97

10 18 10 10 | 15 24 10

Group 29: Justin Chu, Hai Hong Tang, Mark Ang, Punit Shah

VisioLine: A visualization tool for understanding per line static attributes of a github repository

Aspects to Represent

Primary Goal:

With VisioLine we are attempting to represent per file and per line contributions based on commits, of a git repository, over time. This will be done by examining the "who edited last" information of each line of the given code for a single node on the commit tree. We can then apply it to more than one node in the commit tree, to show changes over time.

the later called "annotations"

Secondary Goal:

Based on the visual framework we have built, we will try to add other static analyses and relate them based on features of the code such as (but not limited to):

- Categorization of sections of code (eg. what parts are comments, import statements, etc.)
- Raw information (eg. how many times a line has been changed, how much white space each line has, how long is each line is in characters, etc.)

Rationale

We have chosen this representation because knowing about per line changes over time can be extremely useful for the developers in a code base. However, the stakeholders are not just developers: project managers, people curious about a specific code base, and even computer science educators could benefit from using VisioLine. Our visualization will help users understand how a code base has been physically edited and pinpoint line specific problems in the context of chronological changes made in the code base. More specifically, it will allow developers to see what lines of code they most often change, maybe leading to improved code design. For example if a very large section of code is changed with every commit, it could indicate a serious flaw in design. Developers would be able to easily see this happening and address it.

Project managers can examine the behavior of their developers and how they interact with each other in the context of the code per line. They could easily see if one developer constantly overwrites another developer's changes or if one developer presents a potential conflict in the team. People curious about a certain code base could apply this program to that code base to better understand the software development practices used in creating it. They can use the program to explore a code base to see how it has grown and changed over its lifetime. Educators can use the program to teach students the effects of good and bad software design as well as programing practices. For example, they could use the tool to demonstrate the effects of high coupling in code bases. This would manifest itself visually as changes in many files in a single commit even when the change is primarily a change to one file or class.

These are all use cases based on changes over time, but you've elected to show only one node at a time to start out with.

Input

Inputs: source code, text of the code and git check-in data.

Primary data: Source code with line history (who edited last) via git blame command. Thus, the code must be a git repository. The code will be applied to a single node on the commit graph

(representing a single commit).
Caveats: If we implement other static based analyses, we may need to limit the type of code we use. For example if we plan to categorize the parts of code like comments or import statements, we might need to restrict our program input to a specific language (e.g. java) to make sure we parse things correctly.

Output

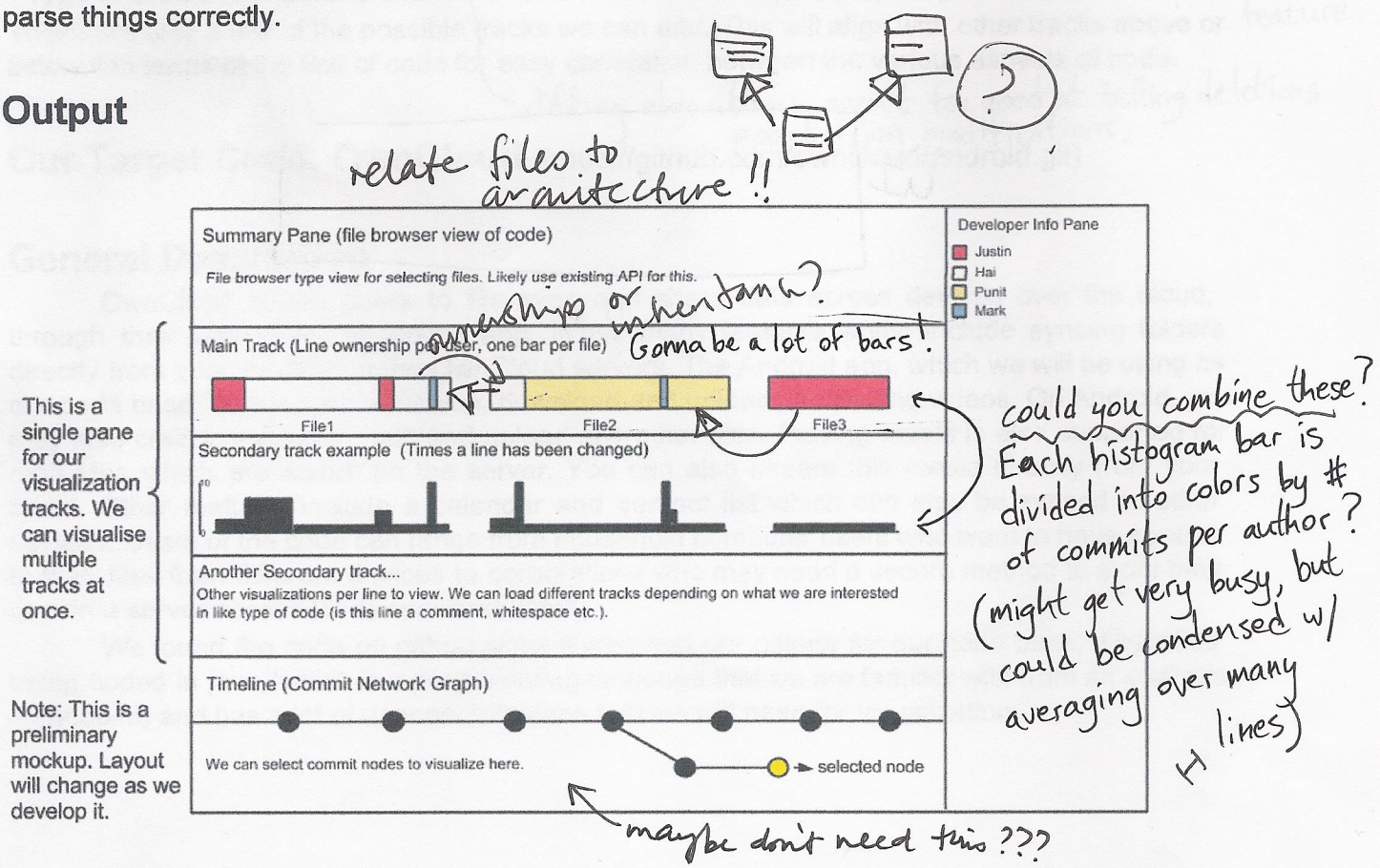

Figure 1. Mock-up of VisioLine. Consists of 4 panels, file summary, timeline, developer and the visualization pane. The visualization pain can visualize multiple tracks that the user can select at once.

Core Features:

Our application is a representation of a file or set of files in the code base at a given point in time in the commit history. Each file will be represented as a bar with each line in the file as a column within the bar. The color of each column will represent the developer who last modified that particular line of code. We define this as our main track (fig. 1) and it will be our base for all other parts of the visualization. The user will also be able to zoom into each file (bar) so that he/she can view more detailed information about that section or line of code such as the name of developer who last modified that line, and the actual source code itself. The commit history of the code base will be presented as a graph with each node representing a commit that the user can select to easily see files changing over time.

Secondary features:

We may also show other representations (as secondary tracks) of each file such as the number of times a certain section has been changed in the commit history up to that point. Potential secondary tracks:

- Times a line has been changed as a histogram (as seen in fig. 1)
- Length of each line as a histogram
- Where lines have been deleted
- Type of code a line is (ie. is this a line code or a comment or whitespace)

These are only a few of the possible tracks we can add. This will align with other tracks above or below it in terms of the line of code for easy correlation between the various aspects of code.

Without this, you won't be able to conclude much from the annotation bar graph. You should make it a primary feature.

? not interesting

diffing algorithms aren't too good at telling deletions apart from modifications.

Our Target Code: OwnCloud (https://github.com/owncloud/android.git)

General Description

OwnCloud allows users to file sync and share data across devices over the cloud, through their web client or mobile app. It has many features which include syncing folders directly from your desktop to the OwnCloud servers. The Android app, which we will be using as our code base, allows you to browse, download and upload photos and videos. On Android, you may also create, download, edit and upload any other files. Playing music is also supported for mp3 files which are saved on the server. You can also stream this media directly from your cloud. Other features include a calendar and contact list which can also be synced to other devices. Users of the code can range from household computer users who want to have access to their files from different devices to corporations who may need a secure method to store their data in a server accessible to specified users.

We found the code on github since it matched our criteria for our code base; it satisfied being coded in java (which is a programming language that we are familiar with from an analysis standpoint) and has a lot of user contributions that we will need for visualization.

Architecture

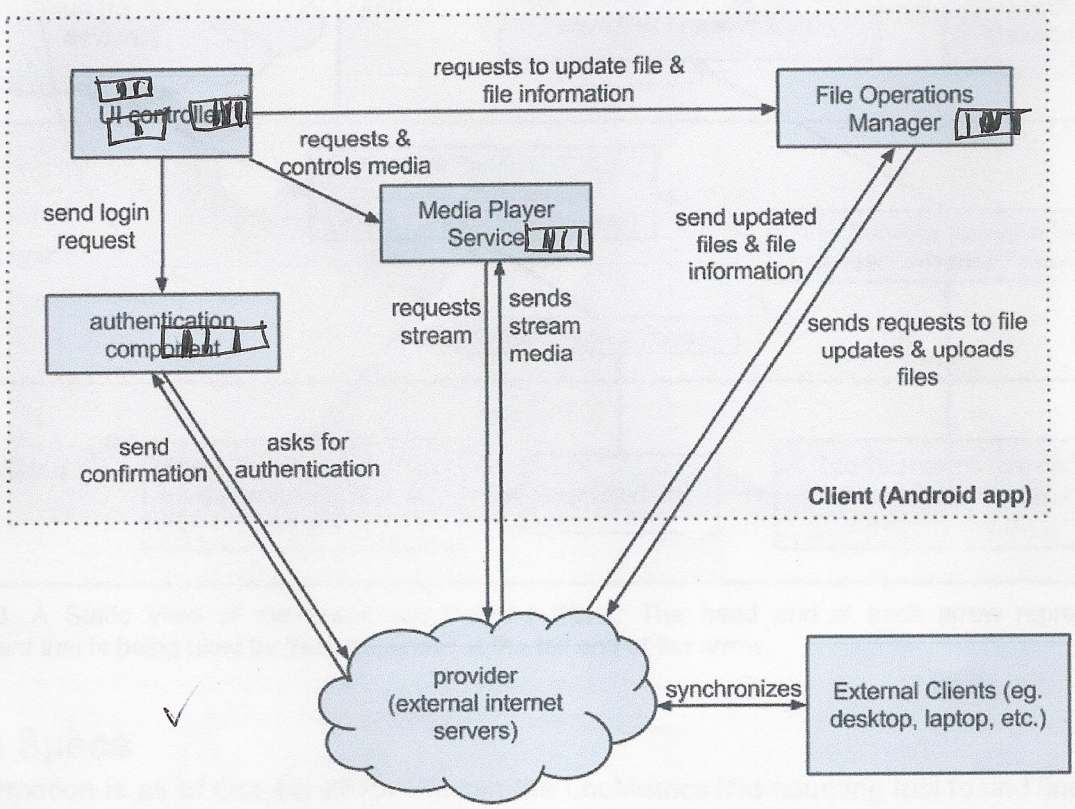

Figure 2. A Dynamic View of the OwnCloud Android Client. Client Code is represented by a dotted box used to distinguish external and and internal application processes.

architectural Mappings

UI Component { ui class 1 ; ..... }

Authent Component { ; }

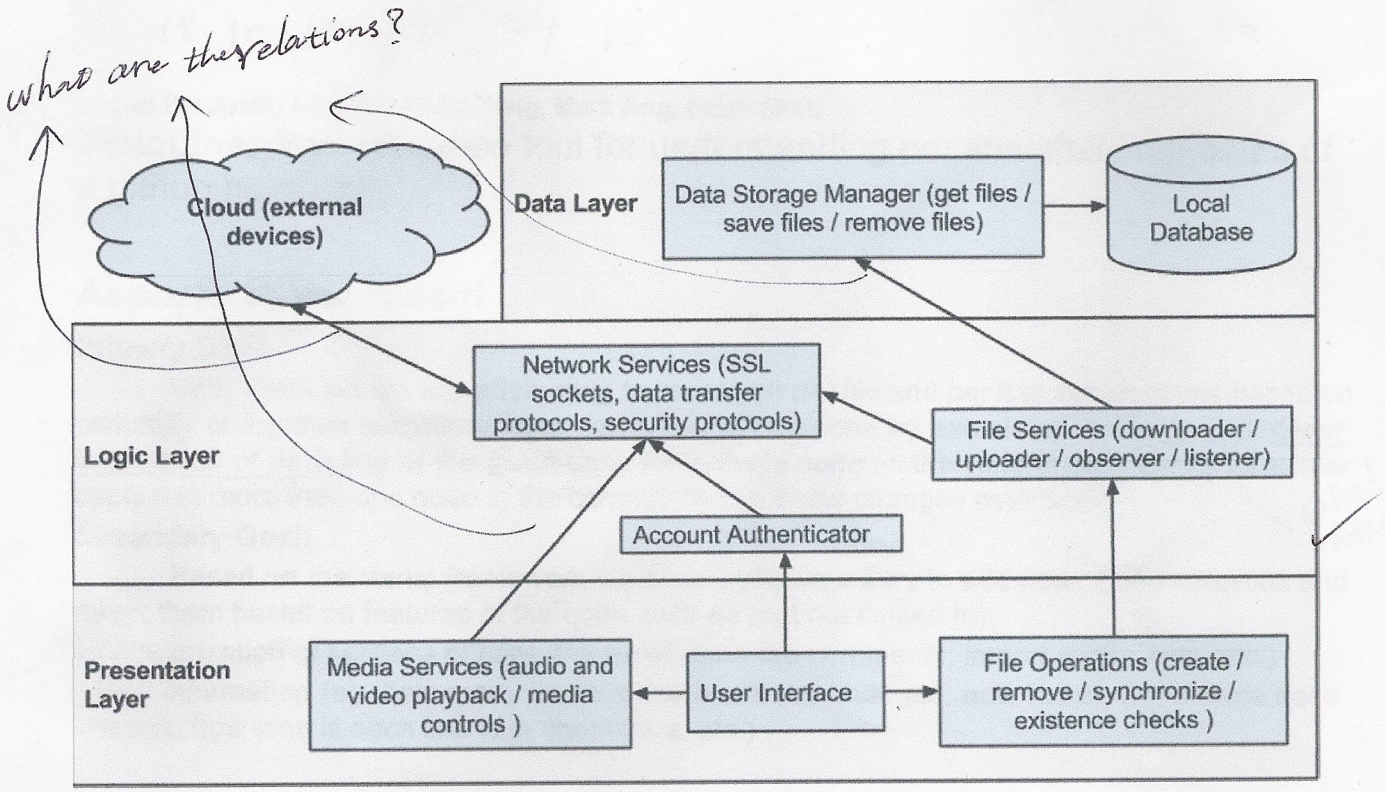

Figure 3. A Static View of the OwnCloud Android Client. The head end of each arrow represents a component that is being used by the component at the tail end of the arrow.

Code Specs

All information is as of Oct 4th 2013. We ran the LocMetrics line counting tool to find line counts and source file count numbers. Github's web interface was also use to acquire much of the other information (commits, contributors, language used, etc.). The rest was manual analysis by looking into the source code

Scale of Code

Source Directory: \android-master\android-master\src
Directories: 28
Source Files: 117
Lines of Code: 27266
Number of Commits: 1010
Number of Released Versions: 15
Branches: 20
Contributors: 14

Code information:

Language: 89.1% Java, 10.9% python, 0.1% shell
APIs used: Android Support Library v4, Transifex, Apache JackRabbit
Source Control: Github

System information:

Client Systems: Machines that run Android OS
Server Systems: Server code (not included) can be run on windows and linux architectures.
Connection Medium: WebDAV protocol