Software Maintenance and Evolution

Hatim, Aida

December 2017

1 Introduction

The following report summarizes the results we have achieved by working on the assignment for the course *"Software Maintenance and Evolution"*. We were requested to write a report on the evolution analysis of a project repository. The assignment was summarized as follows:

"Given a software repository, perform several analyses in order to assess the maintainability, modularity, complexity, and quality of the software in the repository, as well as the development process.

The repository we chose to analyze for this project is "Gimp". We considered other repositories as possible options but in the end came to the agreement that *Gimp* would be the best choice. Gimp is a large open source software that is used for retouching and image editing, drawing and converting between image formats. We reasoned that the files are large enough to perform a fair analysis, there are multiple authors working on the revisions of the software, the source code is mainly written in C/C++.

2 Basic Repository Investigation

In order to answer the questions regarding this section and to get a general overview of the repository we executed some commands through the terminal. The commands and the output will be included for each respective question.

2.1 Revision history

In this report the project is reviewed from its initial commit up to the latest one.

2.1.1 Revision count

The gimp repository is still under development. Up to now **39807** total commits were done in the "Gimp" repository.

```
1 git rev-list HEAD --count
```

2.1.2 First commit

The first commit was done by **Sven Neumann** on **01/01/1997**. The information was retrieved by the command:

```
1 git log --reverse
```

2.1.3 Last commit

The last commit was done by a git user named **Jordy Mas** on **16.01.2018**. The information was retrieved with the command:

```
git log
```

2.2 Largest folders

By executing the command:

```
du -sh */ | sort -h
```

we get a list of all the folders and their sizes. These folders are sorted from the smallest to the largest. Below we are only presenting the 3 biggest folders:

```
68K     libgimpmodule/ \\
72K     m4macros/\\
80K     etc/\\
```

However this folders contain a lot of different information, installers, etc. In order to analyze the source we need to analyze the files with .C and .h extensions. Therefore, we perform the following command to get the information regarding the size of this files:

```
find . -name "*.c" | wc -l    (for .c files)

find . -name "*.h" | wc -l     (for .h header files)

The result we get is \textbf{1643} .c files and \textbf{1470} .h files.
```

2.3 Active Developers

Below there is a list of the 4 most active developers who have contributed since the initial commit which dates back to 1997. We notice that **Michael Natterer** is the most active developer from the start up to date.

```
 12215  Michael Natterer \\
  9281  Sven Neumann \\
  1391  Martin Nordholts \\
  1274  Manish Singh \\
```

The output is retrieved with the following command:
git shortlog -s -n

3 Getting a first Visual overview

3.1 Stable Release

Figure 1 shows us that the release has been increasing linearly since the first commit, with an exception in some time periods. The yellow dots represent commits on the gimp files. There is a stable release during the years 2000 and 2009. During this period of time, there are few yellow dots and the quantity of files did not significantly change. (No significant change in the source code). This could indicate that maybe many GIMP users (or all of them) used the same source code to compile binaries for their different platforms. Just before the third arrow which shows the period of 2012, there is a small deviation or change in gradient. This shows that there is no stability at the start of 2012. A possible reason that might have caused this, could be that a GIMP developer might have decided to alter the source code to their specification. The stability comes in the middle of 2012 because there are very few dots, implying that there is no change of source code applied. All users used the initial GIMP source code to edit their images.

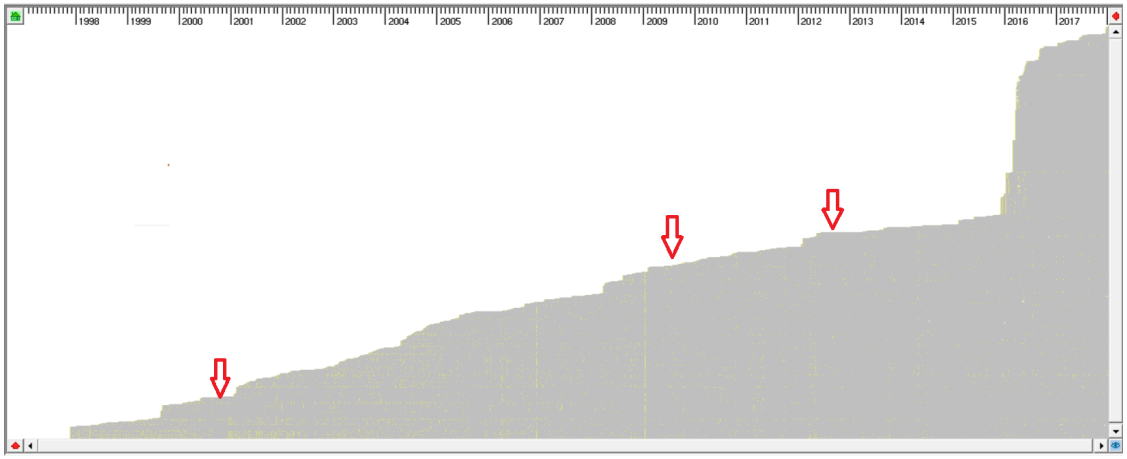

Figure 1: Stable Release

3.2 Unstable development periods

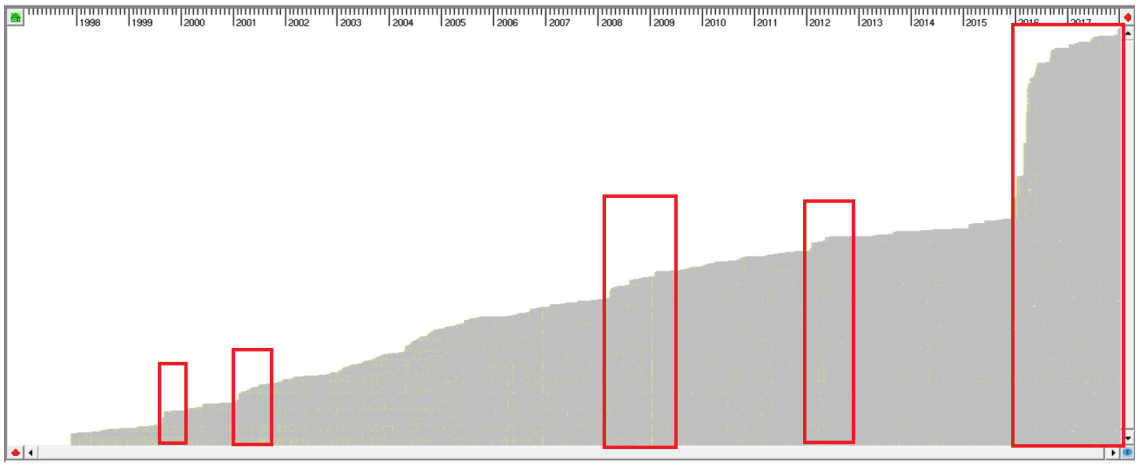

Figure 2: Unstable development periods

In the above figure, it shows that just before in 2000 there is some little changes committed. Some yellow dots changed with a little deviation and slope is somehow rising. This shows that during this period the source code used here was somehow different among the GIMP users hence some level of instability. But centrally to before 2000, after 2000, the slope is rather flat which hypothetically show that there is bit of stability here, hence there is no change in the usage of the GIMP source code. In the beginning of 2001, 2008 and beginning of 2012, the case here is that there a lot of dots which indicates commits. In all these years the slope is somehow steep showing that in these periods there was some lack of stability in the source code. This shows that, during 2001, 2008 and 2012 GIMP user decided to use different source code to develop and accomplish their task. In the 2016, there is big change in the steepness of the slope. This may be caused by the fact there was a big change in the source code and there was heavy development of the GIMP program where there was a lot of different source codes used here which caused unstable development period. This conclusively indicates that over these periods of time there were files or functionalities added to the GIMP repository but the big change was made in 2016. Contributions to master, excluding merge commits

The above graph shows amount of commits over different periods of time, excluding merge

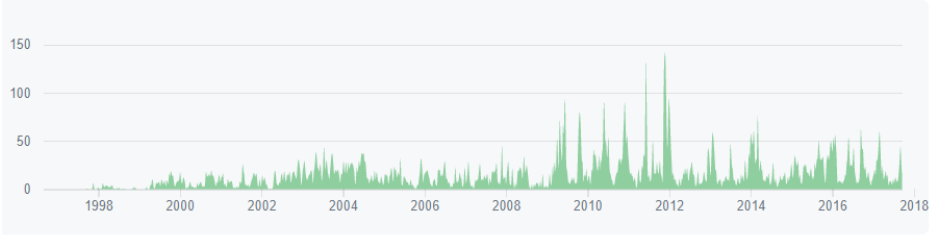

Figure 3: Amount of commits over different periods of time

commits. The big change comes between before 2010 and 2012 which shows that in this period the code was absolutely unstable. In other periods, there were little changes showing hypothetically that there were some little addition or changes of the source code.

3.3 Current State

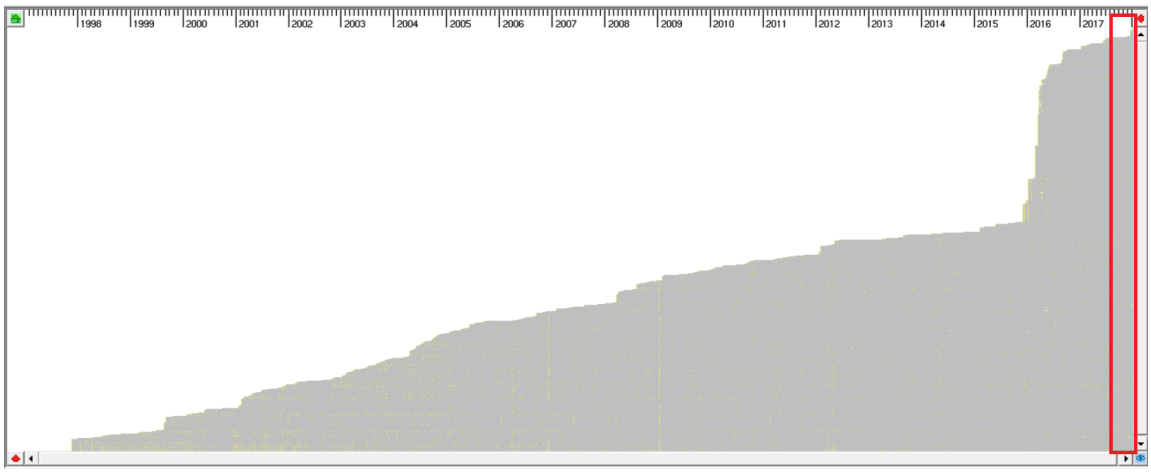

Figure 4:

In the beginning of 2017 GIMP repository investigation shows that there is some changes up to the middle of 2017. This is due to the fact that the slope is rising with a small gradient, leading to the hypothesis that in this period of time the source code has not been stable state currently. As illustrated and show in the figure below, this may be due to some addition of files to the GIMP repository and also considering that changes in original GIMP files may be another due cause. In the middle of 2017, the slope remained steady for a short time leading to the hypothesis that the source code in this period of time was stable. There was no changes committed because there was no addition of files or changes made to the files. In the end of 2017 the slope rose steeply leading to assumption that the stability of the source code was good at all. This is because there may be a lot of commits, reasons being that there was addition or change in source code.

4 Authors Analysis

This step would entail various forms of examinations dealing with developers or authors of the GIMP repository in addition to the evolution view of Solid TA and the author, code size, and the metric of the type of file employed.

4.1 Main contributors

We solicited the ideas about the creation time on the basis of the metrics of the authors. In this case, such will show the changes that have been carried out on the files of every author in different colors. Based on this information we identified that there are two major developers that did major revisions on the projects to an estimated 50% or more on the project: Michael Natterer (maintainer) and Sven Neumann (maintainer) are both maintainers of the GIMP repository. The figure below shows the performed revisions done by the two maintainers.

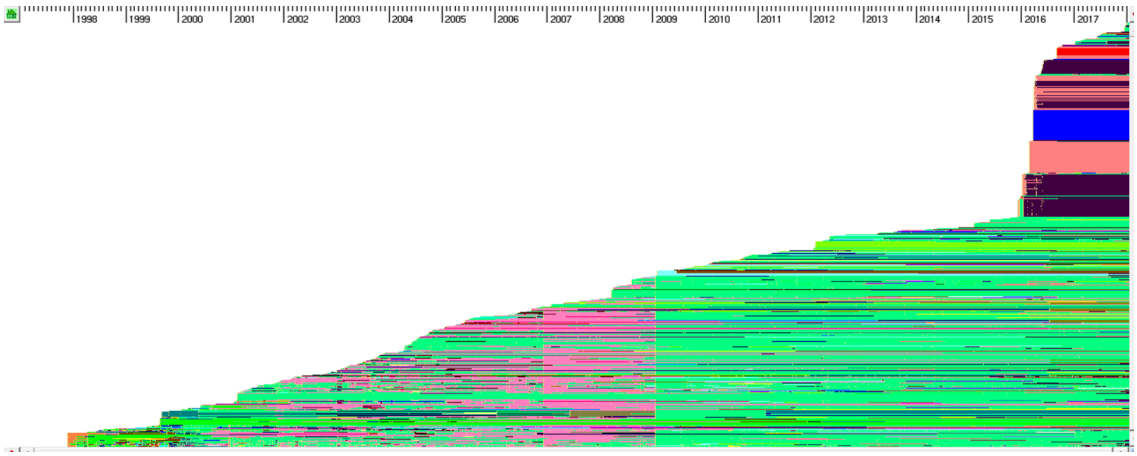

Figure 5: View of all files sorted on creation time with the inclusion of the authors' metric in Solid TA. See Figure 6 for the two major maintainers.

For the two major maintainers' repository, the blue bars in the background indicate the amount of revisions for each author.

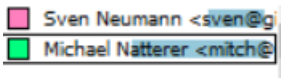

Figure 6: The Gimp maintainers

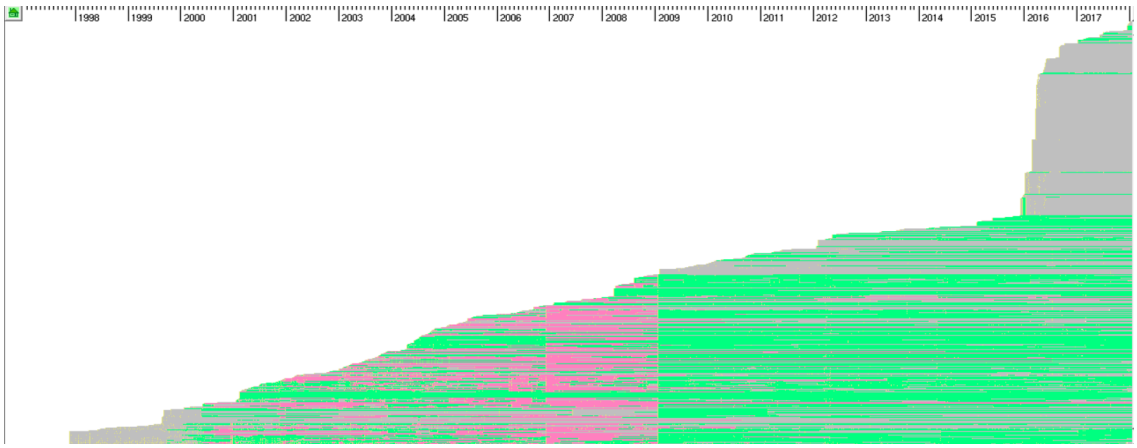

Figure 7: Viewing the contributions of the two major maintainers of GIMP Repository authors for the entire repository.

This figure indicate that Michael Natterer is the chief maintainer of GIMP repository, because

it is clear that the green color performed the largest number of revisions in the entire GIMP repository and we also checked GIMP repository which confirmed that current maintainers are: Michael Natterer.

In the following figures, the trend view of the file count per author is given for respectively all authors, the top two major maintainers of GIMP

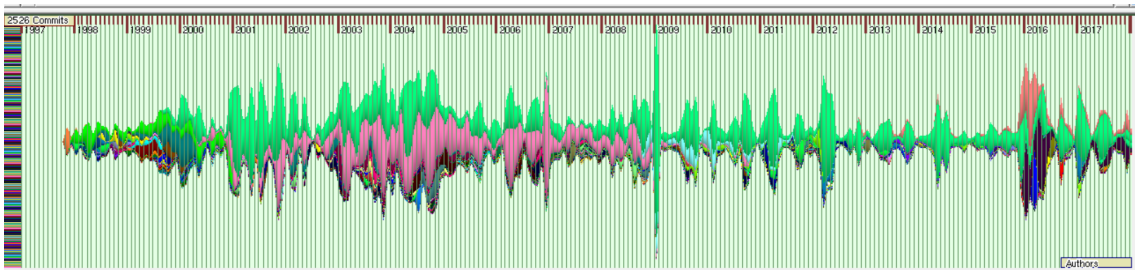

Figure 8: Evolution view of the file count of all authors over all files in SolidTA.

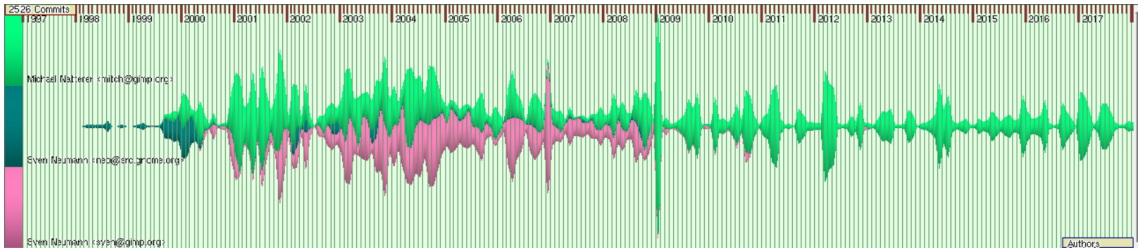

Figure 9: Evolution view of the file count of the top 2 maintainers over all files in SolidTA

We noted that in the list of developers, there seem to be some developer accounts who are actually the same person. This is because they have the same name but different emails (e.g. Sven Neumann listed under the same name but different email address neo@src.gonem.org and sven@gimp.org). In this case, there are eight (8) accounts under Michael Natterer and twelve (12) accounts under the name Sven Neumann, and we checked other accounts where the top two (2) accounts are Michael Natterer and Sven Neumann. As illustrated in the figure at the end of 1997, Elliot Lee it is represented as brown color; the first contributor to GIMP repository. Sven Neumann was one of the first contributors to GIMP repository in the beginning of 1998 represented in moss color and purple color, then Michael Natterer came later in the year 1999 and is represented in green color. From 1998 to 2009, Sven Neumann was the top contributor and after 2009, Michael Natterer took over and he became the top contributor up to date.

4.2 Replacing Chief Developer

Michael Natterer is the current top maintainer of GIMP repository because he consistently performs majority of the revisions on the project which are located in the virtually any location in the repository as shown by Figure. It would be difficult for the project to continue its release cycle if Michael Natterer was to quit the project as shown in the Figure since his presence in the project covers many old and new files. Furthermore, the amount of revisions would drastically decline, as presented. Nevertheless, if Michael Natterer were to leave the project, the files that this chief developer was most active in will have to be maintained by another developer, so a new developer or multiple developers will have to be chosen.

The candidate for taking over the work of Michael Natterer is Sven Neumann because he is the second contributor in the top maintainer list, and he started the project in it is early stages like Michael Natterer but his last work was about 5 years ago. In this case, it makes him as not active

Figure 10: List of the developers

Figure 11: List of authors

maintainer, and this also excludes Martin Nordholts for the same reason. Though Manish Singh is one of the top authors and he started to contribute almost 20 years ago, he did his last work more than 8 years ago. William Skaggs is also excluded because his last contribution was 10 years ago the same as David Odin because his last work was more than 9 years ago. Furthermore, Simon Budig is also excluded because his last work was 1 year ago. We also exclude Klaus Staedtler because his last work was more than one year ago and he is also a new member only two years thereby making all of them not active. Such factors make us to pick EII as a candidate because he is the second top maintainer in the last 12 months but he is a new member because he has contributed for only 2 years. That leaves us with Jehan because he contributed actively in past 12 months and he contributed to the project in more than 5 years thereby making him to have experience, and also an active author.

4.3 Correlation between author and file types

We will now analyze to find out if there is a correlation between the developers and the types of files that they worked on. To investigate this, we will enable the file type metric in Solid TA, and show the amount of files per type. the contents after this action was performed. Subsequently, the file view of all project files was first sorted on creation time, and then sorted on file type while this metric was enabled.

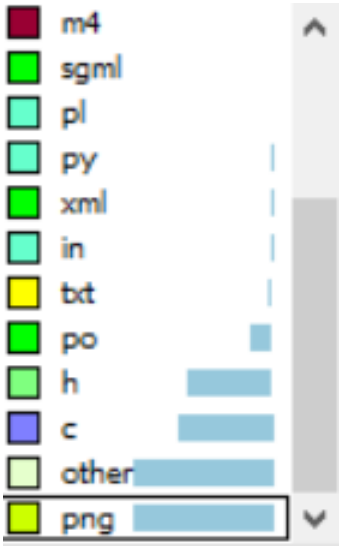

Figure 12: The type of files

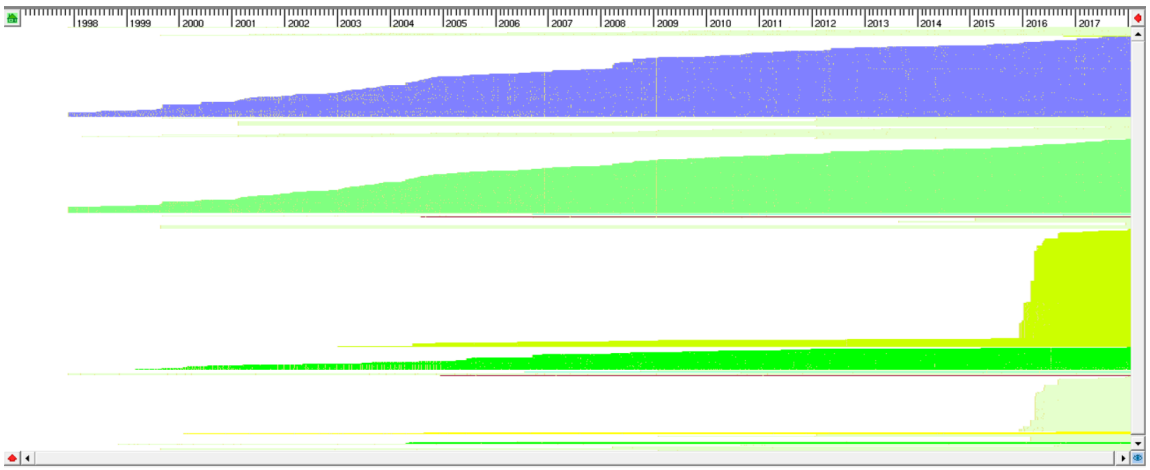

Figure 13: The view of the file types

4.4 Correlation between author and file location

Finally, we will analyze if there is a correlation between the developers and the locations of the files that they worked on. We enable the folder metric in Solid TA, and put the configuration slider on level 1, so that only top-level directories are displayed. The legend for this metric is depicted in the Figures. We enable the folder metric and we sort it in which we can see that the files are grouped together by the folder to which they belong . Similarly, to the previous analysis, we now turn off the folder metric and turn on the authors metric, so that we can see the location in which the developers performed their revisions. We conclude that Klaus Staedtler is the main developer for /icons directory, Michael Natterer and Sven Neumann were present in every folder except the /icons. Michael Natterer's work was not so much and Sven Neumann was not present in the /icons directory. Jehan was present in every directory and he has a big work load on /icons directory. In this case, it can be clearly seen that Michael Natterer and Sven Neumann are the main maintainers and Jehan could replace Michael Natterer.

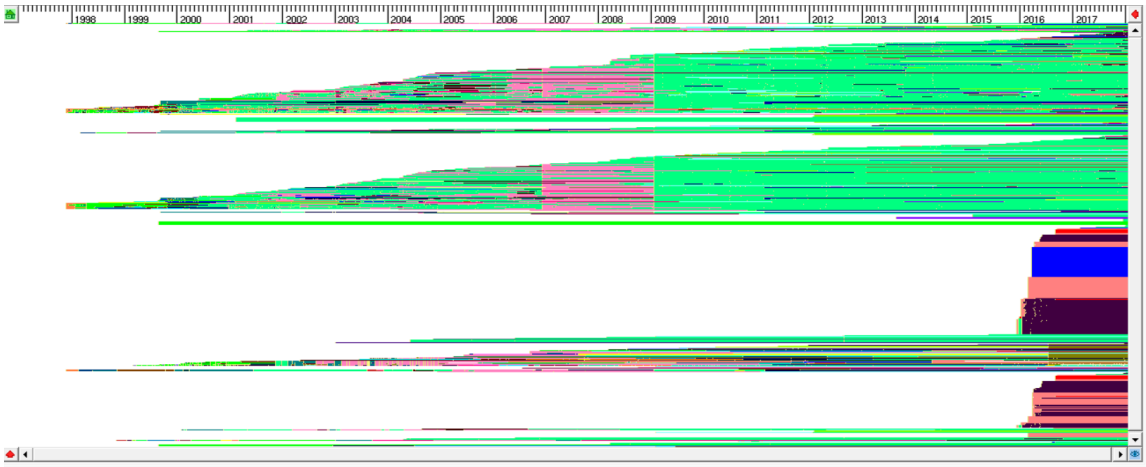

Figure 14: File view of all files

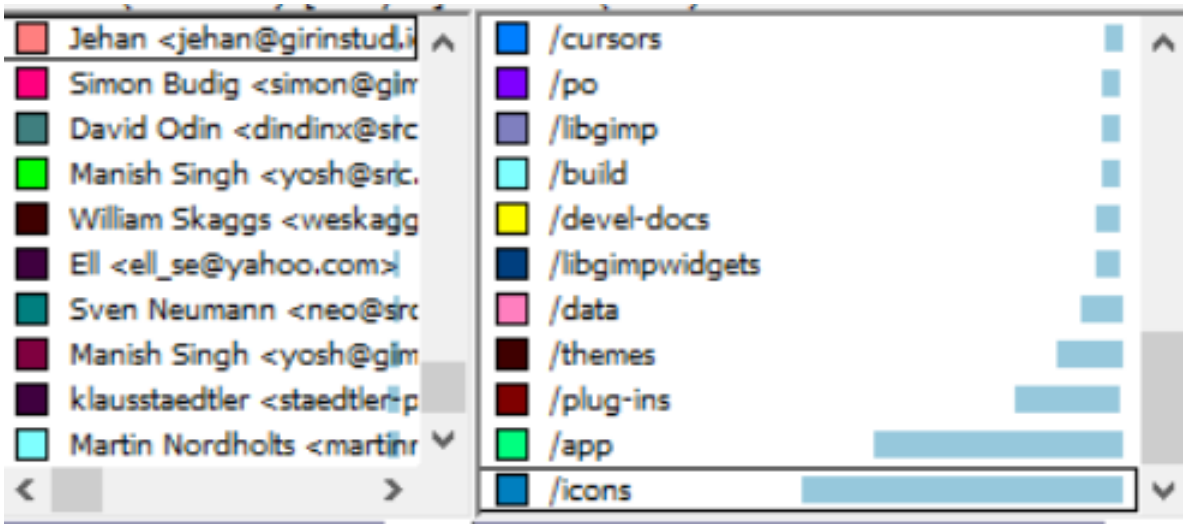

Figure 15: Authors

5 Code size analysis

In this section we will provide the results of several analysis done in order to determine the code size of the project. We will use the lines-of-code (LOC) code size metric and use the evolution view in combination with the code size metric. Since all the source code is located in the app folder, we will apply the metric to this folder. After retrieving the content of the files, the size of the code can be calculated.

5.1 Code size evolution

The Figure below illustrates the Number of the Lines of Code for this project. We can conclude that that the code size growth is generally linear, except for some ups and downs in a few parts of the graph.

The figure below represents the files grouped by size of their Lines of Code during time. Different files have different colors which represent their respective sizes of the files (blue represents fewer files while red represents files with larger amounts of code). We notice an interesting occur-

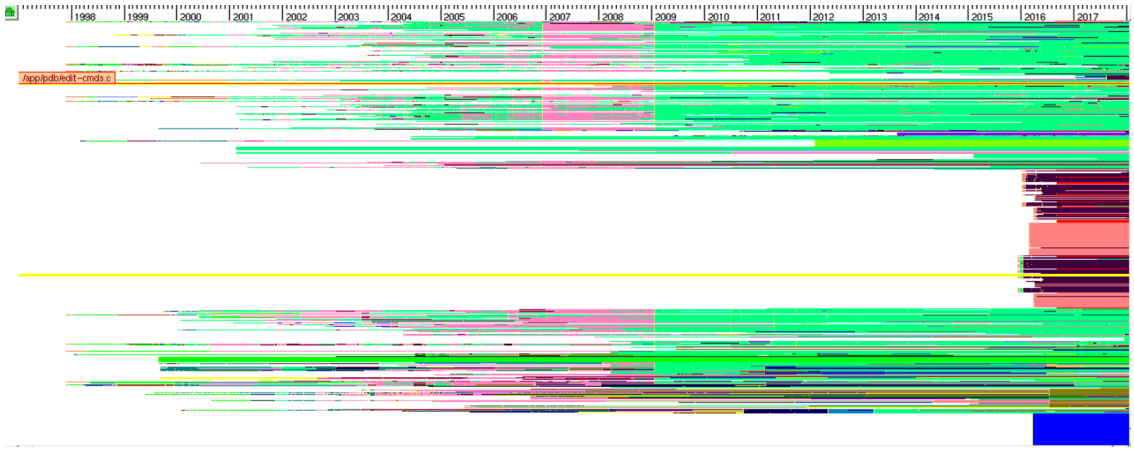

Figure 16:

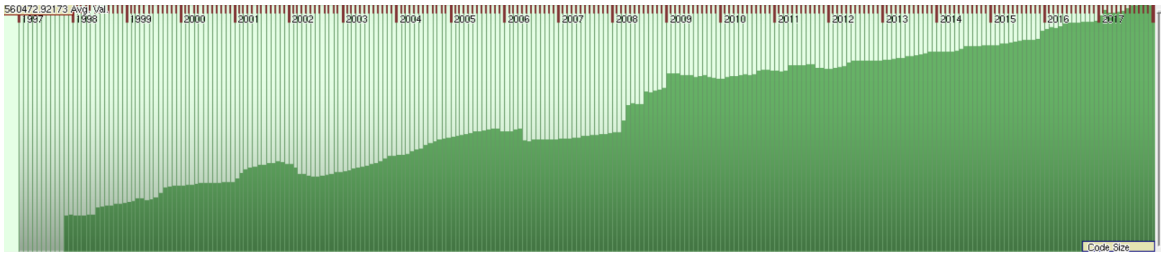

Figure 17: L.O.C metric over time

rence, the category of small files (mostly the blue category) grows rapidly over time compared to the category of the larger ones(red category). We can conclude that on average, the source code files are shrinking, but the distribution of the different file sizes is constant over time.

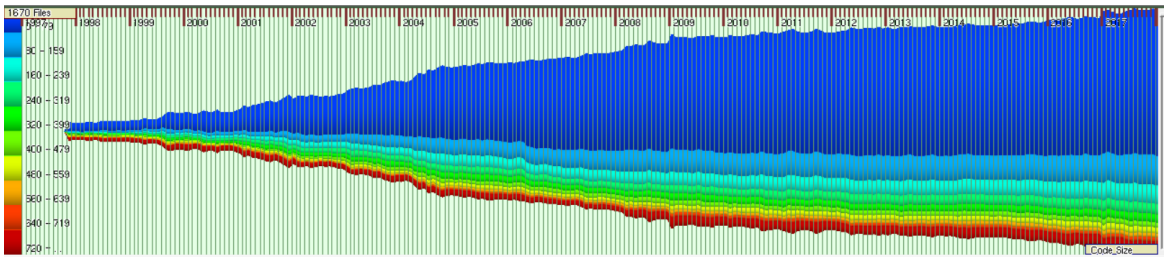

Figure 18: Files grouped according to their size

from this figure we think because the GIMP it always update repository , every new file at the beginning become at the blue color.

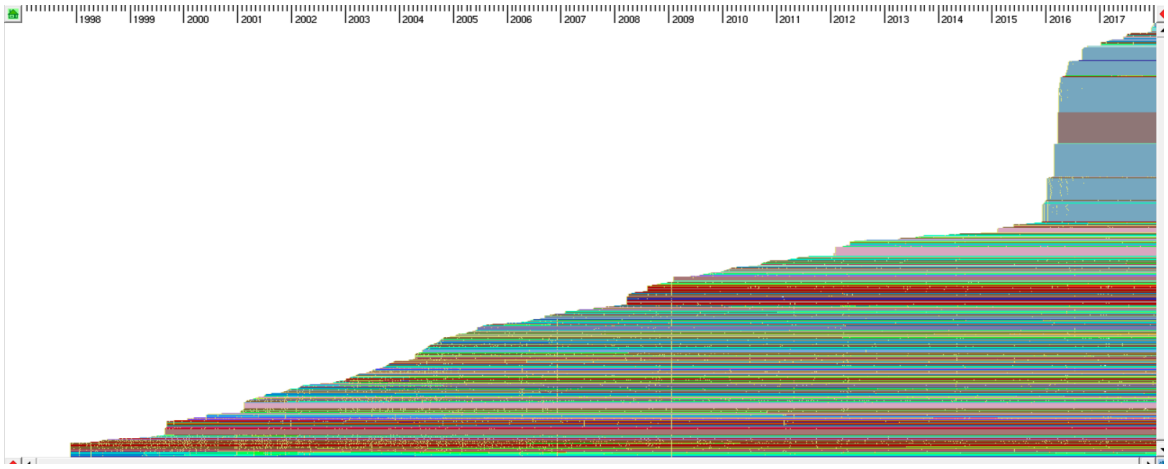

Figure 19: Files and the code size

Figure 20: Files and the code size

what we noticed from the previous picture is the code size of the most recent files are smaller than the older files in the repository.

5.2 Code size vs project size

We can estimate the report between the Code Size and the number of the code files by grouping the files by code size. This shows that more than 50% of the contents in the folders are source code.

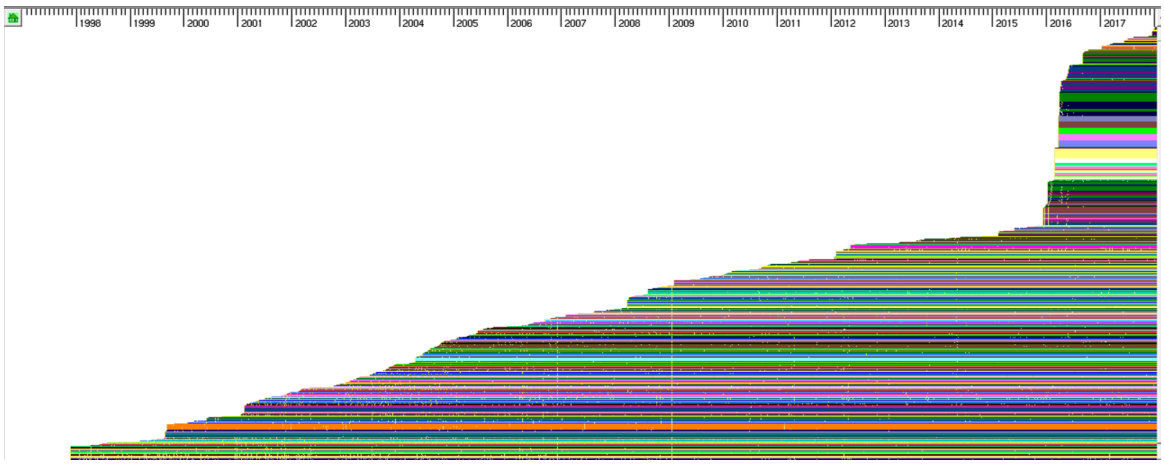

Figure 21: GIMP Folders

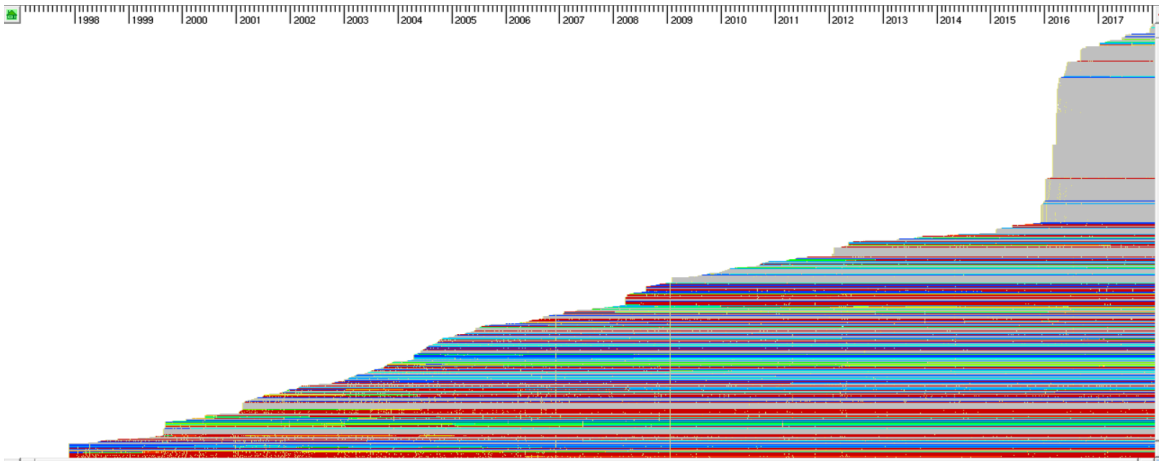

Figure 22: code size

6 Complexity analysis

In this section, several analyses will be performed related to the complexity of the source files in the entire project. During this step, we will describe the complexity of the revisions of files using the Complexity metric, which will be set to "Total Complexity". As for the last question the complexity has to be compared against the code size metric, which is not weighted, and all other complexity sub-metrics apart from Total Complexity are in fact weighted. Therefore, we decided to use the Total Complexity metric everywhere in this step when referring to complexity. The values that are used for the categories of this parameter can be adjusted with the configuration slider.

We opted to set the configuration slider in such a way that the highest category is in the range of "54 - ...". It led to the excellent distribution of the number of files between the different complexity

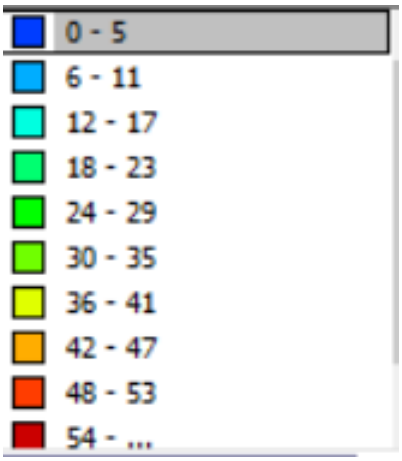

Figure 23: Colors - Complexity

categories, as can be seen in the figure because the values of the central groups are spread relatively evenly between the highest and lowest grade. We have also configured this view with more upper and lower, for the configuration. In general, it only had the effect of shifting the middle categories up and down in this view while increasing/decreasing the size of the highest/lowest class. This value was found to provide the most stable distribution.

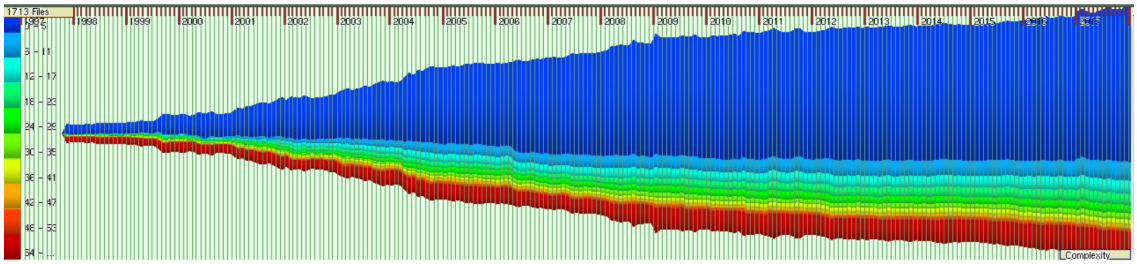

Figure 24: Colors - Complexity

6.1 Most complex files

First of all, we are interested in finding the files in the project that are currently the most complex. We define files as currently the most complex when their latest revision has a total complexity metric that falls into the highest complexity category. Since the complexity metric is only available for source code files, we only use these files in the initial view. In this initial light, which only contains source code files, we first select the highest complexity category, which looks depicted in the Figure.

In this view, we then group the files by the complexity metric, which will put the files with revisions that are highlighted in the top of the figure. These files are then selected and put in their own view, which contains all of the files that have ever had a revision in the highest complexity category.

We found out that each year the revisions consist the files that have highest complexity from 1998 up to date.

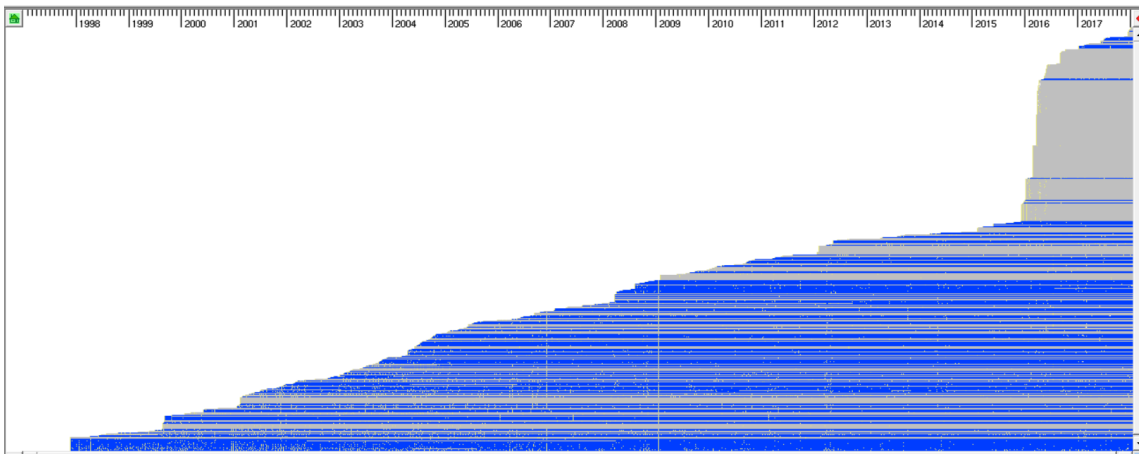

Figure 25: File view of all source files in the project, sorted on creation time, with the complexity metric enabled and the highest category selected.

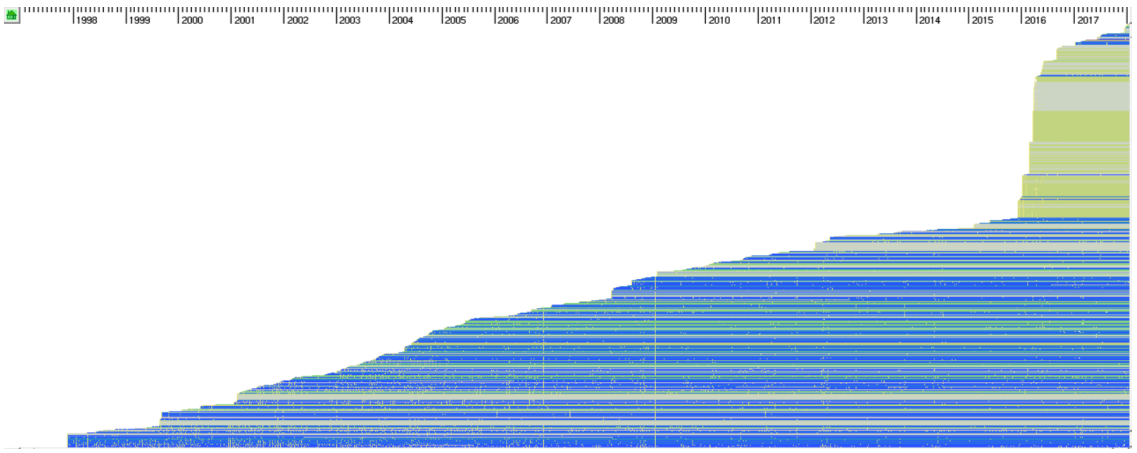

Figure 26: File view of the files that have a revision that is part of the highest complexity category, sorted on creation time

6.2 Complexity changes

A significant decrease in complexity is given, if a file changes from high complexity related colors to low complexity ones and remains like this until the end of the period. The following figure shows those files with a significant decrease in complexity.

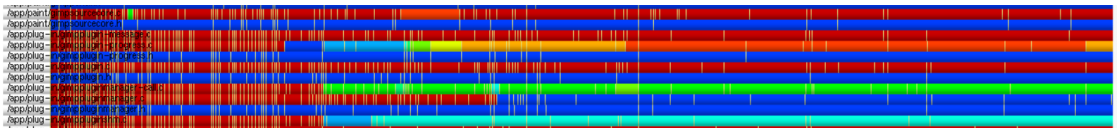

Figure 27: Complexity Decrease

A significant increase in complexity can be noticed, when a file changes from low complexity related colors to high complexity ones and remains like this until the end of the period. The following figure shows those files with a significant decrease in complexity

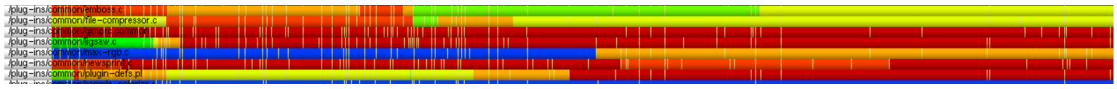

Figure 28: Complexity increase

6.3 Complexity and Activity Correlation

By sorting the complexity based on activity it can be clearly seen that the complex files have more activity compared with the least complex file see the following figure

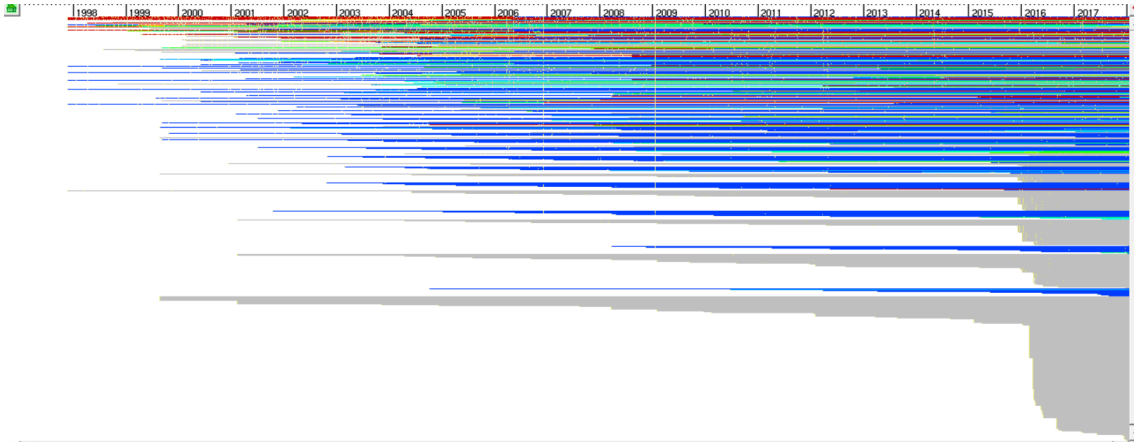

Figure 29: Activity of Files with Complexity Metric

6.4 Correlation between complexity and code size

To find a possible correlation between highly complex files and file size, both complexity and code size are used together and separately. The settings of these metrics and their relation with colors are illustrated.

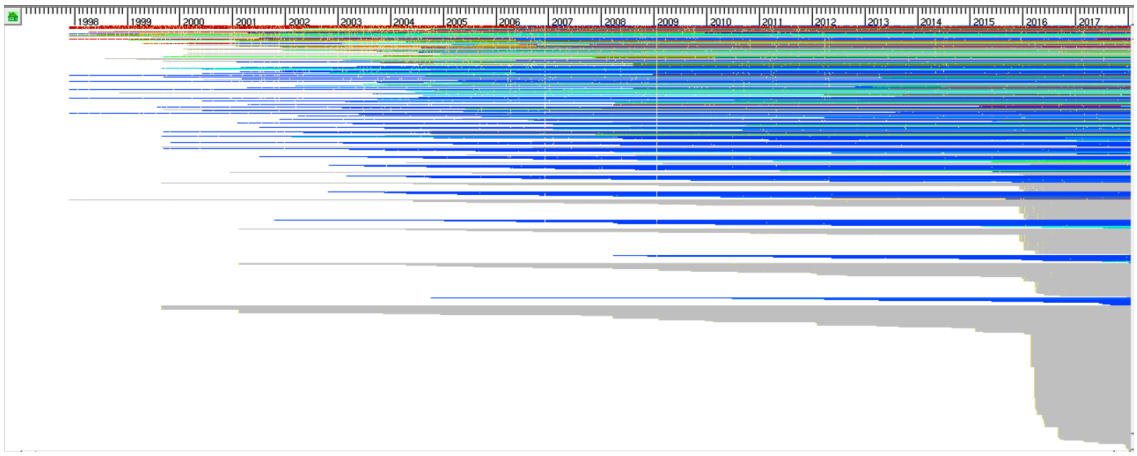

Figure 30: Complexity and code size

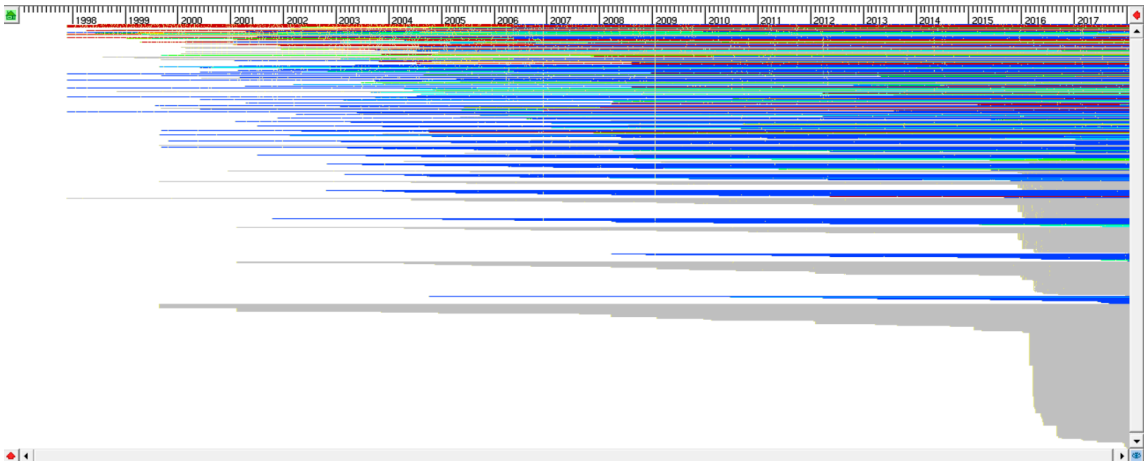

Figure 31: Complexity

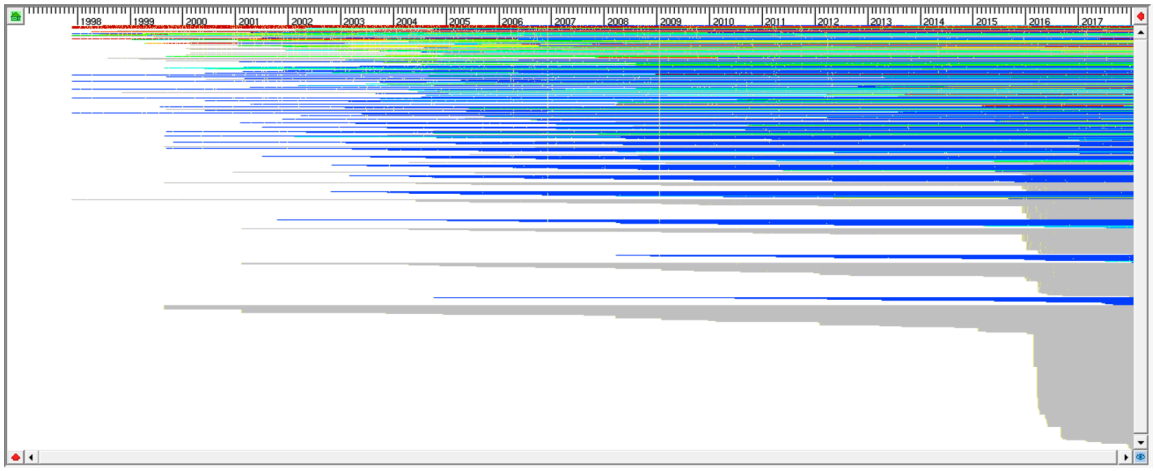

Figure 32: Code size

We conclude from analyzing the figures that the files which contain big code size include higher complexity as well which is realistic because the red color will remain red in the complexity figure and Code size as well as the other colors.

7 Conclusion

In this report we analyzed the github repository of "Gimp", a software product that is used for image editing. We did a thorough analysis of the code size, the folders, the authors(contributors), the average complexity of the code and the ups and the stable growth of the project in general. Overall we would say that this project is a huge repository that has been maintained by different volunteer developers who, in our opinion have done a great job. We would argue that this project is fairly complex and the complexity grows bigger in the larger files.

We used SolidTA tool to perform the analysis and we encountered a lot of issues while using it. The tool tends to crash in the middle of a certain analysis which sometimes would lead us to loosing the work of several hours. We believe anyway, that for experimental purposes it is still quite good.In order to implement the graphs we used some parts of the SolidTA tool but it was interesting how to analyze a repository and Get your hands dirty to learn the history of

the repository.For example,the autores,Code size and the complexity of the code.We noticed the correlation between many things like code size and complexity .

Generally we found this course quite interesting and practical.

8 Essay

8.1 Introduction

Nowadays version controls such as Git, SVN, etc. are being used to store software and ease the interaction between developers working on the same repository and to keep revisions of code. However, in terms of managerial purposes such as analyzing complexity of the code, the work division between developers, code size,etc. version controls do not offer much. In this essay we would like to show a dependency analysis by using call graphs, class interface graphs, containment graphs and build dependency graphs. In order to demonstrate the degree of complexity we need to visualize it. The essay will present a data model and a visualization strategy for displaying the dependency graphs.

8.2 Dependency data modeling

8.2.1 Dependency graphs

The structure of the data model is relative to the type of data that is stored and the type of output that has to be created. Below we briefly describe the types of graphs that are taken into account in the data model.

- **Call graphs** are graphs that display interactions between source code components, in order to get a better understanding of their interaction. They provide a clear indication of the control flow between different modules. In this case nodes are the function definitions, while the edges are the function calls.
- **Class inheritance graphs** are graphs that display the inheritance relationships between different classes. In this case the nodes are classes or objects, while the edges point out the different relations between them.
- **Containment graphs** are graphs that represent the hierarchical structure of the system in terms of containers and contained items. In the containment graph nodes would be: packages, folders, files, classes, etc. and the edges are the containment relationships between them.
- **Build dependency graphs** are graphs that are used to determine the dependencies that build the program when a certain part is changed. It can give an indication of the build time when parts of the code are changed. In this case nodes would be the files while the edges indicate the dependencies between those files related to compilation.

8.2.2 Data model

We are required to present a data model where all the dependency graphs can be stored. This model should store the nodes and edges of the graph itself, as well as contain the information that is communicated between all the nodes and edges. It should indicate how an element of version i corresponds to an element of version i+1.

"Repository" entity is one of the entities in the graph which represents the repository containing the source code. Versions represents the different versions of the software that are stored in git. Author represents the different developers that have contributed in the software repository. They are stored by the author name. Each version also keeps track of the authors that contributed in it. A Node can represent different software entities depending on the graph that is making the call. E.g if a call graph needs to be stored, the node-type of that node will be the method that

does the method call and the edge would be the method call. In case of containment graphs the node would be a package, folder,etc while the edge would store the relationships between these objects(folders, classes, etc). We can see from the Figure 5 that the node is connected to the edges which, as mentioned before, consists on a method call. The relationship between the two methods is stored in a edge.

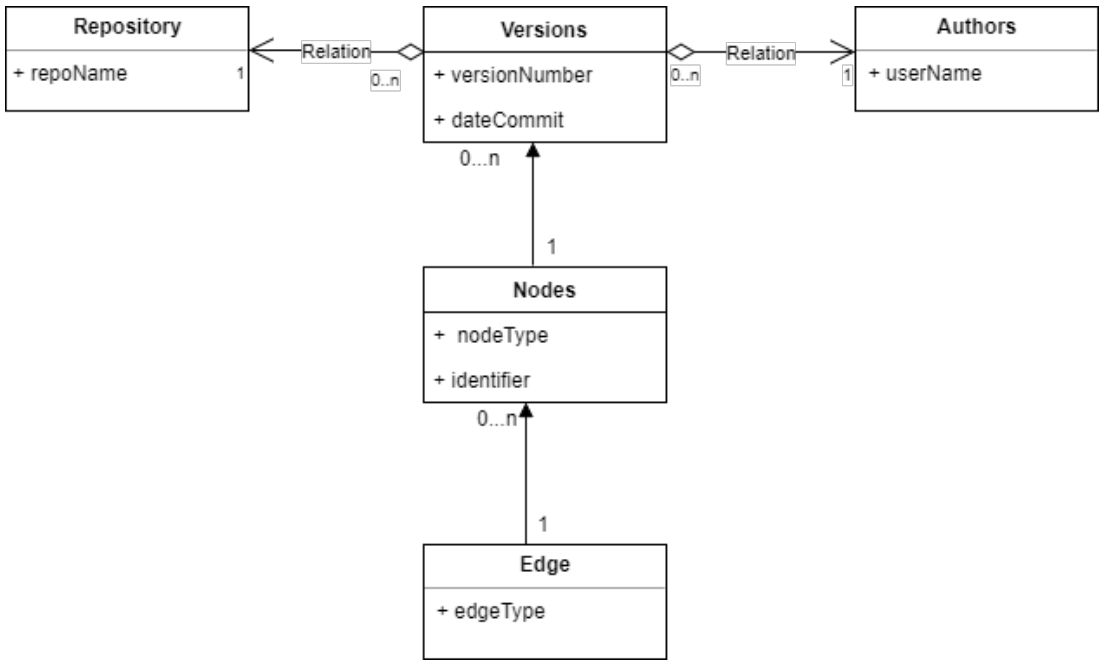

Figure 33: Data Model

8.2.3 Trade-offs

One downside of the graph dependency combination is that capturing all the information of all dependency graphs in a single graph makes it computationally more expensive to show the evolution of the software than by using only one graph. This increases the overall complexity of the software.

8.2.4 Implementation decisions

The larger the software repositories, the bigger and more complex do they become in terms of nodes and edges. The space for storing the graph should be as little as possible in order to have higher performance of the program.

Therefore, finding the most optimal way to store the graphs is one of the issues to be solved. We discussed several ways of achieving this desired output. One of them would be to store a new graph every time a new version of the code is done. This approach could be easily reached but presents a major drawback: low performance. Another way of solving this could be as below:

Whenever there is a new code version, a check is done to determine if a specific relationship between the nodes and edges already exists or if it is a new combination. If it already exists then the new version is added to the edge. If not, this new relationship between nodes and edges is added and merged with the existing graph. We compare the versions between the nodes and edges in order to determine if the relationship already exists.

8.2.5 Updating the model

As previously mentioned, graphs are updated each time there is a new version. The process that is followed every time a new version gets updated is:

1. A new version is pushed to the repository.
2. If the repository doesn't already exist than its "repoName" is stored in "Repository".
3. A parser will go through the code to determine the nodes and edges.
4. If there is already a node "n" in the graph, a new version will be added to N.
5. If there is no node "n" in the graph, then the node will be stored and merged with an existing version.
6. If an edge E already exists in the graph, a new version will be added to .
7. If an edge E doesn't exist in the graph, E will be stored combined with a version.
8. All nodes will be updated with the right file entity.

8.2.6 Dependency data visualization

Solid Trend Analyzer would not do a good work in displaying the call graph, it doesn't make a clear distinction between different files or their contents. It could happen that they get mixed, or that the functions are not executed properly. The figure below shows a possible visualization of the data model presented in the previous subsection.

On the left side of the picture we have nodes and on the right we have the edges. When a new commit is added, it appears in the left side of the picture. When there is a new version then this connection is transmited to the right side, the edges.

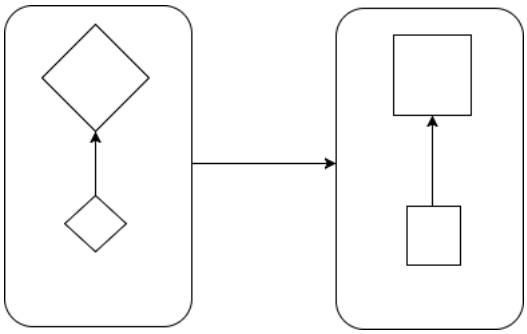

Figure 34: Dependency visualization

8.2.7 Different types of graphs

The aforementioned graphs could be categorized in three different types: tree, directed acyclic graph and general (cyclic) graph. Distinguishing graphs by these properties could make it easier to use them for analysis. Below we are listing the features that are used to categorize each of these graphs:

1. **Call graphs** are just **cyclic graphs**. They have a graph structure and it is possible for the graph to have cycles, meaning that e.g it would be possible for a function to call itself or another function.

2. **Class inheritance graphs** are **acyclic graphs**. In most popular OOP languages classes cannot inherit a class that already inherits that class. Therefore we have no cycles(making this graph acyclic). However, any class can inherit from any other class, so this graph could not be a tree.

3. **Containment graphs** are **trees**. Containment graphs represent a chain of software entities that contain other software entities. This means that if an element X is contained in another one Y then its properties are automatically contained.

4. **Build dependency graphs** are **acyclic** graphs. A node can have multiple incoming edges, but it can't create cycles since two dependencies can't depend on each other. E.g : If node A can depend on B then B cannot depend on A.

Having information regarding the type of the dependency graph that needs to be visualized could ease a lot of tasks and bring a lot of advantages. For example if we had information that the graph is a tree then we know that the structure of the tree is not complex like the one of the other graph types. It is easier to draw a tree without cluttering than to draw e.g a cyclic graph without cluttering. Trees don't have cycles and the child nodes don't have multiple parents, avoiding thus cyclic inheritance or additional complexities.

8.3 Conclusion

In this essay we proposed a new way of displaying the 4 types of dependency graphs. First we introduced a dependency data model to store different dependency graphs which was illustrated with a visualization of the stored data. This approach offers a new possibility for project managers or software architects to keep track of the dependency evolution of software and analyze complexity.