteworthy open-source git
repositories to see the process overtime and answer several questions that will give us
greater insight into the development process. In each repository, we will look at if the
development process starts of with huge lines of code in the beginning and, if so, if that
code tends to move from a very big, messy, initial state to progress to a better and more
finished product. In addition, we will look to see how many problems and smells in code
going better as the project progresses and also how much code is removed and inserted from
commit to commit.

After we do this, we will look at our results as a whole and decide if student
programmers being taught to make small changes and test them before integrating them
into a bigger project is a common practice in the field. We will be able to see if small
changes are introduced bit by bit, from commit to commit, or if most changes tend to be
big ones that occur quickly. Upon completion of this project, we will be able to show
how the development process actually looks like from the beginning through all of the
work necessary to reach the finished product. We expect that code will initially start off
in a relatively messy state and as time progresses, with numerous corrections, will
become a more tailored finished product.

**Tools Investigated and Used**

Parsing is vital for many reasons. One good reason is that it gives us a means to
compare and analyze code. We intend to use a parsing tool to calculate how many
functions, properties, methods, classes, variables, and language specific properties are in
a project. With this, we can use the output to determine different metrics, such as single
lines of code, complexity, etc., and then be able to provide a means of code comparison,
whether it is between different versions of the same code or different projects. A Parser
is also useful in that it allows for us to capture structured information from git hub and
store them in databases for later analysis.

Some of the most common issues we have observed during this research are
documentation and installation. There are a large amount of tools available for use, but
the amount of time required to look through them all is not feasible for the time span on
this project. To minimize the amount of time spent looking for a parser, we have limited
ourselves to a select few, based on comments and reviews found. Essentially, we were
looking for a parser that works, is user-friendly, has a relatively small learning curve, and
has minimal hurdles to overcome. The tools we have focused on are Bison & Flex, Antlr,
JavaParser, Codeworker, and Ctags which are all described below.

Bison & Flex are actually two separate tools. One is a lexical analysis tool and
the other is a syntax analysis tool, or a parser. We were able to get a working example of Bison & Flex to parse through a file. Bison & Flex was a possible option, because it provides us with the information we need from different code. All of these different tools have their own learning curves, and Bison & Flex is no exception. Bison & Flex seems relatively user-friendly, however, one must have a decent understanding of C/C++ in order to write Bison & Flex parsers.

Java Parser is written in Java and from our research, is used to parse only Java code. The libraries are well documented and seem to be fairly easy to use. However, in order to use this parser, one has to be able to install it with maven. We do not have prior experience using maven and are currently learning it so that we may test this code. We have had some issues in the installation process, but we hope to overcome them very soon. In the mean time, we have went ahead to test other parsers.

Codewerker has been around for sometime and was also a potential candidate. There is an eclipse plug-in for this tools as well, but the tool seemed to be outdated, with its last update in 2010. Ctags was very simple and easy to use and learn. It seemed like the ideal candidate until we looked further into the results it returned. It is actually an indexing tool, not a parser. Ctags is able to return all of the different classes within a given code, but it did not give us the detailed data we desired. As such, we decided that this tool is not a viable for parsing.

Antlr is a new parser and ended up being the tool that we might decide on. It is written in Java and also has target for C#, python, and other languages. This parser has received good reviews in comparison to its counterparts, such as Bison & Flex. The
structure of the grammar rules utilized in Antlr are similar to that of Bison & Flex. One thing that gives this tools an edge over the others observed is its newness. The other tools are much older and do not seem to be maintained as well as Antlr. Antlr is also very simple and well documented, compared to some of the other tools we have looked into. Listed on Antlr's site is a link to a repository of different grammar scripts we can use to parser through different languages. Once we got this parser to work, it became clear that this would be the one that we used. It also offers plug-gins to integrate it with IDE's such as Eclipse, IntelliJ and Netbeans. This tools also has a stand alone IDE, AntlrWorks, where one can create grammars to parse languages and display the information in the form of a tree.

The above tools were all parsers, while these next ones are other tools that were necessary for the implementation of the project. Cloc is one of those tools that we also looked at. Cloc is a physical line of code counter. Our intent was to use it to generate the counts for different files. Cloc is a script written in Perl and requires minimal installation. This tool seemed very promising because it was able to generate physical lines of code counts, comment count, and blank line counts. It was then able to export the information into different formats, such as a text file, SQL statements, CSV files with you own specified separator, and much more. This tool can be ran on many different language and can be extended to work on many more. However, we saw that it did not give us enough information. We also wanted to count conditionals, loops, and other different types of statements, but it is not able to do this. Although we decided not
to use this tool to generate LOC counts, we still used it to test against the counts our program was generating against different files.

DORIS is a data mining tool designed to mine information from git repositories. We have searched for different tools that may be of benefit, but the tools we encountered seems to be using git commands and returning the information in its own special format. One tools that we look in depth at was DORIS. It does the job of mining information, but the information received is the same obtained from using basic git commands. We have decided to use basic git commands, such as *git log*, *git diff*, and *git checkout*, to get project data from open source repositories. Part of the reason is that DORIS is currently not able to mine information from git repositories without an Internet connection. Also, by using git's commands, we have more freedom in ascertaining specific information we desire.

What we are mining from git repositories is the snapshot of the project at each time. To do this, we will be using git commands. In our research into how this is done, we saw that there aren't many tools that do so. One tool that seemed to be very promising in doing so was DORIS. DORIS is able to successfully mine information from git, but upon further inspection, it seemed to be doing so via common git commands, such as log and diff, then present the information in XML format. Also, we noticed that this tools requires Internet access in order to function, which can prove to be an issue for those times when no Internet connection is available.

So instead, we decided to just use git log, git diff, and git checkout commands to get the information we want from git repositories. Much of the other research into
mining git repositories tend to use these commands as well or a GUI to also get information. We believe that the commands associated with git are already powerful and give a wealth of information. Hence, we agreed upon using these commands because they can easily tailored to specific need. DORIS may still be of use, but we have decided on writing scripts to get information in the mean time. We will be running these commands via a scripting language, such as python. To handle parsing, we require a parsing tool (explained in detail above) to parse through the source code and retrieve only the elements that we are interested in.

3. IMPLEMENTATION

For this project, we have used Antlr4 to create a Java parser to sift through code for the parts we want. To do this, we used the Java grammar file, found on their github repository [21]. We made some minor alterations to suit our needs to make the process of parsing cleaner and easier to handle. Once we have obtained all of the parts we want, our next step is to generate statistics on that information, then store it in a database, for later analysis.

Currently, our parser works only on Java code. Our parser reads in a file, goes through the different rules we explicitly call. Once the parser enters a rule we would like to analyze, we then begin to count lines of code, comments, and the different statements we are interested in. To check and count for these different parts, we have made the class Stats separate from the parser to do so. This decision was made to separate this crucial
part of our parsing process to better test and develop our parser. With the parser generated by Antlr, we focus on a few different parts of the Java source code. We look at the overall file, classes, methods, functions, comments, conditionals, loops, and exceptions.

In Antlr4, we are able to specify channels to send specific segments of code or characters to. Without specifying the channel, tokens would automatically be sent to the default. We decided to make three channels to send specific tokens to. These channels are Comments, Whitespace, and NewLines.

On the comment channel, we are sending both single and multiple line comments to the same channel. In the grammar file, a single comment is defined as a string preceded by two successive forward slashes, “//” and is terminate, but does not include, a new line char. A multiple lines comment is very much similar to a single line comment. The main differences are that the comment is preceded by a forward slash and asterisk,’/*’, and is terminated by an asterisk and a backslash “*/”. In Java, there are also Javadoc comments that are very similar to multiple line comments. The grammar file is written to group Javadoc comments and multiple line comments together. We place all comments on the same channel, so that we could separately count through them.

Whitespace is then dedicated to holding all form of blank space in the target code, except new lines. This includes tab character as well, “\t”. At first, the grammar categorized whitespace as every blank space and new line and skipped over anything that fell into this category. We decided to place whitespace in its own separate channel, to preserve their place in code lines and to prevent remaining character form being fused into one big line of code. This proved to be very handy later on in our analysis. We will
speak more on these different channels and how their contents all play their part in the larger picture.

On the new lines channel, as its name states, this channel is for characters that match the new line character \"\n\". It is of value to note that this character is different in Windows, where the return character \"\r\" is used. If this token is not stated in the grammar, there will be occurrences of new lines not being recognized when the parser is ran on a Windows machine. It is generally best practice to incorporate this character in the grammar definition. The reason we made this channel was to keep track of the new lines in the target file, since we are measuring physical lines, blank lines, and comments of code. By doing so, we are able to identify the content of each line as it is being processed and sifted through by the parser.

In our code, we define physical lines of code as lines where there are code statements. In our definition, we count anything that is not a comment and not a blank line, as a physical line of code. By this definition, we also include grouping symbols that are on their own lines as well. By separating these characters out onto their own channel, we are better able to separate out each line's content by the parser.

Our Stats class takes three parameters: A token stream, the start token, and the stop token of the rule we have entered. The Stats object then goes through the token stream and recomposes the program bit by bit. Once we have reconstructed the different parts of the code, we use the tokens to check for lines of code, blank lines, etc. To check for lines of code, the string we are inspecting must pass certain criteria. It must not contain any new line characters, can not be an empty string once the string is trimmed of
extra whitespace characters, and can not be part of the comment channel we spoke of earlier.

If the token passes all of these criteria, we then add their text to a string. The string is then trimmed from any leading and trailing whitespace (including new line characters). The counter is incremented and the string is emptied, to prepare for the next line. We implement a slightly different method to tally the lines of comments.

Using Antlr as a parser proved to be very useful, especially in keeping track of comments. Comments can be a little more tricky to keep track of, because they can appear anywhere in files. Antlr made this task very simple with the use of channels. This allows us to send specified tokens to the channel we desire. We send comments to their own channels and once a token's channel matches that of the comment's channel, we then count that token as a channel, without too much string manipulation. Anything that is within the start and stop criteria of a comment is grouped as one token. Single line comments span across an entire line, so they would be counted once. Multiple line comments can span across many lines.

To count the lines consumed by these comments, we used the Java String split() function to separate a multi-line comment by the new line characters within. The split function returns an array of strings where each index in the array corresponds to a string. The only part of this array we are interested in is its length. We then call Java's array .length function to get the length and add it to the comment counter. This basically describes the basic counting done in our code. We then do this to gather information for all the parts of the target source code we are interested in.
With our *Stats* class, we then use the parser to check for entry into the different parts of the target source code we are interested in. In each of the parsing rules we are interested, a new object form the *Stats* class is created and we then have that object run its count function on the current rule that has been entered. This continues until the end of the file. To get the specific parts of the source code we are after, we implemented an extension of the Base Listener file generated by Antlr. After we get the information that we need using the Antlr parser. With this information, we intend to use this to gain a measure of how code changes overtime.

*Python & other tools used in this project*

To handle the transfer of information between Git and our Java parser, we are using Python. We have chosen Python because it is the scripting language we both share with the most comfort. We are mainly using Python to access the Git repositories in question, pull out only Java files in each commit, run the Antlr4 parser on the specified file, take the results, then store them into our MongoDB database. Once the information is stored in our Mongo database, we then retrieve it later on via queries and analyze the information to find patterns between the different types of repositories that we have selected. To help with the transfer of information from place to place, we have also introduced three aditional tools: Pymongo, json-simple, and GitPython. With the exception of the json-simple tool, GitPython and Pymongo are Python libraries; json-simple is a java library. To communicate between Java and Python, we are passing information in JSON format and through operating system calls. We have divided this process into several steps.
Information flow: Python, Git, Antlr4 parser, MongoDB

First, through Python, a Git repository is set and selects the very first commit. Once the repository is set, our Python script goes through and looks only for Java programs one by one, then sends them to the Antlr4 parser. The parser then parses through the file and stores the information as a JSON object with the help of the \textit{json-simple} tool [16]. Once the parsing operation is completed, the parser returns a json-string object to the Python script. From Python, we take that json-string object and add file specific information to it, such as the file name, author information, commit sha, commit number (0..n). Then, we establish a connection to our Mongo database and store the modified JSON string object directly into our database. Once the operation is successful, this process continues until all commits have been visited, or until we stop the process. We then populate our database with this information.

As mentioned earlier, we did integrate other libraries into our project to aid in the transfer of information from one place to another. Hence, we agreed upon using JSON as the means of sending information. We chose JSON because it seem the most simple at the time. Java and Python both have libraries that easily support encoding and decoding JSON formatted information into each respective language. We looked at several different JSON libraries for Java that would be easy and quick to learn, as well as simple to install. Some of the tools we looked at included Jackson and GSON. These tools had great reviews and were noted for being well known and trusted. However, there was an associated learning curve and dependencies attached to them that encouraged us to
continue in our search. Ultimately, we choose the *json-simple* tool, because it satisfied the conditions that we were looking for in a Java JSON library.

*Json-simple*, as it name states, is simple to use, it does the job needed, and it required no external dependencies in order to run or integrate into projects. *Json-simple* is a stand alone jar file that is well documented. From this tool, we created three different classes that would serve as containers for the information generated by our parser being ran on a file. We divided a file into three different parts, each described in detail below. The file itself, the metrics derived from our parser on the file, and the members, or parts of the file (methods, classes, etc.)

Our *File Object* class is made to be a representation of the file and its specific information after the parser has been called on the file. This class is an extension of the JSON object provided by JSON simple. The *Metrics* class is used to create an object that holds the information generated by our *Stats* class. Objects made from this class takes *Stats* object and store the information. In essence, the *Metrics* class stores and translate a *Stats* object into a JSON format for latter use, when the parser returns the File object back to our Python script. Both our *File* and *File_Item* classes can have a *Metrics* object.

The *File_Item* object represents the different parts of a Java program. For example is a Java file contains a class with seven different methods, the *File_Item* class can be used to represent the class, constructors, and methods of the file. The reason we also store the file as a *File_Item* is due to the fact that a Java file has the potential of containing multiple classes. Storing classes as file items has the possibility of redundancy, however inflation that can result from this can be easily corrected for.
Objects of these types are stored in an JSON array that holds the parts of a class. Once we defined the individual parts of the file and how to store them, creating the means of storing the information was not too difficult.

**Storing information**

Once we were able to format the information generated by our Antlr4 parser, it was now ready to be stored into our database. We are using MongoDB, because it proved to be very simple despite having little experience with it. Because of this, along with the documentation that can be found on the site, we did not have a difficult time integrating Mongo into our project. However, there are limitations as to what we can do with it. In the end, we had to revert to an earlier version of Mongo due to the system we are running.

The first issue is support. We are currently using version 2.6.12. We are running this project on Linux Mint 17.2, Rafaela, 32-bit system. Versions of MongoDB from 3.0 forward are discontinuing support for 32-bit systems. We do not have access to the newest features and the new search engine introduced in the later version. The use of an earlier version of Mongo is recommended on their site if one is running a 32-bit system. Second, the version of Mongo we are using has been depreciated. With the current system we have, a database can only hold at max, 1GB of data. This memory limit may also be smaller than what is written due to hardware protocols and resource use.

There are many benefits to using Mongo as a database for this type of project. It is well supported; it does not require a defined schema to store information, allowing
greater flexibility; and it is not too difficult transferring one's knowledge of SQL databases.

The flexibility provided by using a NoSQL database provided to be very useful and time saving. There were several times that we altered the structure and the data returned by our Antlr4 parser. With Mongo, we did not have to rewrite the database nor change anything in our Python program to account for these changes. With MySQL, we would have to reformat the database schema for every change. In addition to the flexibility, the documentation presented on Mongo’s website was very instrumental.

Mongo’s documentation has been very helpful in transferring our knowledge of SQL databases over to NoSQL databases [14]. This eased our concern of how easily we could retrieve information from Mongo in comparison to MySQL. The site is easy to navigate, well maintained, and also has a community forum where questions can be answered.

In order to transfer information between Python and MongoDB we are using PyMongo. PyMongo is a Python tool used to interact with MongoDB. With this tool, we created our own class named \texttt{MongoConnection}, as a basic means of handling queries, database selection, inserts, and deletions.

\textbf{Selecting Repositories}

In the selection of repositories to use in this research, we recognized that we would need some that are well known. We decided to use a total of six repositories: three noteworthy and three relatively unknown. Finding noteworthy repositories were not
difficult, however, finding those that would be their opposite was. We termed these other repositories as the *other*.

To find noteworthy repositories, we first searched github to see what we could find. We started our search on github by searching based on most forked and most stars. From there, we also searched through google to find other person's opinion on good and bad repositories. In our search, several of the top repositories that we found on github appeared many times with honorable mentions as noteworthy.

Finding reviews on projects that were considered bad was difficult. Since we did not find what he hoped for, we decided to use repositories on github that were not well known to use for the comparison. This does not means that they are bad, but it did result in extra research into the projects we selected. We also went through Java files in the repositories by hand to see wether there was sufficient documentation, structure. This was done so that we can establish a difference before comparing repositories.

The repositories that we selected to be our *noteworthy* are Sliding Menu, Picasso, and Netflix Hystrix. The *other* repositories are as follows: Bubble, json-simple, *Catacomb-Snatch*. In selecting these repositories, we recognized that our hardware limitations had to be taken into account. First was size. Through github’s api [], we searched for repositories that had received the most stars, the most forks, and were of a specific size. We set our search on repositories that were of 30MB in size or less. This was done to assure that our repositories would not generate too much information for the Mongo databases we are using and to also ensure that they would not take too long before
all of the Java files we are looking for, would be parsed. The next criteria was popularity.

To find the noteworthy repositories, we search github based on most forks, stars, and size of 30MB. Sliding Menu, Picasso, and Hystrix were among the very top appear based on this search. Using the given criteria, finding noteworthy repositories was not difficult. However, finding their opposites proved to be very challenging. Finding the other repositories was not as easy for several reasons. One, they had to be of a certain size. When searching Github for least popular repositories, it takes into account every repository hosted on github

4. RESULTS

In our analysis we decided to gather averages from each repository for the number of files per commit, lines of code per commit, number of methods and classes per commit, conditionals per file and per commit, lines of comment and code per file and commit. Through these different statistics, we are looking for patterns to help identify how code changes through time. By looking at code itself, we believe that trends and patterns can be seen in the development of software. Through these different metrics, we hope to see what are the actual patterns that occur overtime during the development of code.

For our analysis, we decided to analyze the first 100 commits. We decided on a set limit for all of the repositories we analyzed to help with comparisons. Some repositories had 300 while others had over 1,000. Also, due to the hardware limitations we are facing, we saw 100 as a sufficient number where not too many issues would
appear in terms of speed and repositories of different sizes would not take too long to analyze.

**Overall patterns**

In all of the repositories we analyzed, we looked at the number of files per commit. In this measure, we saw that most of the repositories started at very low points compared to other commits. As the total files per commit increase, so do other statistics such as comments, total lines of code, and classes methods. In our noteworthy repositories, we saw that all but Hystrix started off as relatively stable when compared to the first few commits. Hystrix showed patterns that were different from the other repositories analyzed.

**Noteworthy Repositories**

In *Hystrix*, there were several commits pushed to github where there were no Java files. This happened at several different times through the history of this project. At first, we thought that some error occurred during parsing and extracting, so we went back and checked the commits where these holes appear. and found that it was the same. This repository started off with a few files, then dropped to zero twice (Figure 1). The first time, it stayed this way for one commit. This first drop occurred at commit 8. The second drop took place at commit 10, and remained at this point for 18 commits. At commit 29, there is a sudden spike in the number of files as they jump from 0 to 74 files at once. From this point on, increases occur relatively stable in comparison to this earlier spike. In contrast, we saw an opposite trend in the *SlidingMenu* repository.
In *SlidingMenu*, there is a sudden spike at the beginning followed by a sudden drop (Figure 2). This repository goes from 0 to 155 files from commits 0 to 4. The repository remains at around 155 files from commit 0 to 67. At commit 68, the repository drops from 161 to 105 files. The repository drops again and remains relatively constant after commit 88. After this commit, we also see that there aren’t many more changes in the repository and the repository stays at a constant level. The last noteworthy repository we looked at in our research is *Picasso*. 

---

**Figure 1. The number of files per commit for Hystrix**
Figure 2. The number of files per commit for SlidingMenu

In Picasso, we saw that there were many additions and deletions throughout development, but in the overall picture, growth tends to be the common trend (Figure 3). Picasso, similar to SlidingMenu, spikes quickly within the first 5 commits. As we go through commits, we also see that there are several other spikes and drops. Unlike SlidingMenu, Picasso recovers and even surpasses its relative maximum. A similar increase is also seen at the beginning of Hystrix, where development seems to be stagnant for several commits, then suddenly spikes and continues to move upwards.
In summary, there were different trends present in the three repositories we selected as noteworthy. *Hystrix* showed sudden spikes and continued moving up, with few points of plateau, while *SlidingMenu* quickly increased in a matter of four commits, remains at the same high level, drops, and then slowly rises again. In *Picasso*, growth was a common trend. This repository shared some notable characteristics of *SlidingMenu* and *Hystrix*. *SlidingMenu*, in that *Picasso* spikes early and does have points that are like holes in the repository. *Hystrix* in the sense that growth was the common trend.

**The Other**

*Json-simple* is much smaller than the other repositories we analyzed, with a total of 29 commits, since it is still in early development. Compared to the other repositories we looked at, *Json-simple* is the smallest. We selected this repository to see if there was a pattern in its development, even though it is relatively new. Unlike the noteworthy repositories, *Json-simple* is not an Android project. It is a java tools used to encode and
decode JSON objects. This project is also characterized relatively stable growth (Figure 4).

Figure 4. Number of files per commit for Json-simple

*Catacomb-Snatch* shows a general growth trend with editions and deletions from commit to commit (Figure 5). This repository is most resemblant to *Picasso* in trends. Like *Picasso*, *Catacomb-Snatch* continues to show relatively constant growth overtime, however, this repository did have a non zero commit where the number of files in the repository was zero. This point occurred at commit 95. Another significant point in *Catacomb-Snatch* is commit 134, where the number of files suddenly spike from 63 to 85. In the following commit, the number of files drop down to 66. Other than these two points, development is pretty constant, similar to the trend seen in the *noteworthy* Picasso.
Figure 5. Number of files per commit for Catacomb-Snatch

*Bubble* is characterized by many spikes and drops (Figure 6). Despite its many ups and down, what stood out about this one was number of files in comparison to the other statistics we measured (Figure 7). The statistic that resembled *files per commit* was the *class per commit*. There is a constant trend of growth among files, comments, and total lines as the number of files spike and drop. Every stat drops at commit 113. There are four significant drops before commit 113, but there is not a significant change in these stats before this.
After we analyzed all of these repositories, both the noteworthy and the other, there was not a major distinction based on the statistics we analyzed. Each repository had
its own unique characteristic, but at the end, there was not much of a difference between the *noteworthy* and the *other*. In all of the repositories, we saw that most of the java files contained or were Classes. There were few interfaces and even fewer enum and annotation files. We also saw that all of the repositories, both *noteworthy* and *other*, shared one common trend: the respective files and packages numbers were identical. So, were the annotations and do statistics. In the case of *annotations* and *do*, these statistics were the same in each respective repository because they were both zero. The *files per commit* and *packages per commit* were identical.

5. CONCLUSION

The software development process varies greatly from one team to another. In this research, the variation was made very clear. With the use of several different tools, we attempt to find a means of measuring the process with the hopes of finding patterns that emerge amongst two different categories of repositories. In choosing the repositories, we had to keep in mind the limitations of the machine we are using. So, we selected repositories that were at most 30 megabytes in size. From there we selected our *noteworthy* repositories using Github’s search api. Once we had our repositories, we then proceeded to parse through each repository, commit by commit, to see what kind of results we would find. In this research, we did not find much to distinguish the *noteworthy* from the *other*. 
A common trend we saw was spike and drops during development. At one commit, there can be few files and few statistics and in the other, there would be twice as much as before. During our research, we saw that the statistics we observed were very much similar in appearance. Due to these similarities, we decided to exclude graphs that would not provide any additional information. We also noticed some interesting occurrences in several of the repositories we analyzed. In the noteworthy *Hytrix* and the *other Catacomb-Snatch*, we saw points where the measured statistics dropped to zero. This is seen more presently in *Hytrix*. *Hytrix* remained at a zero point for several commits, then suddenly spiked. Each of the repositories did display trends of spikes and growth. From this, we can see that the development process can be very messy with much of the code development occurring at once, and is then being tailored.

In many cases, once the code is added, it is then tailored. More is later added as the process continue. The repositories where there were large additions followed by large deletions could possibly be a sign of some type of Rapid Programming. Others where the changes were more tame can be a sign of Incremental Programming. From these results, it looks like the theories that students are being taught about developing code are actually similar to those used in the field.

6. **FUTURE WORK**

Future work for this research would be to extend the reach of our parser. For example, in the stats we calculated, we looked at the single lines of code (*SLOC*), instead
of logical lines of code (*LLOC*). To obtain a better sense of the evolution of code through
time, using LLOC as a means of quantifying change and growth can be of more value
than *SLOC*. *SLOC* only measures physical lines of code, while *LLOC* looks at logical
lines. There is a factor of inflation in using *SLOC*, however it can be adjusted for. To
better grasp changes that occur, *LLOC* can be of more use, since it looks at lines of code
that have logical function.

Another future extension would be to extend the parser to analyze many more
languages. Due to time constraints, we focused mainly on Java files, however these do
not capture the full magnitude of development in a project. Some projects encompass
many other languages and file types, such as XML, and different shell languages.
Extending the Antlr4 parser is possible. There is a growing repository filled with
grammars targeting different languages and file types. Extending the parser should not
require much change to our source code. Capturing the token stream and calling the
correct rules would be the main issue. Another extension we would also like to see is the
incorporation of a larger variety of metrics, to better quantify and view the changes seen
in the development process.

Metrics are a very useful means of grasping and quantifying software. In our
research, we saw that metrics have been around for many years and is not a new thing.
However, they are not widely used, despite their age. Metrics such as Complexity and
McCabe Cyclomatic would be useful. They can help get a better view of how code
changes overtime and also how complex it becomes. Ultimately, there are no limits as to
how far this project can be taken. We hope to spark an interest in trying to find some
means of measuring code development. Not only would it reveal to us how the
development process changes overtime, it can also help us identify good and bad habits
in the process and improve from there.
REFERENCES


code and design smells." *Software Engineering, IEEE Transactions on* 36.1

10. Fokaefs, Marios, Nikolaos Tsantalis, and Alexander Chatzigeorgiou. "Ideodorant:
Identification and removal of feature envy bad smells." *Software Maintenance,


massive scale using language modeling." *Proceedings of the 10th Working


14. MongoDB: SQL to MongoDB mapping chart

   [https://docs.mongodb.org/manual/reference/sql-comparison/](https://docs.mongodb.org/manual/reference/sql-comparison/)

15. *Catacomb-Snatch* github repositior

   https://github.com/Catacomb-Snatch/Catacomb-Snatch.git


17. *Bubble* [https://github.com/nkanaev/bubble](https://github.com/nkanaev/bubble)

18. *SlidingMenu* [https://github.com/jfeinstein10/SlidingMenu](https://github.com/jfeinstein10/SlidingMenu)


20. *Netflix Hystrix* [https://github.com/Netflix/Hystrix](https://github.com/Netflix/Hystrix)