Git Writeup: Sagar Laud

Git is an open-source program created for the purpose of creating a Version Control System (VCS) that acted in compliance with the Linux kernel. Initially, Git was developed for Linux developers but has since expanded its scope to developers utilizing all platforms. That being said, it can now be used by anyone who wishes to program with others. Initially, the Linux kernel was unusual and because of the large number of committers as well as the large number of contributions that were taking place, the core development community had a significant amount of difficulty finding a Version Control System that worked well for them. This however, was not enough for a solution to the problem to come around. An incentive came soon after however. At this point in time, the Linux kernel codebase was serviced using two different forms of Version Control, BitKeeper and CVS. After some time, BitMover (the creator of BitKeeper) announced that it would revoke the licenses for several core Linux kernel developers. It was at this point in time that Linus Torvalds (the creator of Linux) started working on a solution to this problem. His first goal was simply to have a collection of scripts that would allow one to easily manage email patches to be applied in succession. This set of scripts was to be used simply to allow a user to abort merges to allow the person maintaining the codebase to modify it “mid-patch-stream” to merge several patches at a time. It was Torvalds' intent to offer several safeguards against data corruption in conjunction with better performance than ever before.

Git itself utilizes a peer-to-peer architectural style. A peer-to-peer architectural style relies heavily on the fact that different clients can collaborate and view common information without the need for a centralized server. The beauty of this versus another type of architectural style (such as client-server setups) is its simplicity. It does not require for a central server to exist! Rather, it utilizes several different clients as “peers” to allow all members of the peer-to-peer network to have access to each other's files (rather than each person owning private files). Yet another advantage to having a peer-to-peer network versus a client-server model is the fact that in the latter style, there is a clearly defined supplier and consumer. What this means is that in the client-server style, the server is the only part of the network capable of sending information. Likewise, the clients are the only part of the network are clearly defined receivers of information. This is not the case in a well defined peer-to-peer network simply because it eliminates the need for a central server as a result of how it keeps/stores information. As mentioned earlier, each and every user/client is classified as a “peer” and treated as a node. These “peer nodes” combine the qualities of both a server and client to become true peers. Rather than having certain portions of the network be designated as a server (an information sender) while other portions are designated as clients (information consumers), the peer-to-peer network treats each and every peer node as both a server and a client (making them perfect for a VCS). This allows for quick information storage and provides the capabilities for all users to collaborate as equals.

That being said, if I had to choose a style for creating Git, I would have chosen a peer-to-peer network as well. The reasons for this are obvious and are related to the issues that Torvalds strove to remedy as well as the function that any form of VCS was/is supposed to perform. The purpose of a VCS is to allow several users/clients to collaborate on a project! For this reason, employing a client-server style would be rather inefficient as not all of the users would have the ability to commit information to the repository. As a result, it would become very difficult for several clients to collaborate and the purpose of Version Control would be lost. For this reason, choosing a peer-to-peer architectural style is, in my mind, the correct decision. This architecture style allows several clients to collaborate collectively without any restrictions and is truly the best way to employ a VCS.

For version control, the amount of information that is present at any given time can be massive. It is imperative that a VCS does a good job of packaging these files so that the absolute minimum amount of space is used (therefore allowing more information to be stored for clients to use). For this paper, I will explain both how Git's source code is packaged and how Git packages other files that are committed. Git's source code actually does not utilize packages. Because it is coded in C, actions are determined simply based upon the filename (in the style of C projects). All of the Git files are in the

top level directory and all have descriptive filenames. These source files are coded for maintenance as well as reuse. It can be considered as maintenance as an individual file in the source code could be modified but can also be considered reuse because a plug-in can take each and every file in for use. Git packages files that are committed based entirely upon when they are committed. Meaning, each file that is committed is packed individually as a GitObject(see Figure 1). What essentially happens is that each set of code that is committed is all actually under the Commit class once it is pushed (see Figure 2). Git then stores all code in a Directed Acyclic Graph (DAG). Each time code is pushed, a new DAG is created. The histories of each node file in the DAG is linked all the way up its hierarchy (using nodes that represent each and every directory) all the way up to the root directory. This root node is then linked to the commit which in turn marks the root of that particular DAG. Each commit node can actually have other parents (parts of other DAGs) which, when looked at in detail actually represents the hierarchy of the entire project. Tag classes then point to the commits. This is how Git packages committed code. It groups each bit of code into different classes which are in turn converted into a DAG. This DAG is then pushed to the repository and merged with the current DAG.

At this point, an analysis of different aspects of Git are in order. The first thing that needs to be analyzed is simply the actual architecture of this VCS. It is my opinion that a peer-to-peer structure for Git is the perfect structure for the job. I don't believe that any other structure will necessarily work as well for a VCS simply because its purpose is to allow several “peers” to work on a network and collaborate with one another. A peer-to-peer network accomplishes this beautifully by allowing all of those users who are on the network itself to work together and accomplish whatever goal they have set without the need for any one person to have more power than another. I do not feel that this could have been accomplished better with any other architectural style. It is also worth analyzing the Object Oriented components of Git. While C itself is not necessarily Object Oriented, certain portions of SOLID still do apply. Git definitely does follow the Single Responsibility Principle. Each class/file in the Git Object Database has one purpose and one purpose only (see Figure 1). The Commit class “points to a tree representing the top-level directory for that commit as well as parent commits and standard attributes.” Likewise, the Blob class represents a single file stored in the repository. The Tag class just points to a commit. Lastly, the Tree class essentially holds all data. The Open/Closed principle seems to be in effect as well. Each and every class can be extended but not modified. With each commit, the hierarchy builds upon itself but the original files are also kept separately (see Figure 3). Next, I do not believe that the Liskov Substitution Property is necessarily followed. This is simply because one needs to take the behavior of a given hierarchy into account. While all Objects present are all GitObjects, it is not necessarily the case that every single one of those Objects can be substituted for every instance of a GitObject. That being said, Liskov herself says that it is impossible to determine whether or not the principle applies unless the model is viewed without isolation. Just because those Objects are GitObjects and theoretically can be substituted for GitObjects in methods does not mean that there is not a scenario where this will break down. It is for this reason that one cannot say with certainty that this architecture follows the Liskov Substitution Property. I also do not believe that Git follows the Interface Segregation Principle. The reason I say this is simply because, judging by the hierarchy shown in the article, the Git Object hierarchy does not actually make use of an interface or abstract class. Therefore, it cannot follow the Interface Segregation Principle. Lastly, the Dependency Inversion Principle needs to be taken into account. This principle fails as a result of the same issue as the previous principle (because the hierarchy does not use any interfaces or abstract classes).

In terms of complaints and a potential problem that could be fixed, there exists one significant one. According to the article, several users have complained that Git lacks the IDE integration that other VCS's (such as Subversion) possess. The blame itself goes towards the design pattern of the toolkit that Git uses. Because Git was created originally for the Linux operating system, it utilizes a structure that was built to be primarily from the command-line (like Unix systems). To resolve this issue, I posit the use of the Mediator Pattern (see Figure 4). The purpose of the Mediator Pattern is to

“introduce an object whose job it is to coordinate between other objects.” In this case, the two objects are the IDE and Git. The issue that was found was simply that Git was not present in enough IDE's. To resolve this, my proposed solution would be to create a plug-in for different IDE's such that this plug-in acted as a mediator. Just as Subversion created Subclipse for developers who use Eclipse, my proposed solution is to create a plug-in to allow IDE's quick and easy access to Git. The mediator itself will in turn send the information to the repository (by using Git) and will then pull back the changes from the repository and synchronize the workspace of the IDE with the repository. This mediator will essentially solve the problem that is present as it will be compatible with both the IDE and Git itself.

For the purpose of evaluating different aspects of Git, one must utilize different criteria for assessing the an application's quality. In this case, the criteria that will be used are the FURPS and the “ilities.” Functionality is the first aspect that requires analysis. With this, Git does a relatively good job. Git provides all documentation for commits as well and provides documentation for branch merges as well. The source code also allows one to roll back changes. Meaning, if one has made a mistake it can be rolled back and undone/redone. As a result, it provides for functionality. The fact that all of these things are logged keeps Git traceable and definitely allows it to meet the functionality aspect of FURPS. Now, for the next criteria: usability. Git allows for good usability. Torvalds' version of Git was created solely for the purpose of running from the command-line which may be difficult for some users. This is both good and bad. While some may consider this to be low usability, I'd say it can also provide good usability because a user can make his/her own GUI. That being said, GitHub actually provides users on Windows and Mac with a very clean GUI which allows the user to use Git without difficulty. In terms of reliability, it is safe to say that Git is very reliable. This is a result of the vision that Torvalds had for Git to begin with. Because it is open source, several people can find issues. This allows several different people to check for mistakes which promotes reliability. In addition, it handles scaling (can take a lot of data) and still be fast. The way that Git compresses and stores packed code is remarkable and truly allows for a very reliable experience (Torvalds' vision). Git will not allow anything that can destroy/corrupt data without first forcing the user to take the proper precautions. This as a result makes Git very reliable and a true pleasure to use. The fact that one can recover lost changes is also phenomenal. Git's performance is also stellar. This is a result of being written in C. C is fast and because it's compiled separately for each OS, it is extremely fast. This again was a part of the vision that the creator of Git had for his brainchild. His goal was to provide a service that had very high performance and Git definitely provides this. It is extremely fast in terms of runtime and is a premier VCS. I'd say that it provides the “S” in FURPS as well. Git is definitely sustainable. The fact that it is a peer-to-peer network helps as it will continue to exist so long as there are users willing to use it. The fact that it's open source which makes people keep it going provides sustainability. In addition, the RSA key that needs to be accepted from the repository makes it secure. Stability comes from the safeguards from corruption (SHA's) put in place by Torvalds.

In terms of the “ilities”, Git does a good job of applying them. Git is definitely maintainable. It does not require any excessive work to use and is very simple to keep up with. Git does fail however in terms of portability. In the article itself, it is stated that originally a few commands were “implemented as shell scripts.” These shell scripts actually resulted in Git being less portable, especially to operating systems such as Windows. It is also worth noting that this is a reason that Git is not as popular as it could be (because several companies chose not to use Git because of its early-day portability issues).

All in all, Git is a revolutionary project. It utilizes a peer-to-peer network to accomplish its goal of being the best VCS that currently exists. Interestingly enough, Git is only Object Oriented in some aspects. Upon further analysis, it has proven to be a very high quality VCS (according to FURPS and the “ilities”) that will be around for generations to come. In general, if there is any VCS that should be used, it is Git. It allows one all the freedom to do what one wants but is also restrictive enough that it prevents corruption. To be honest, it is a phenomenal creation and I hope that it grows in popularity so that each and every developer will experience the wonder that is Git.

Figure 1:

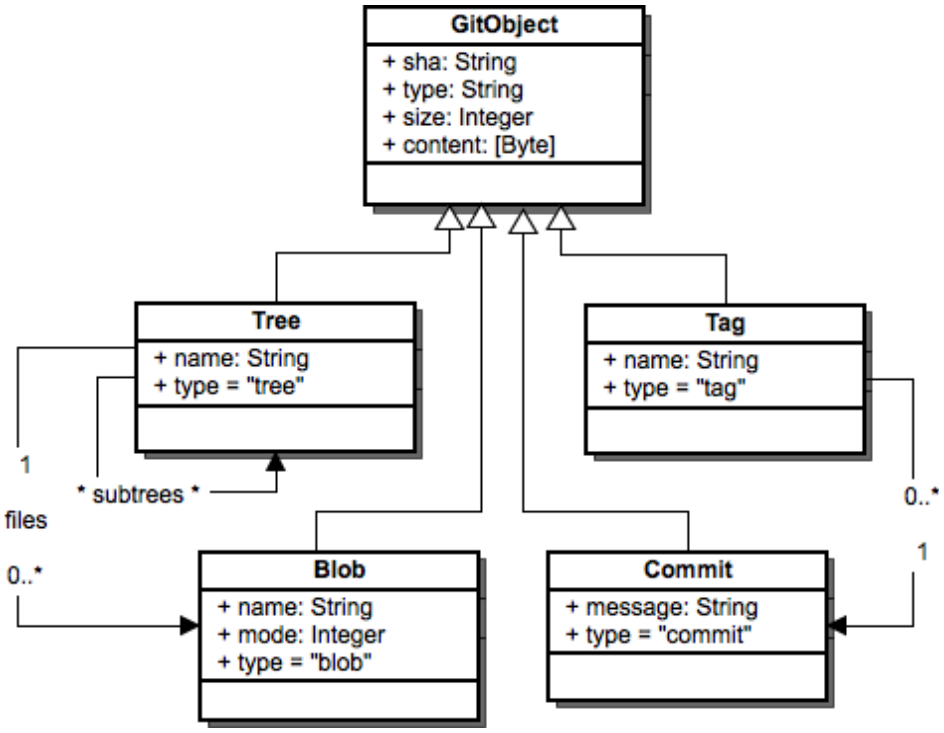

Figure 2:

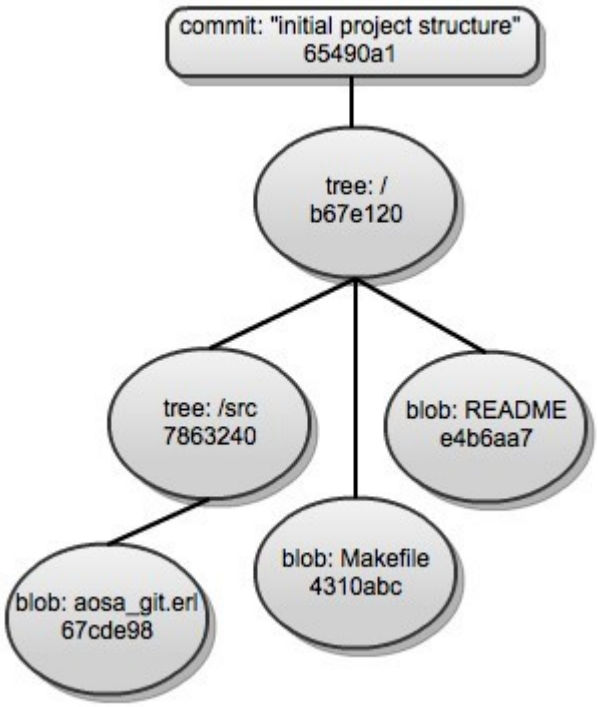

Figure 3:

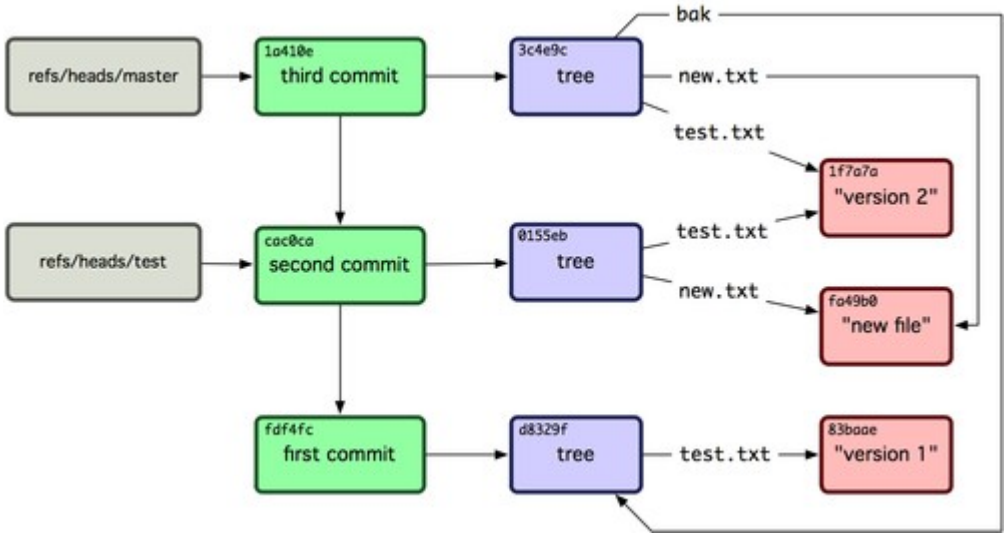

Figure 4: