

UNIVERSITY OF GRONINGEN
Faculty of Mathematics and Natural Sciences
Department of Computing Science
Software Maintenance and Evolution

The Chromium Project

Analysis of a software repository

Ioakeimidis, Spyros
Manteuffel, Christian



1.0
Groningen, November 22, 2011

Contents

Contents	I
List of Figures	II
List of Tables	III
List of Listings	III
1 Introduction	1
1.1 About Chromium	1
1.2 Timeline	1
2 Basic repository investigation	2
2.1 The Chromium Repository	2
2.1.1 Revisions	2
2.1.2 Folder-structure	3
2.2 Authors	6
2.3 The renderer directory	9
2.3.1 Revisions	9
2.3.2 Folder-structure	9
2.3.3 Amount of Data	9
2.3.4 Purpose	10
3 First Visual Overview	11
3.1 Stable Development Periods	11
3.2 Unstable Development Periods	12
3.3 Current State	13
4 Authors Analysis	15
4.1 Overview	15
4.2 Most Active Developers	15
4.3 Important Authors	16
4.3.1 jam@chromium.org	17
4.3.2 aa@chromium.org	17
4.3.3 mpcomplete@chromium.org	18
4.4 Distribution of contributions	19
5 Code Size Analysis	20
5.1 Size of Code Files	20
5.2 Amount of Source Code Files	20
5.3 Source Code Files Investigation	21
5.4 Content API Refactoring	22
5.5 Average File Size Mystery	24
6 Complexity Analysis	25
6.1 High Complexity Files	25
6.2 Complexity Rate of Variation	26
6.3 Complexity and Activity Correlation	27
6.4 Complexity and File Size Correlation	27

7	Dependency Analysis - Explore a Scenario	30
7.1	Introduction	30
7.1.1	Problem	30
7.2	Data Model	31
7.3	Correspondence	32
7.4	Updating	33
7.4.1	Implementation	33
7.5	Visualization	33
7.5.1	Evolution of Build Dependency graphs	34
	References	37
	Appendix	37
	A1 Release dates	37
	A2 Command-line output	37
	b) Results Revision Analysis	37
	A3 Time-tracking	38

List of Figures

1	Release velocity of Chromium	2
2	Checkedout Chromium Repository	4
3	Chromium Repository without SVN Meta information	4
4	Only source-files (Colors changed)	5
5	A closer look at <i>src</i>	6
6	Specifying the range of revisions	7
7	Statistics	7
8	Statistics	8
9	Four most active developers in the second half of the project.	8
10	Structure of the renderer directory (source-code only)	9
11	Conceptual application layers	10
12	Stable Periods	11
13	Stable Periods - Trend View	12
14	Unstable Periods	12
15	Unstable Periods - Trend View	13
16	Current Unstable State	13
17	Current Unstable State - Trend View - File Count	14
18	Current Unstable State - Trend View - Commit Count	14
19	Overview of active authors in Renderer	15
20	Most active developers in Renderer	16
21	Color encoding of file types and folders	16
22	Activity of jam@chromium.org on file types (black)	17
23	Activity of jam@chromium.org on folders (black)	17
24	Activity of aa@chromium.org on file types (red)	18
25	Activity of aa@chromium.org on folders (red)	18
26	Activity of mpcomplete@chromium.org on file types (blue)	18
27	Activity of mpcomplete@chromium.org on folders (blue)	19
28	Contribution of the top 20%	19

29	Files Code Size - Trend View	20
30	Amount of Source Code Files in the Directory	21
31	Colors - Code Size	21
32	Fastest Growing Files	22
33	Fastest Shrinking Files	22
34	File count	23
35	Average File Size	24
36	Colors - Complexity	25
37	Most Complex Files	26
38	Files with Significant Decrease in Complexity	26
39	Files with Significant Increase in Complexity	27
40	Activity of Files with Complexity Metric	27
41	Metric Color Composer - Complexity & Code Size	28
42	Complexity and Code Size - Evolution View	28
43	Complexity - Evolution View	29
44	Code Size - Evolution View	29
45	Data Model	32
46	Build Graph (1)	34
47	Build Graph (2)	34
48	Build Graph (3)	35
49	Evolution of Build Impacts	35
50	Evolution of Build Costs	35
51	Colors - Building Impact, Building Cost	36

List of Tables

1	Release dates	37
---	-------------------------	----

Listings

1	Querying information about the latest revision	2
2	Querying information about the first revision	3
3	Commits in March 2010	6
4	Find commits by chrome-admin@google.com	8
5	Number of commits in renderer	9
6	Commit Messages that corresponds to refactoring of chrome/renderer	23
7	Pseudocode to find correspondent vertices	33
8	Information about the latest revision	37
9	Information about the first revision	38

1 Introduction

The following report investigates aspects of the software evolution of the Chromium open-source browser project.

It starts by providing a general overview of the project and its history in Section 1.1 and 1.2. Afterwards the repository structure of the project will be examined, focussing on revisions, folder structure and authors (cf. Section 2.1). However, operating on the whole repository requires a lot of time, for the tools to analyze the source code repository as well as to perform the manual in-depth investigation. Sadly, the time allocated to complete the project is limited. Therefore, a smaller part of the repository will be investigated in detail, namely *trunk\src\chrome\renderer*. In the following a more detailed analysis of the renderer-folder is conducted like development phases (cf. Section 3), in-depth author analysis (cf. Sectionrefsec:authorAnalysis), code size analysis (cf. Section 5), complexity analysis (cf. Section 6) and dependency analysis (cf. Section 7).

The document describes which task has been performed in order to verify a hypothesis and which tools has been used. For instance, a command line input is indicated by an `$`.

1.1 About Chromium

According to the homepage¹ of the project, Chromium is an open-source browser project that aims to build a safer, faster, and more stable way for all users to experience the web.

The Chromium open-source project emerged from the development of the Google Chrome Browser. When Google released the first version of the Chrome Browser in **September 2008** they open-sourced large parts of the project under the label Chromium².

The intention of Google is to develop the browser in an open-source manner under the Chromium label and release the Google Chrome Browser as their product by adding functionalities like an integrated Flash Player and PDF Viewer, the Google branding and an auto-update system. The naming of the projects reflects their relation since Chromium is the required chemical element to create Chrome (plating), which adds decorative value, provides corrosion resistance, eases cleaning procedures, or increase surface hardness³.

1.2 Timeline

Compared to other browsers like the Internet Explorer or Firefox, Chromium has a high release velocity. They tend to release a new version every one or two months, as illustrated in Figure 1. The chart is based upon values taken from the Wikipedia page of Chromium⁴. The concrete release date of each version can be found in Appendix A1.

¹<http://http://www.chromium.org/Home>

²[http://en.wikipedia.org/wiki/Chromium_\(web_browser\)](http://en.wikipedia.org/wiki/Chromium_(web_browser))

³http://en.wikipedia.org/wiki/Chrome_plating

⁴Concrete days were missing for release 2 and 6. Instead the middle of the corresponding month has been taken as value.

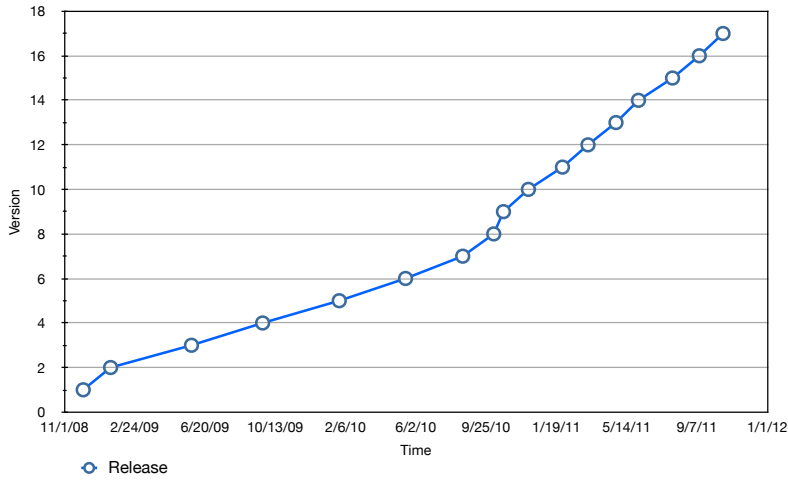


Figure 1: Release velocity of Chromium

The chart shows that after the release of version 8 the intervals between each consecutive version shortens. The reason for that can be either that the project got more attention by *increasing the time of man hours per release* or that *the number of features per release has been reduced*. A reason for the increase of the amount of man-hours could be that more developers joined the project or the existing developers had more time that they could spend on the project.

In general, the Chromium project constantly releases new version of their software. The graph shows no evidence that there has been a problem, which caused a significantly delay of a new version. Although it does not imply that there were no problems within the project.

2 Basic repository investigation

In the following section the structure of the Chromium repository and especially `src/chrome/render` are investigated.

2.1 The Chromium Repository

The first level of the Chromium repository is divided into three folders: Trunk, Branch and Releases.

The Release-folder contains dependency files for each release.

The Trunk is the main folder and always contains the latest version of the project.

The Branch folder contains copies of code derived from a certain point in the trunk.

2.1.1 Revisions

In order to get information about the revisions, the command-line tool `svn` is used. The command to get information about the latest version is illustrated in Listing 1 and respectively for the first Version in Listing 2. The results of the queries can be found in Appendix A2.

Listing 1: Querying information about the latest revision

```
$ svn info http://src.chromium.org/svn/ -r HEAD
```

Listing 2: Querying information about the first revision

```
$ svn info http://src.chromium.org/svn/ -r 1
```

There are **108459** revisions in the repository⁵. The latest one, the HEAD-revision, has been committed on the 3. Nov 2011 at 15:09 UTC by the user *rogerta@chromium.org*. The first revision has been checked-in on the 25. June 2008 at 23:13 UTC by the user *initial.commit*.

On the one hand, this information shows that the project is still active. On the other hand, the repository has been used roughly two months before Google made the Chrome source-code publicly available. Furthermore, the user *initial.commit* could be an indicator that the repository has been started on an existing code basis.

Besides the number of revisions, it would be interesting to know, how many minor releases have been made. It is assumed that the folder *releases* contains a list of all releases of Chromium. The following command returns 1942 as the number of releases within the folder, which means that on average every 56 commits a new version of Chromium is released.

```
1 $ svn list http://src.chromium.org/svn/releases |
   egrep '[0-9]+\.[0-9]+\.[0-9]+\.[0-9]+' | wc -l
2 1942
```

The first committed release is 2.0.169.0 on 10.03.2009. However, the earliest release is 1.0.154.53. The latest release is 16.0.911.2 (17.10.2011) although the latest committed release is 15.0.874.97 (18.10.2011).

```
1 svn log http://src.chromium.org/svn/releases
2 svn log http://src.chromium.org/svn/releases -l 10
```

2.1.2 Folder-structure

The analyzation of the folder-structure is conducted on the trunk folder, without considering branches/releases. The tool DaisyDisk⁶ has been used to get an overview of the repository. Therefore the repository has been checked out, which results in a 16GB large local working copy, as illustrated in Figure 2.

⁵At the time this report has been written.

⁶<http://daisydiskapp.com/>

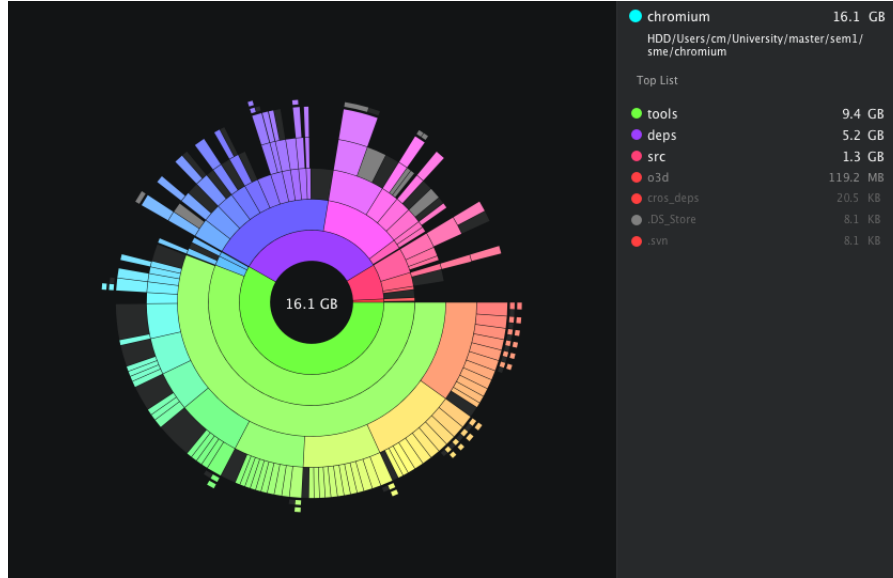


Figure 2: Checkedout Chromium Repository

After a short investigation of the largest folder of the project, it became obvious that a large amount of the total size is caused by meta information of subversion. Hence, the svn information has been removed from the repository, which reduces the size by approximately 50% (cf. Figure 3) .

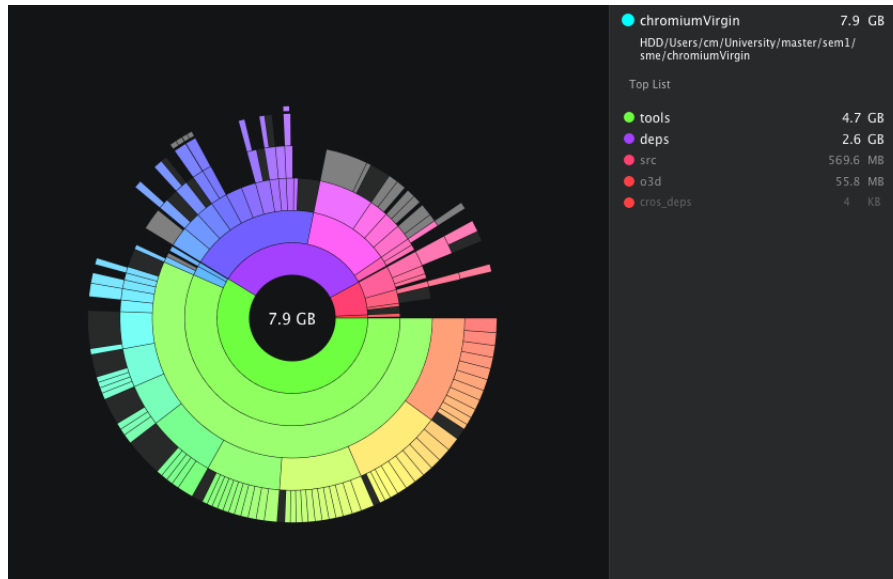


Figure 3: Chromium Repository without SVN Meta information

As illustrated in Figure 3 the largest folder is *tools* (59.5%), followed by *deps* (33%) and *src* (7.2%).

However, in order to perform an analysis of the evolution of the project, the folders that contain mostly source files need to be identified. In the first step, a list of file-endings has been created ,which are considered to be source files. These are cc,c,h,html,js,py,css,mm,sh,pl,cxx,hxx and pm.

In the next step the find command has been used to create a list of files that match the file-ending criteria. This list has been used to extract the files into a tar archive, because it keeps the folder structure. The unpacked tar-archive has been analyzed with DaisyDisk.

```
$ find . - type f - name "*.cc" - or - name "*.c" - or -
name "*.h" - or - name "*.html" - or - name "*.js" - or -
name "*.py" - or - name "*.css" - or - name "*.mm" - or -
name "*.sh" - or - name "*.pl" - or - name "*.cxx" - or -
name "*.hxx" - or - name "*.pm" > sources.include

$wc -l sources.include
47261

$ tar - cf onlysources.tar -I sources.include
```

The repository contains **47261** source-files, this number has been calculated by counting the lines in the list. As illustrated in Figure 4, the total size of source-code is **599.2 MB** (7.5% of the total size of the trunk).

In relation to source files, the *src* folder contains the most source code (46.5%), closely followed by *deps* (39%). The *tools* directory is now the smallest (12.3%). Since the *deps* folder mostly contains sources of third party libraries, the folder will not be further analyzed within the scope of this report.

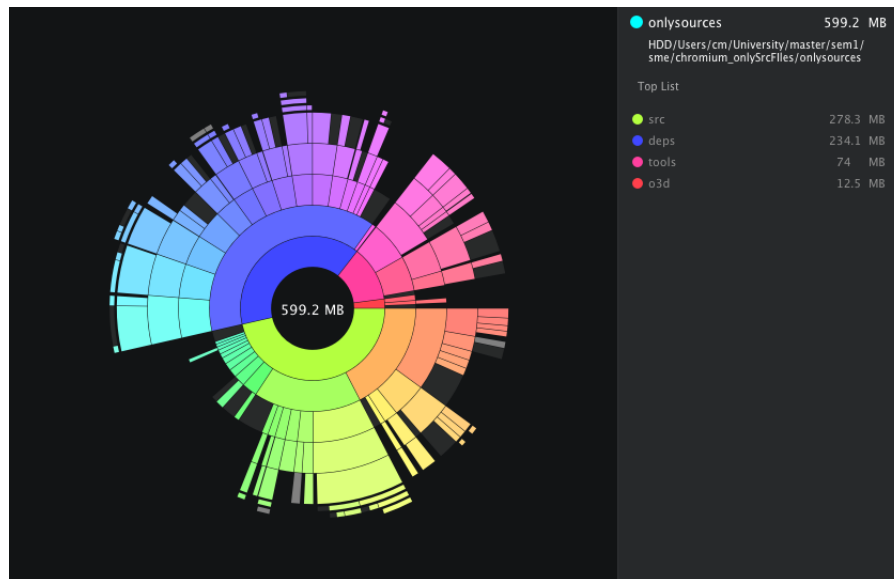


Figure 4: Only source-files (Colors changed)

In the next step the *src* directory will be further investigated. The size distribution of the various sub-directories of *src* is demonstrated in Figure 5. The directory *chrome* (104.2MB, 37.5%) and *third-party* (101.2MB, 36.3%) make up the biggest share, followed by many directories smaller than 15MB.

It can be concluded, that *trunk/src/chrome* is the folder with the largest share of source files and probably the most important one of the project. Although *third-party* has nearly the same size, its name indicates that the chromium project is not responsible for the development and maintenance of these sources.

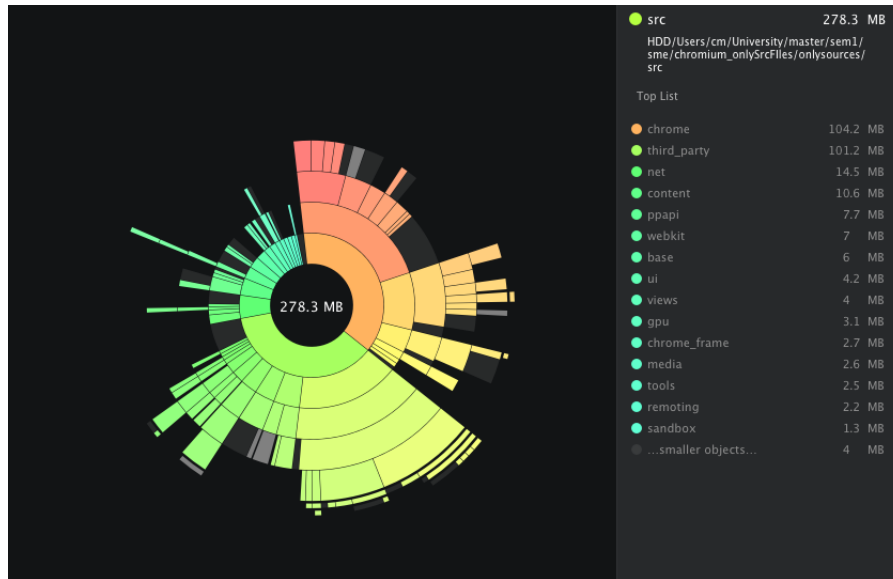


Figure 5: A closer look at *src*

2.2 Authors

In this sections a basic investigation of the authors is performed based on the *src* directory. The three most active developers of the first half and the second half of the project will be identified.

The first questions is, what is considered the midpoint of the projects evolution? It can be either based on revisions, which would be $\text{revisions}/2 = 108459/2 = 54229$ or based on time. Between the start of repository in July 2008 and November 2011, 41 months passed by. Hence, the time-based midpoint lies approximately in March 2010.

Considering the increase of release velocity presented in Section 1.2 , a revision-based midpoint probably differs from the date-based midpoint, because an increase of commits since September 2010 can be assumed. In order to verify this assumption, the revision number in March 2010 is identified (cf. Listing 3).

Listing 3: Commits in March 2010

```
$ svn log -r {2010-03-01}:{2010-03-31} | egrep lines
```

In fact, it turned out that the two midpoints differ. The date-based midpoint is somewhere between revision **40249-43137**. Just for convenience the mean value between the two has been taken, which is **41693**.

In order to investigate the three most active developers, the tool TortoiseSVN is used. After a right click on <http://src.chromium.org/svn/trunk/src> in the *Repository Browser*, the option *Show Log* appears. Within the Log the *Show Range* button in the bottom left corner is pressed. In this window *Start Revision* 41693 and *End Revision* 1 are selected (cf. Figure 6).

Start Revision

☐ HEAD revision

☒ Revision

End Revision

☐ HEAD revision

☒ Revision

OK Cancel

Figure 6: Specifying the range of revisions

A click on the button *Statistics* shows an overview of the repository. In the statistics the slider is moved to the left in order to show only the three most active authors. According to Figure 7, the 3 most active authors are:

1. estade@chromium.org
2. tc@google.com
3. darin@chromium.org

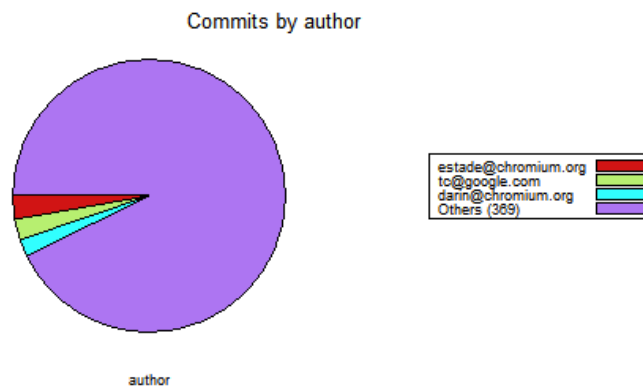


Figure 7: Statistics

The same is done for the second half of the project with a range from 41694 to HEAD. According to Figure 8, the 3 most active authors are:

1. tfarina@chromium.org
2. chrome-admin@google.com
3. thakis@chromium.org

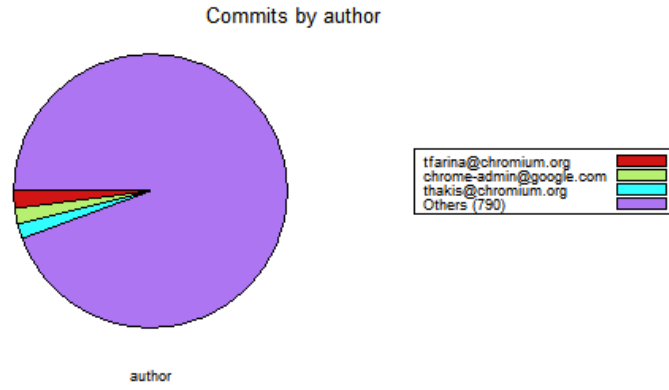


Figure 8: Statistics

However, the name of the developer *chrome-admin@google.com* indicates that this user is responsible for administrative tasks within the project. In order to prove that, a look at his commits has been performed (cf. Listing 4). It turns out that all commits contain the message "Update .DEPS.git", which would indicate that this user is actually some kind of cron job.

Listing 4: Find commits by chrome-admin@google.com

```
$ svn log | sed -n '/ chrome-admin@google.com/,/-----$/ p'
```

As a result the analyzation has been performed again but this time displaying the four most active developers (cf. Figure 9). Without chrome-admin these are:

1. tfarina@chromium.org (Thiago Farina)
2. thakis@chromium.org (Nico Webe)
3. thestig@chromium.org (Lei Zhang)

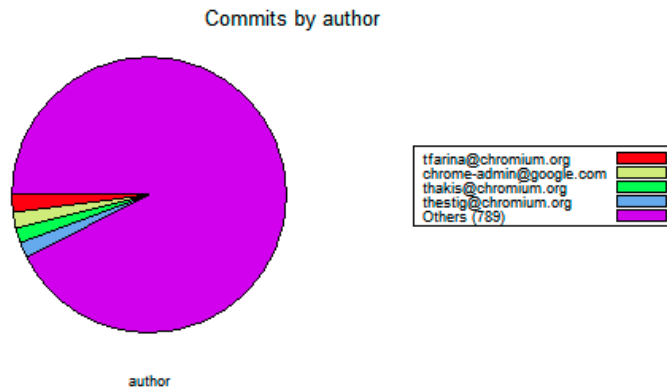


Figure 9: Four most active developers in the second half of the project.

Besides the fact that the three most active developers changed in the second half of the project, it is also noteworthy that the results support the hypothesis that the higher development velocity is based on more developers joining the project. In the first half the total number of developers was **372** in the second half this number increased to **793**.

2.3 The renderer directory

A complete in-depth investigation on the whole src directory would be very time-consuming. Hence, one sub-directory within the src has been picked, that is analyzed in detail.

The directory src/chrome/renderer has been chosen. However, the decisions has been made without any prior knowledge about the contents or purpose of this directory.

In the following sections, the results of a first basic investigation of this sub-directory are exemplified.

2.3.1 Revisions

The first revision is 15 (2008-07-27 01:55 UTC) the last 106880 (2011-10-22 03:47). A total of **4446** commits have been made in the renderer directory (cf. Listing 5).

Listing 5: Number of commits in renderer

```
$ svn log chrome/ renderer | grep ' r[0-9][0-9]* ' | wc -l
4446
```

2.3.2 Folder-structure

Share on total amount of source code in the trunk is 0.3% and 0.6% of the src. The directory contains 94.7% source-files (1.8MB of 1.9MB according to DaisyDisk).

From this follows that the findings of this particular directory cannot be applied to the whole chromium project, since the amount of analyzed data is too small.

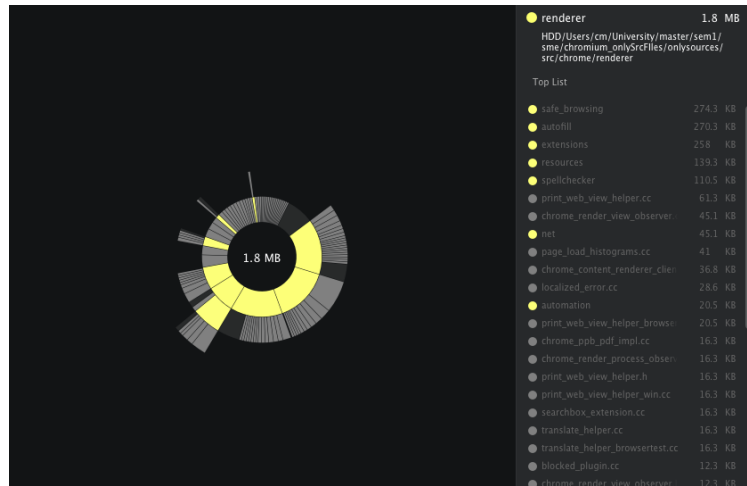


Figure 10: Structure of the renderer directory (source-code only)

2.3.3 Amount of Data

The numbers presented in the previous section can be used to estimate the maximum amount of data that needs to be analyzed. It is assumed that all files need to be checked out for each consecutive version of the renderer directory.

Furthermore, it is assumed that the code size is constantly growing, hence 1.8MB is the maximum code size and the size of all prior version is less or equal.

The upper bound of data that needs to be analyzed can be set to 8GB, based on the following estimation ($S_{\max} * V = 1.8MB * 4446 = 8002.8MB \approx 8GB$, where V is the number of versions and S the code size)

2.3.4 Purpose

In order to find out the purpose of this particular directory, the users in the official IRC channel #chromium has been asked. They pointed to the architectural documentation on the homepage for further information.

The following extract, describes the purpose of the files in chrome/renderer.

Renderer / Render host: This is Chromium's "multi-process embedding layer." It proxies notifications and commands across the process boundary. You could imagine other multi-process browsers using this layer, and it should not have dependencies on other browser services.⁷

However, many files that have been mentioned in this documentation could not be found in this directory but instead they were located in content/public/renderer. Probably, files have been moved to that directory.

In Figure 11 the conceptual application layers of the rendering process are depicted⁸.

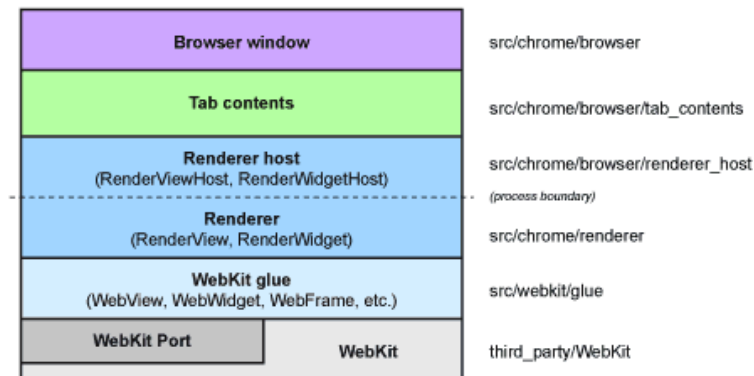


Figure 11: Conceptual application layers

⁷<http://www.chromium.org/developers/design-documents/displaying-a-web-page-in-chrome>

⁸Image taken from <http://www.chromium.org/developers/design-documents/displaying-a-web-page-in-chrome>

3 First Visual Overview

In this section, the stable and unstable periods of renderer are investigated. A central research question is, if the project is in a stable state at the time of investigation.

A stable state can be defined as a period of time in which the code does not change its content or size. An unstable state is a period of time, in which a source code file changes either its content or size.

The investigation has been performed by utilizing the evolution view functionality of SolidTA. In this view the horizontal axis (X) represents time, whereas the vertical axis (Y) represents the files.

3.1 Stable Development Periods

The files have been sorted by *Creation Time* in order to get a basic overview of the creation and evolution of files during time.

Additionally, the *Code Size* metric has been applied to extract information about changing file-size. This is necessary, in order to identify growing or shrinking files. A prerequisite for the code size metric is to execute the *CCCC metric calculator*. The code size metric has been adjusted to indicate source code files larger than 350LOC⁹ in a red color.

As illustrated in Figure 12, it can be observed that this part of the project has many small sized files and only a few files (≈ 7) are above 350LOC

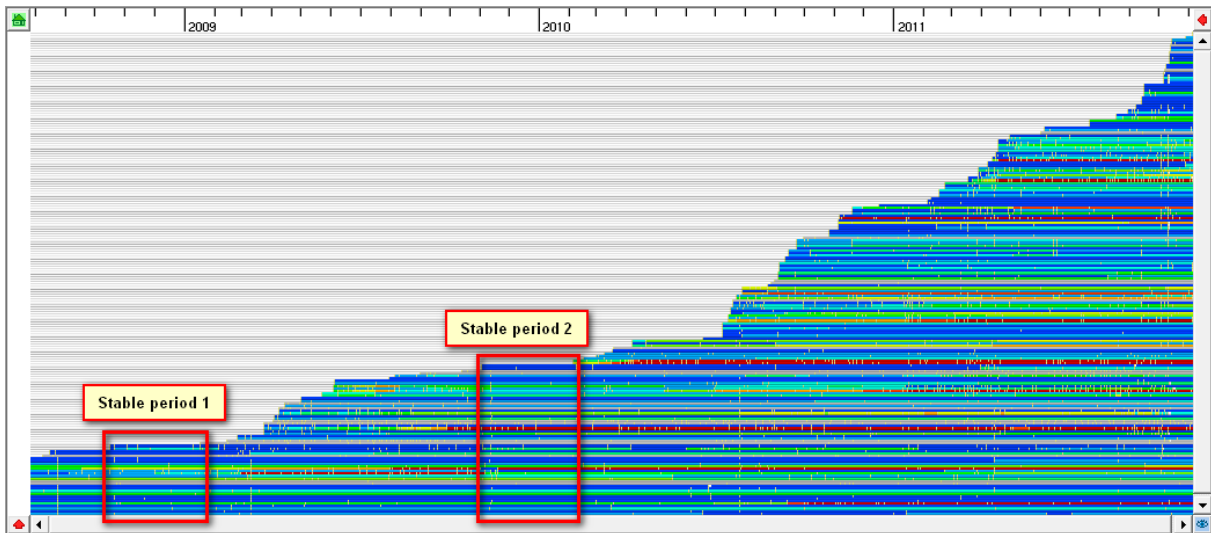


Figure 12: Stable Periods

Figure 12 depicts two stable development periods for the renderer directory. These periods indicate that no files have been added to the repository nor do the existing files change in terms of code size. The conclusion has been drawn because the gradient of the graph is zero and the color indicating the code size does not change. Furthermore, the activity is very low as indicated by the small yellow bars, which represent commits.

⁹Lines of Code

The results are supported by the trend view as shown in Figure 13. The sampling interval has been set to one month, the bar chart style has been chosen and the filter has been set to *average value* of the size of files.

During the stable periods, which are outlined by the red rectangle, it can be seen that the average file size stays almost the same. This means that neither more files are added in the repository, nor significant changes have been made, so for the average file size to change.

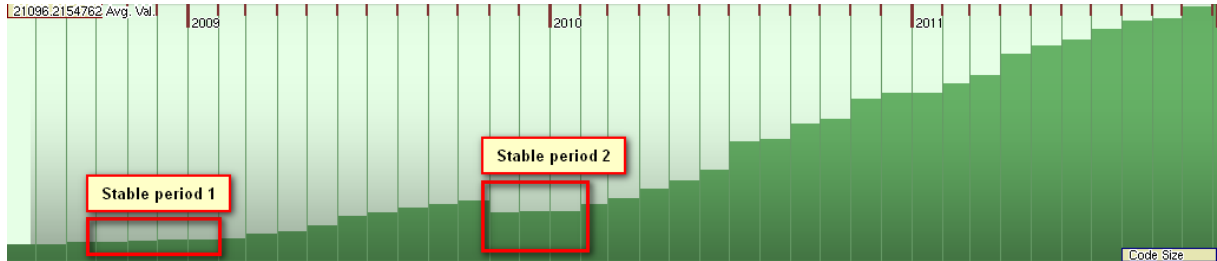


Figure 13: Stable Periods - Trend View

3.2 Unstable Development Periods

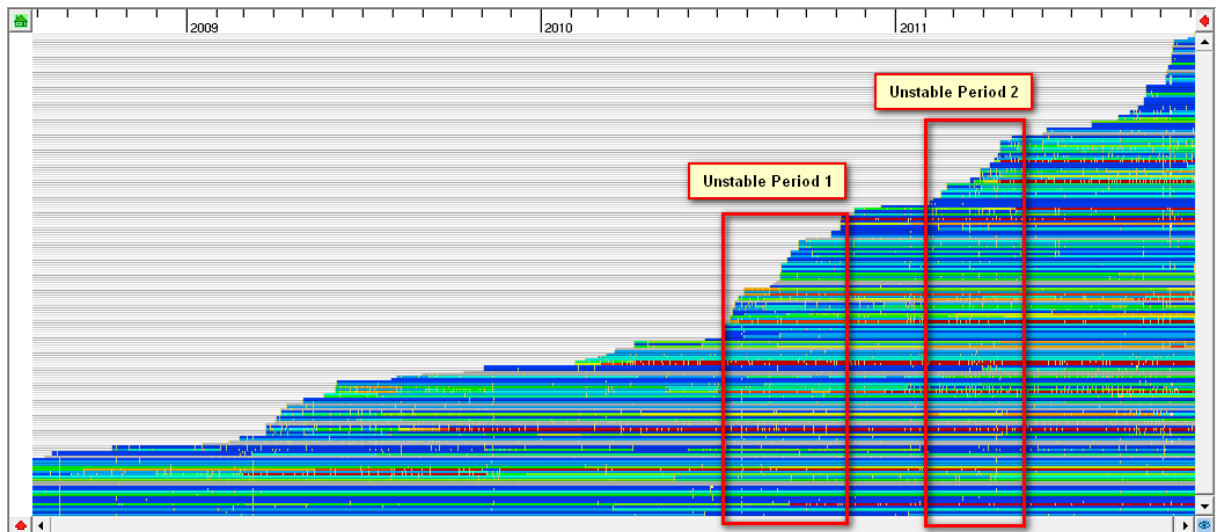


Figure 14: Unstable Periods

Figure 14 shows two unstable development periods for this part of the project. During these periods many new files are being added to the repository. Also during both of these periods almost 40% of the files are either growing or decreasing by size because of the fact that the color of these files is changing.

During the whole period, there are indeed other unstable periods. However, only those periods have been outlined in which the code undergoes intense changes.

In order to analyze the unstable periods in depth, the trend view has been utilized (cf. Figure 15). The sampling interval is one month. The *bar chart* style and the *average value* filter have been selected. Throughout these two periods it can be seen that the code size has significantly increased and hence the periods are considered unstable.

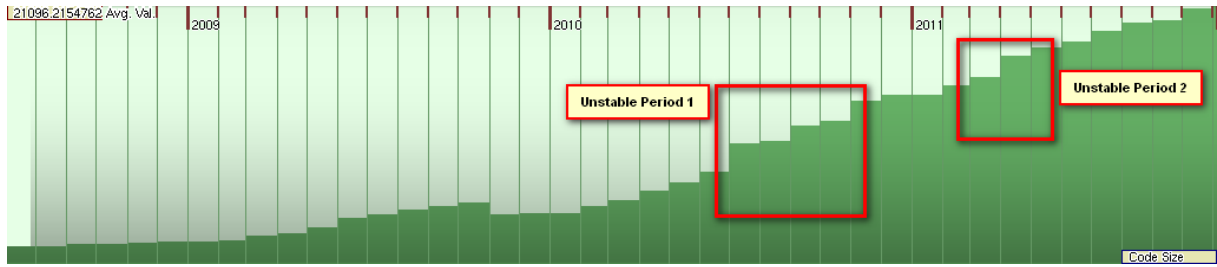


Figure 15: Unstable Periods - Trend View

3.3 Current State

During the basic repository investigation a continuously number of commits in the repository have been noticed. This leads to the hypothesis that the code might not be in a stable state currently¹⁰.

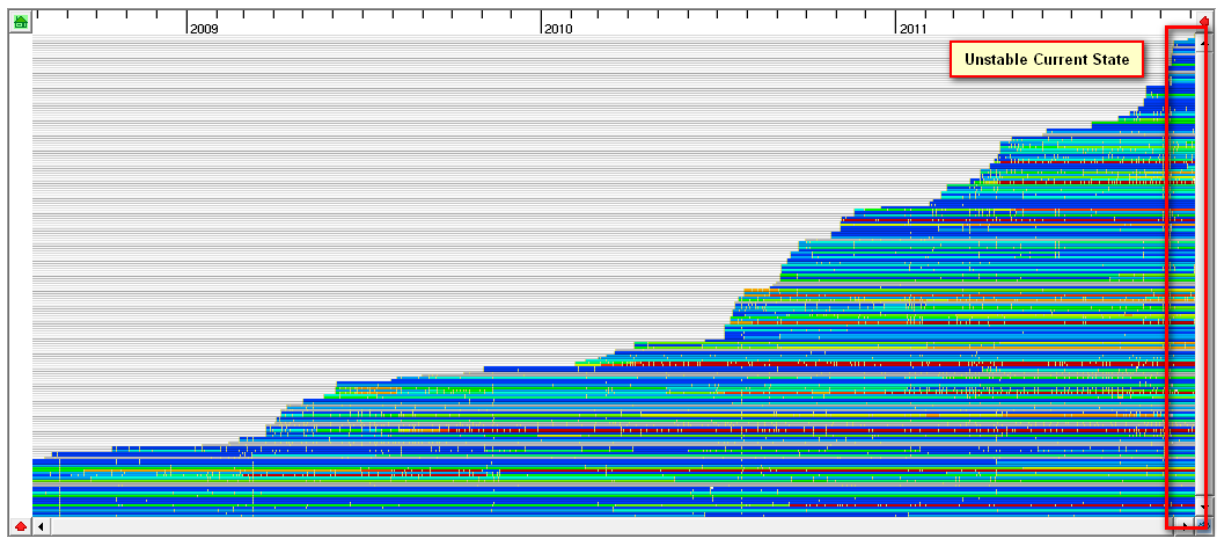


Figure 16: Current Unstable State

As shown in Figure 16 the code is not in a stable state for the following reasons. First, many files are being added to the repository. Second, a lot of files have significantly changed because they often change color.

Again, the trend view has been utilized for a further investigation as shown in Figure 17 and Figure 18.

Figure 17 uses the *file count* option and Figure 18 the *commit count* option.

¹⁰The last month of the project

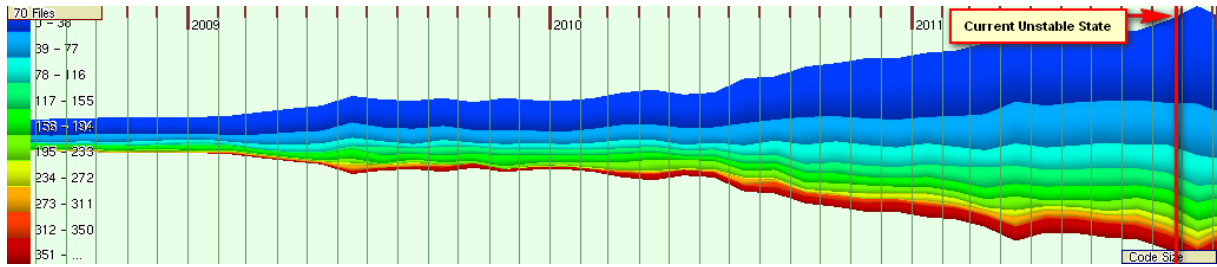


Figure 17: Current Unstable State - Trend View - File Count

Figure 17 shows that the number of files is not stable during the last month. Furthermore, it is clear that the code is not in a stable state. This verifies the initial Hypothesis. However, from Figure 17 two new hypotheses emerge.

Due to the fact that approximately 50% of renderers are files with 75 LOC at maximum, it is assumed that the code might not need refactoring and that it is in a good state and that it has a low complexity.

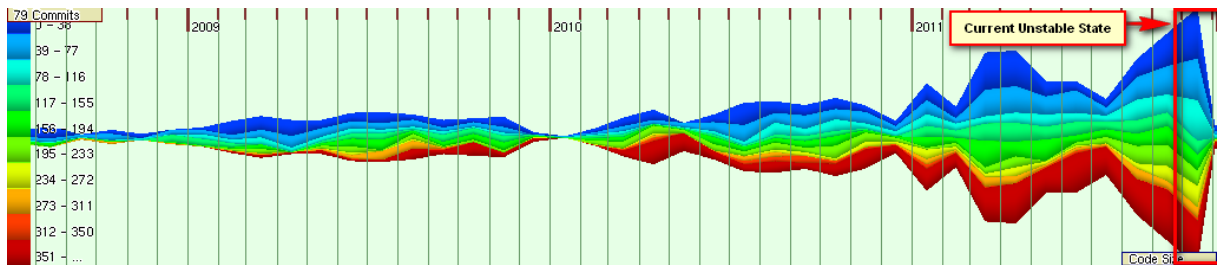


Figure 18: Current Unstable State - Trend View - Commit Count

In Figure 18 it is emphasized that during the last month the number of commits is not stable. After a peak around the middle of October, the number of commits plunged to almost zero.

4 Authors Analysis

In this section the authors that contributed to chrome/renderer are analyzed. Therefore the evolution view of Solid TA and the author, code size and file type metric has been used.

4.1 Overview

According to the statistics function of TortoiseSVN, the total number of distinct developers within the renderer subpart is **366**. However, the number does not consider developers that changed their username during the project e.g. mpcomplete@chromium.org and mpcomplete@google.com.

In general, it can be assumed that there is one main developer. However, as illustrated in Figure 19, a main developer can not be clearly identified. Taken as a whole, the graph indicates that many different authors contributed code during different periods of time.

Nevertheless, five periods can be identified that indicate a change in active developers. These periods are marked with red bars. For instance, in the transition period in the middle of 2010 the author indicated by a green color became very active. Another example would be early 2011, when the yellow author (jam@chromium.org) became very active. Actually, based on the amount of files, it can be assumed that the yellow author took over the responsibility for this component.

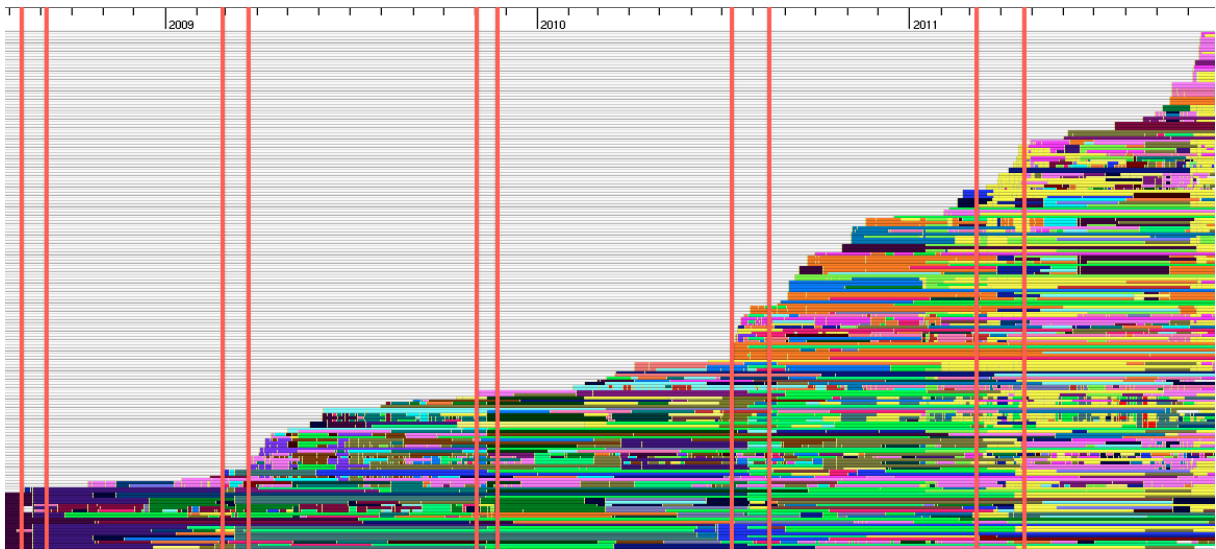


Figure 19: Overview of active authors in Renderer

4.2 Most Active Developers

Based on the author analysis in Section 2.2, the hypothesis has been created that the main developers of the overall project are also among the main developers of this subpart of the project.

In order to verify this hypothesis the author metric of SolidTA has been used in conjunction with *Show # Version* and *Sort By Version*. An extract of the resulting list of authors is depicted in Figure 20.

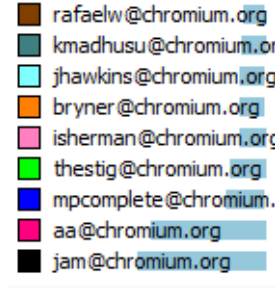


Figure 20: Most active developers in Renderer

The initial hypothesis, that the most active developers of the whole project are also active in the renderer subpart, is valid.

1. thestig@chromium.org (Rank 4)
2. darin@chromium.org (Rank 12)
3. estade@chromium.org (Rank 16)
4. tfarina@chromium.org (Rank 32)
5. tc@google.com (Rank 33)
6. thakis@chromium.org (Rank 56)

4.3 Important Authors

As indicated in Figure 20 the top three active authors of the renderer component are:

1. jam@chromium.org
2. aa@chromium.org
3. mpcomplete@chromium.org

Because of the name, mpcomplete@chromium.org has been considered to be a bot that does some kind of completion task. After looking at his commit logs, it turned out that it is a regular author.

In the following Section the three users will be described in more detail. Therefore two diagrams have been created for each user. The first one shows the activity of the user on various file types (From top to bottom: cc,h,html,js,png,other). The second one shows the activity of the user inside of the sub-folders. The color encodings are illustrated in Figure 21.

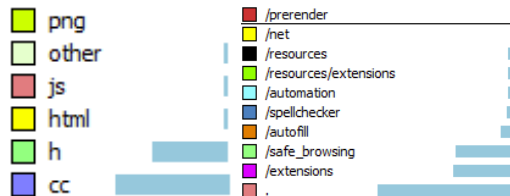


Figure 21: Color encoding of file types and folders

For convenience the @chromium.org part of the username is not mentioned each time. Hence aa refers to aa@chromium.org.

4.3.1 jam@chromium.org

Jam is the most active author in chrome/renderer. As illustrated in Figure 22, it can be seen that jam became first active at the end of 2009, but he didn't contribute that much. With the beginning of 2011 his activity increases drastically. However, he works on new files but also on very old files that have been created in 2008. He is mainly active in .cc or .h files.

In accordance with the findings in Section 5.4, it can be assumed that he was responsible for the refactoring.

However, the initial assumption that jam is responsible for the component at least since 2011, is supported by the fact that he is active in all sub folders (cf. Figure 23).

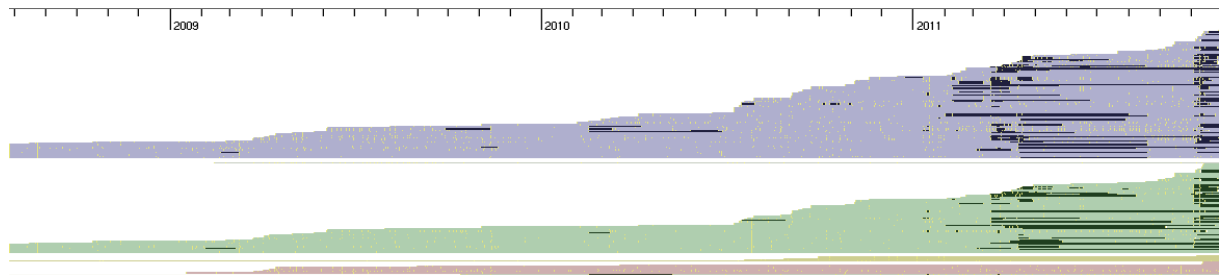


Figure 22: Activity of jam@chromium.org on file types (black)

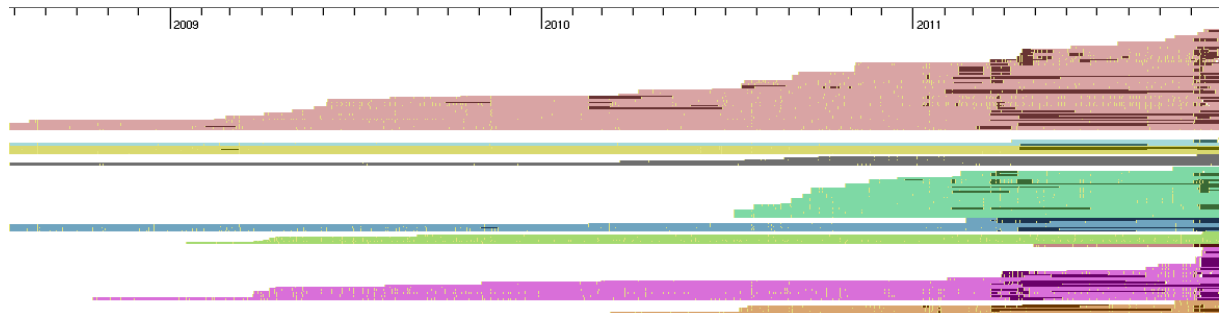


Figure 23: Activity of jam@chromium.org on folders (black)

4.3.2 aa@chromium.org

Aa is the second most active author and is as nearly as active as jam. However, by looking at his activity in Figure 24, it can be seen that he also became active around 2009, but in contrast to jam he is constantly contributing to chrome/renderer. Furthermore, he is also contributing large parts of JavaScript code and some C-Code and C-Header-files.

In contrast to jam, the author aa mainly works on files inside of the subfolder extensions or resources/extensions (cf. Figure 25). It could be the case that aa is a developer that is especially familiar with the extension mechanism of Chromium.

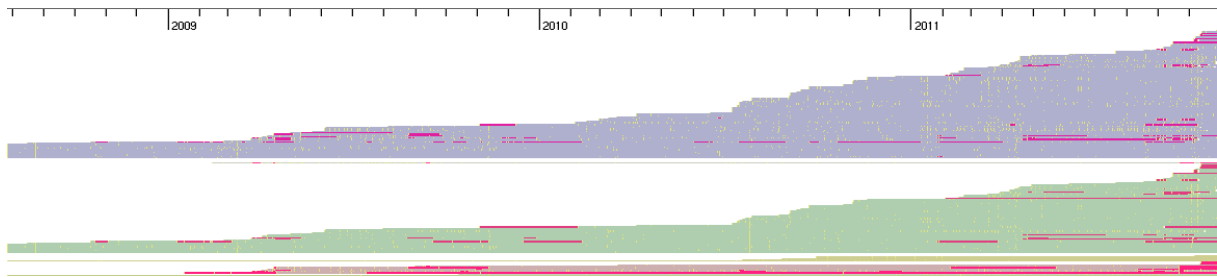


Figure 24: Activity of aa@chromium.org on file types (red)

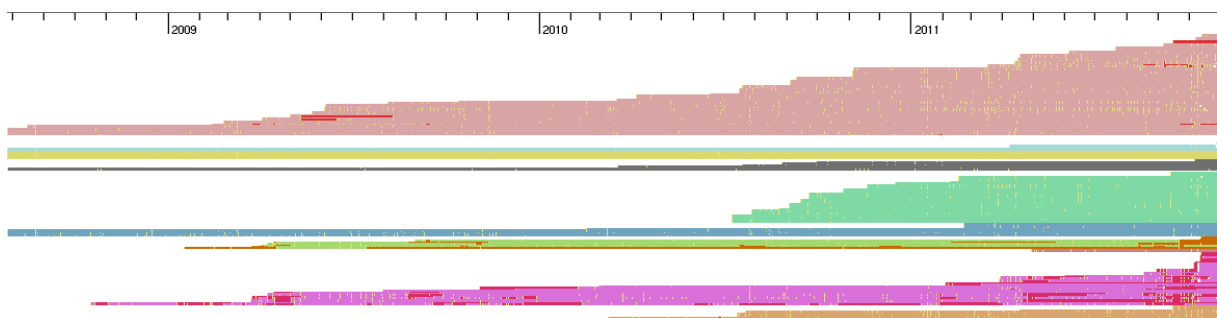


Figure 25: Activity of aa@chromium.org on folders (red)

4.3.3 mpcomplete@chromium.org

Mpcomplete is the third most active author, although he is clearly less active than jam or aa. As illustrated in 26, he became active in the middle of 2009, working on cc,h and js files.

In contrast to aa and jam, he only works on file that have been created between 2009 and 2010 but similar to aa, mpcomplete works mainly on files inside the folders extensions and resource/extensions (cf. Figure 27).

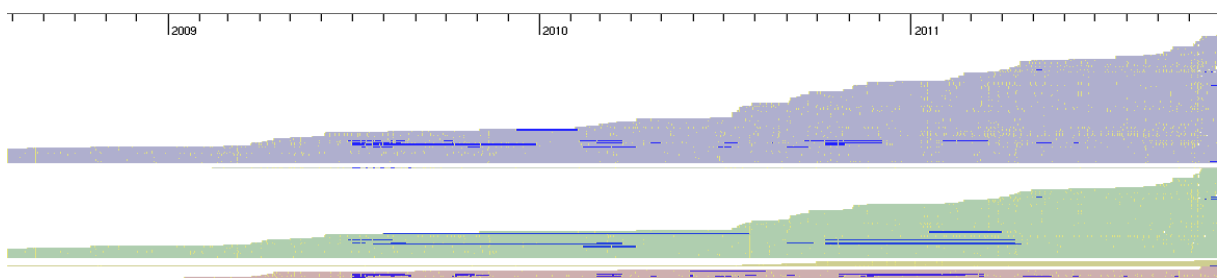


Figure 26: Activity of mpcomplete@chromium.org on file types (blue)

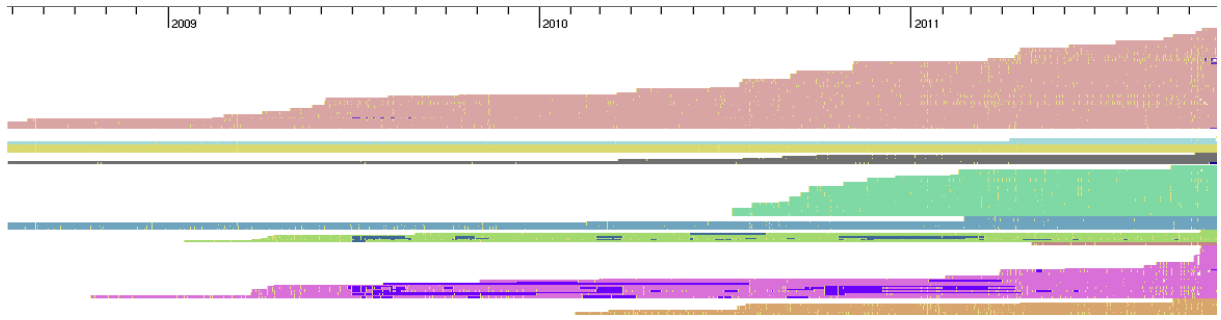


Figure 27: Activity of mpcomplete@chromium.org on folders (blue)

4.4 Distribution of contributions

The author list in SolidTA suggests that a relatively small number of users have contributed a large amount of source code (cf. 80-20 Rule or Pareto-principle) (cf. Figure 20).

In order to verify this hypothesis, the evolution view of SolidTA has been utilized by selecting the top 20% ($0.2 * 366\text{Authors} \approx 73\text{Authors}$) of the authors. As illustrated by Figure 28, only a few gray spots are visible. This supports the 80-20 hypothesis.

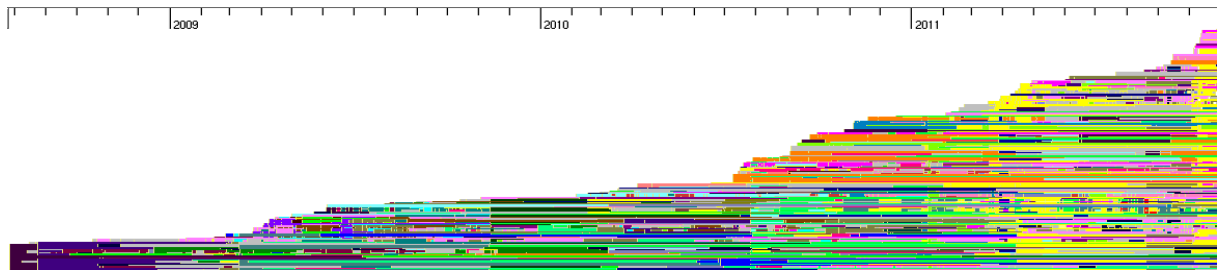


Figure 28: Contribution of the top 20%

5 Code Size Analysis

In this section, the code is analyzed with regards to its size. The *CCCC metric calculator* as well as the *Lines of text counter* have been already executed in the beginning. Again, the trend view with a sampling interval of one month is used.

5.1 Size of Code Files

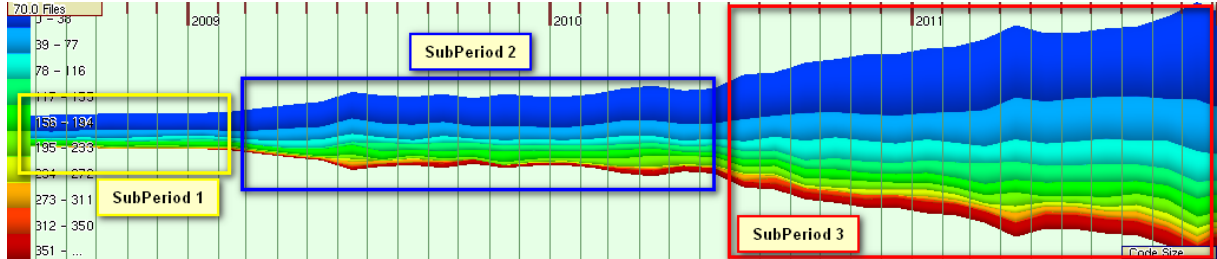


Figure 29: Files Code Size - Trend View

Figure 29 shows the trend view with the selection of *Flow* type and *File count*. According to Figure 29 the general idea is that the size of the files keeps increasing during the whole period. The graph has been split in three sub-periods.

- sub-period 1

This period lasts from the initial commit in the middle of 2008 until the middle of February of 2009. During this period the size of the files remains approximately the same.

- sub-period 2

This period starts in the middle of February of 2009 and ends in the middle of June of 2010. During the first three months of this period, the code size increases constantly but only very slow. During the next months the code size slightly fluctuates, with a small peak in March of 2010.

- sub-period 3

The last period starts in the middle of June of 2010 until the end of the investigated period. Throughout this period there is a steady increase in the size of code files, with two small exceptions. The first one is in April of 2011 when there is a slight peak. The second one is during the last two months of the whole period when the size of the code files is showing a considerable increase.

A comparison of Figure 29 and Figure 1 shows that the growth rate of code size increased at the same time the release velocity of Chromium increased. One reason for this is probably because more authors joined the development of the project, hence more code is produced and new releases are realized in shorter time. However, this hypothesis has not been further investigated.

5.2 Amount of Source Code Files

In order to investigate the amount of source code, the *Code Size* metric is used. The files have been grouped in the evolution view based on the *Code Size* attribute as shown in Figure 30.

The files in the evolution view have been sorted based on *File Type*. Figure 30 shows that approximately 90% of the files are source code files with extension .cc and .h.

The difference to the previously mentioned number of 94.7% in Section 2.3.2 is because SolidTA does not analyze source code files with other extensions that exist in this part of the project. That is exactly the reason of the existence of gray color files in the lower part of Figure 30.

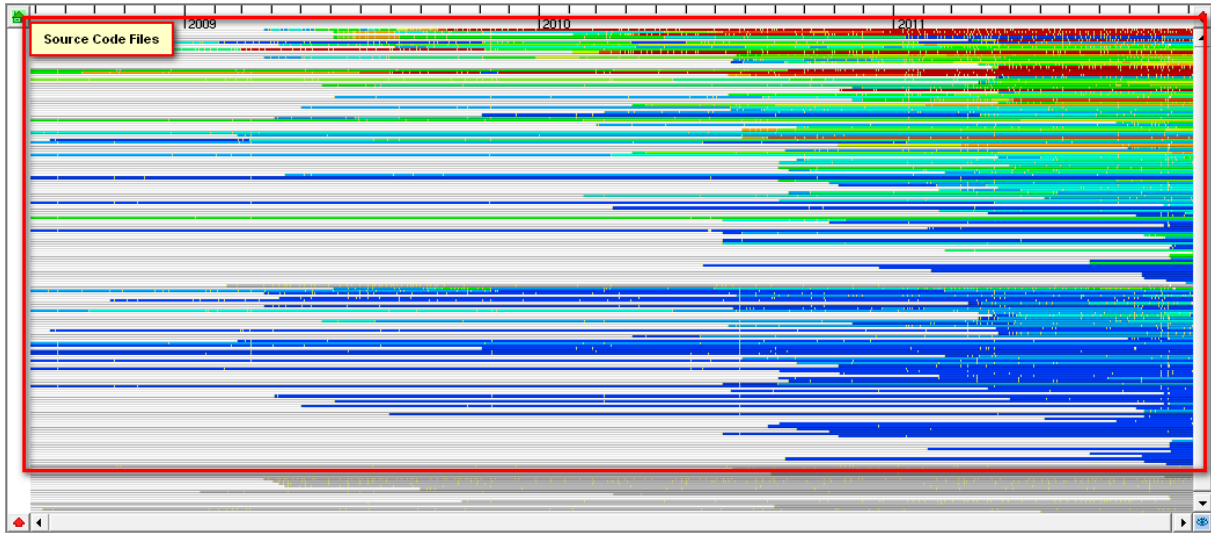


Figure 30: Amount of Source Code Files in the Directory

5.3 Source Code Files Investigation

From Figure 29 it can be distinguished that the size of source code files is actually shrinking on the average. This is due to the fact that small sized source code files are added to the project, which leads to the decrease of the average size. The relation of code size to color is illustrated in Figure 31.

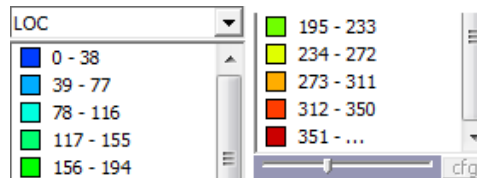


Figure 31: Colors - Code Size

There are two lists below. The first one contains the fastest growing files and the second one the files that shrink the most, both in terms of code size.

The evolution view is used along with the *Code Size* metric in order to identify files, which grow rapidly in code size. Fast growing files are those files that increase their code size in a relatively small amount of time, e.g. change from a blue to a dark red color within a few months. Figure 32 shows these fast growing files. It can be seen that the size of the code of these files considerably increases with particular commits..

- chrome_render_view_observer.cc
- extension_dispatcher.cc
- extension_helper.cc
- chrome_content_renderer_client.cc
- form_autofill_util.cc
- form_autofill_browsertest.cc
- extension_process_bindings.cc

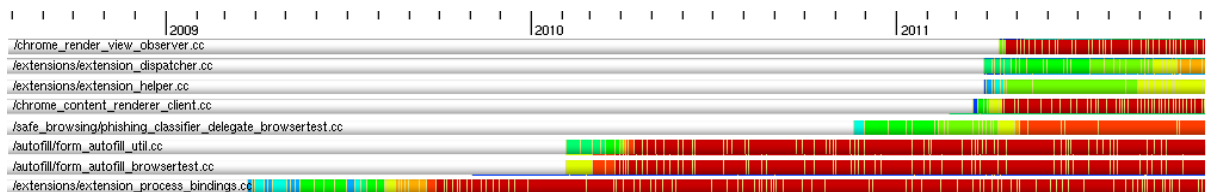


Figure 32: Fastest Growing Files

Files that shrink the most can be defined as files that show a fast and significant change from large sized related colors to small sized related colors. Figure 33 shows the files that shrink the most. However, we can see that there are not many files that have a fast and significant decrease in their size.

- chrome_render_view_observer.h
- event_bindings.cc



Figure 33: Fastest Shrinking Files

5.4 Content API Refactoring

In Section 2.3.4, it has been stated that a lot of files that are mentioned in the documentation actually do not exist in the chrome/renderer directory. The hypothesis has been drawn that files have been moved, which would indicate a refactoring of this component.

In order to verify this hypothesis, the tool StatSVN¹¹ has been used, because it also considers the number of files during the evolution. The resulting Graph is illustrated in Figure 34.

This Graph clearly indicates that a lot of files has been removed between February and April 2011. The file count is dropping from nearly 300 to approximately 175 files.

¹¹<http://www.statsvn.org/>

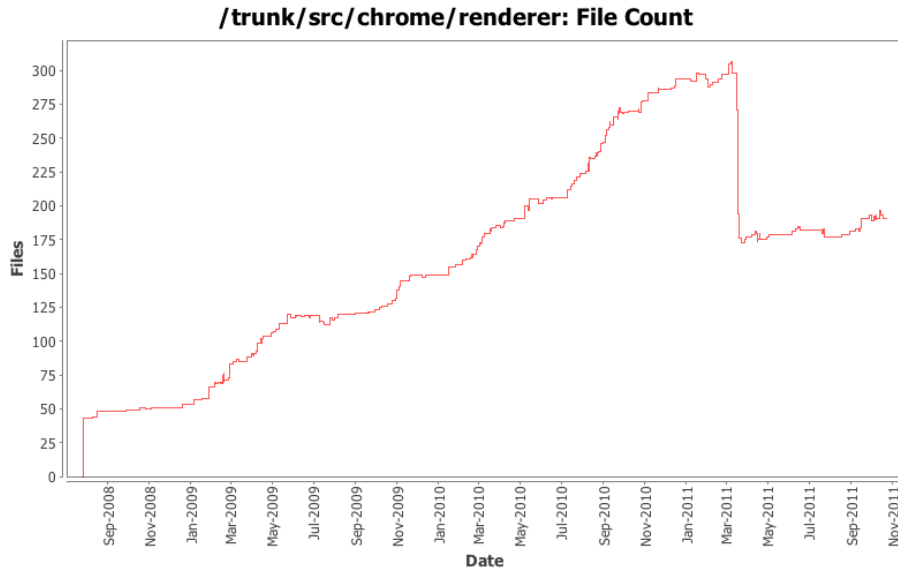


Figure 34: File count

A reason for this massive decrease of files must be reflected in the commit messages of renderer. Hence, the commit logs of the month March 2011 have been analyzed in order to find commits that involve the deletion of many files. The found messages are outlined in Listing 6.

Listing 6: Commit Messages that corresponds to refactoring of chrome/renderer

```

1  jam@chromium. org 2011-03-19 02:43 #147
2  Rev.: 78781
3  Move a bunch of remaining files from chrome\renderer to content\
   renderer.
4  ---
5  jam@chromium. org 2011-03-18 18:51 #141
6  Rev.: 78703
7  Move render_widget files to content.
8  ---
9  jam@chromium. org 2011-03-18 03:33 #134
10 Rev.: 78643
11 Move the renderer_web* files to content.
12 ---
13 jam@chromium. org 2011-03-17 21:15 #128
14 Rev.: 78579
15 Move a bunch of gpu/worker/plugin renderer code to content. I
   temporarily disabled the sad plugin code while I add a renderer
   chrome interface in a follow up.
16 ---
17 jam@chromium. org 2011-03-17 00:30 #117
18 Rev.: 78449
19 Move a bunch of html5 renderer code to content.
20 ---
21 jam@chromium. org 2011-03-16 22:40 #115
22 Rev.: 78422
23 Move core renderer subdirectories to content.

```

The messages indicate that many files have been moved from chrome/renderer to content/renderer. With this knowledge, the chromium documentation has been searched. In concrete, Google has been used with the following search query: "content/renderer" site:dev.chromium.org.

An investigation of the results shows that the chromium project introduces a new API in order to "isolate developers working on Chrome from inner workings of content" and to "make the boundary between content and chrome clear to developers and other embedders"¹².

Furthermore, it is stated that the chrome subdirectory should only provide generic APIs like Extensions, SpellCheck, Autofill, Prerendering, Safe Browsing, Translate¹³. The documentation also indicates that the refactoring is complete: "The current status (as of 9/19/2011) is content doesn't depend on chrome at all."

All in all, the hypothesis of a refactoring and especially a code movement is verified.

5.5 Average File Size Mystery

During the investigation of the refactoring another graph has been found, which has been created by StatSVN. The graph shows the evolution of the average file size (cf. Figure 35).

It can be seen that the average file size increases drastically from 300 LOC to 425LOC. The time of this change actually corresponds to the time of the refactoring mentioned in Section 5.4. Hence, it can be assumed that there is a relation between the increase of file size and the deletion of files.

A possible explanation could be that many small files have been moved to content/renderer and the big files like images remain in the chrome/renderer directory. Thus, large files would have more weight in the calculation of the average file size.

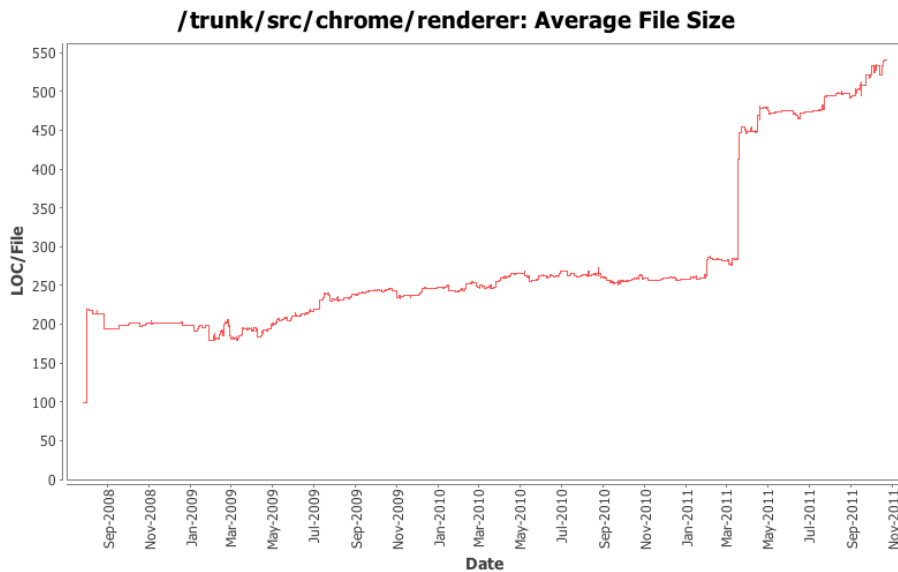


Figure 35: Average File Size

¹²<http://dev.chromium.org/developers/content-module/content-api>

¹³Extracted from <http://dev.chromium.org/developers/content-module> at 11/20/11

6 Complexity Analysis

In this section a complexity analysis on renderer is performed. For the analysis the *McCabe complexity* metric is used, which has been computed using the *CCCC metric calculator*. This metric works only with source code files and hence a selection of source code files has been created.

As aggregation of this metric the *Total complexity* has been chosen, in order to perform the analysis on file level. The slider has been adjusted in a way that the range of values of the red color is higher than 90. The red color of course means high complexity. This value has been chosen due to the fact that the total range of complexity is high on file level. Figure 36 shows the relation between colors and complexity.

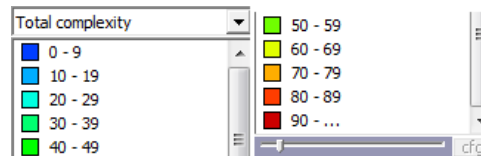


Figure 36: Colors - Complexity

6.1 High Complexity Files

Very complex files are those files whose color is red during the end of the period in the evolution view. The *fit to line zoom* button allows to view the files in detail and check their complexity. The following list presents the most complex files in the project (cf. Figure 37).

- form_autofill_util.cc
- password_autofill_manager.cc
- chrome_content_renderer_client.cc
- chrome_render_view_observer.cc
- extension_process_bindings.cc
- page_load_histograms.cc
- print_web_view_helper.cc
- searchbox_extension.cc
- translate_helper.cc

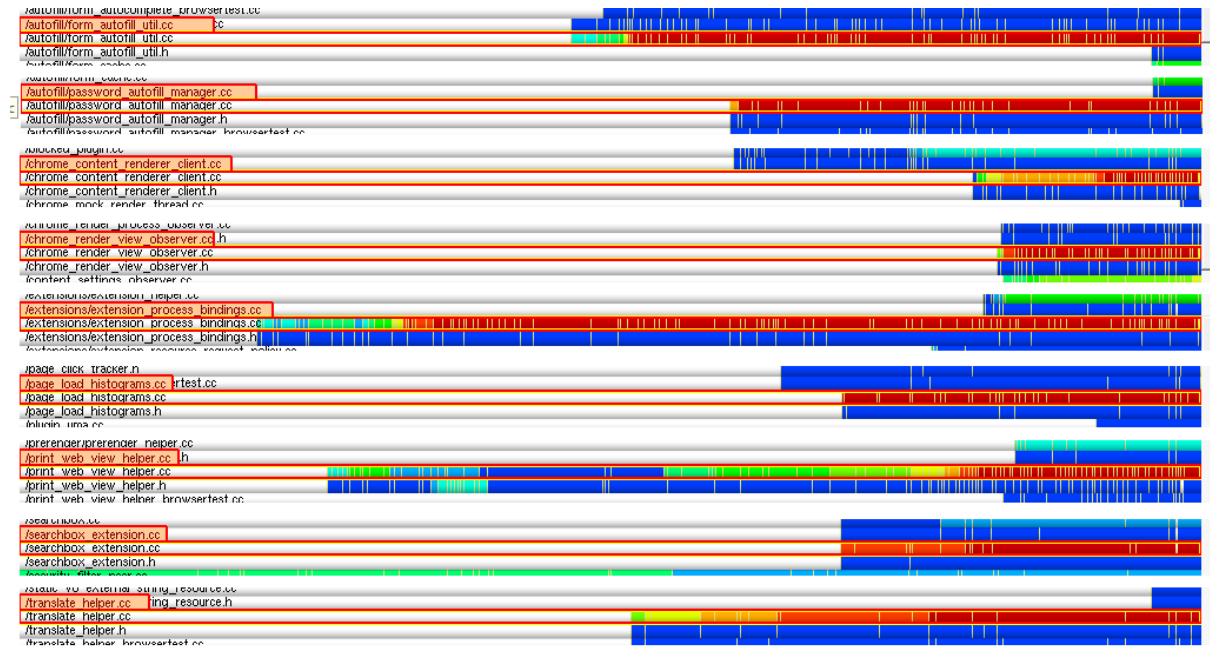


Figure 37: Most Complex Files

6.2 Complexity Rate of Variation

To investigate a significant decrease or increase of complexity of files, the *fit to line zoom* button is used again. 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. Figure 38 shows those files with a significant decrease in complexity as well as the list below.

- print_web_view_helper_win.cc
- event_bindings.cc
- user_script_slave.cc

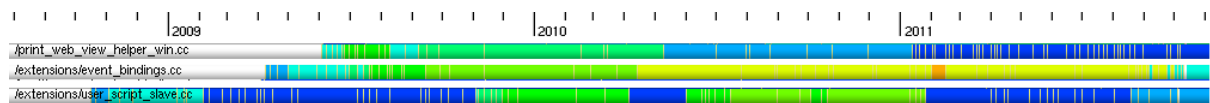


Figure 38: Files with Significant Decrease in Complexity

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. Figure 39 shows those files with a significant increase in complexity as well as the list below.

- chrome_render_view_observer.cc
- chrome_content_renderer_client.cc
- translate_helper.cc
- form_autofill_util.cc
- print_web_view_helper.cc
- extension_process_bindings.cc

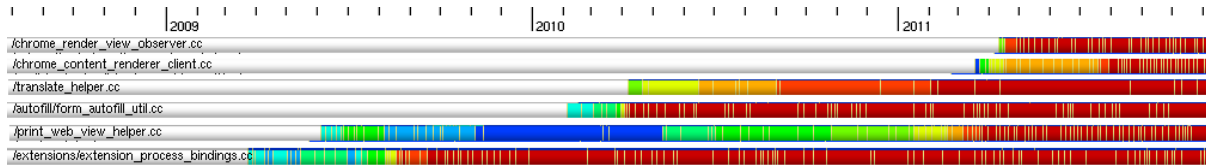


Figure 39: Files with Significant Increase in Complexity

6.3 Complexity and Activity Correlation

In order to investigate the activity, the files in the evolution view have been sorted based on *Activity*. Of course the *McCabe complexity* metric is used. The files that are in the top part of the stack in the evolution view are the most active ones. All of the files above (high complexity and/or complexity rate of variation) are inside the red box which is shown in Figure 40. This means that by being in top third part of the whole stack of files, the activity of these files varies from high to very high.

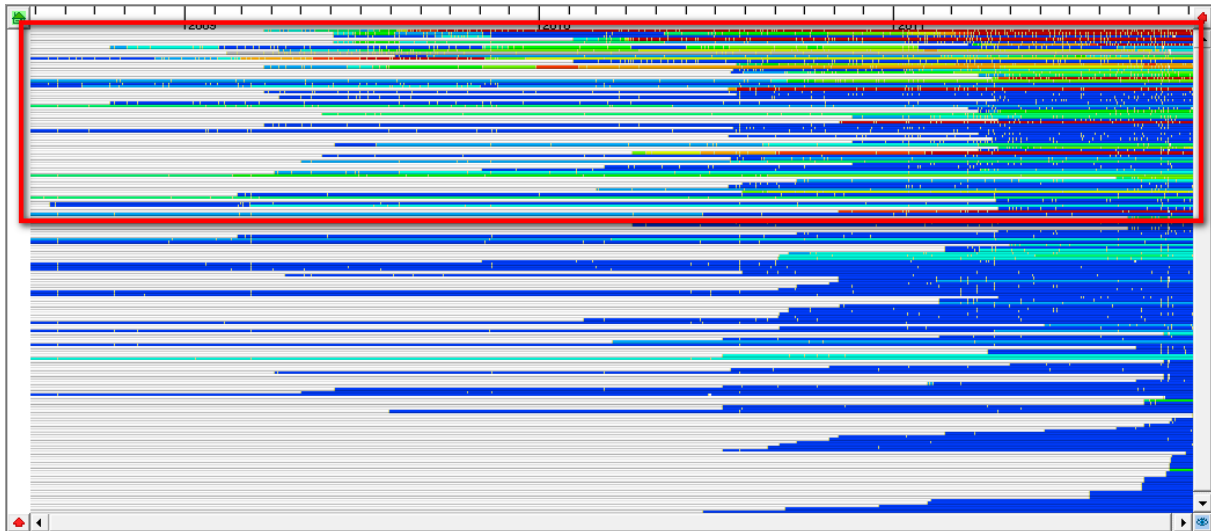


Figure 40: Activity of Files with Complexity Metric

6.4 Complexity and File Size Correlation

In order to find a possible correlation between highly complex files and file size, both the metrics *McCabe complexity* and *Code Size* are used together and separately. The settings of these metrics and their relation with colors, are illustrated in Figure 36 for complexity and in Figure 31 for code size.

In Figure 41, of the metric color composer, it can be seen that the observer has been placed in such a way so its distance from each metric is equal.

The result is that each metric has the same weight in the color of the evolution view. Figure 42 shows the evolution view with the files are colored corresponding to complexity and code size.

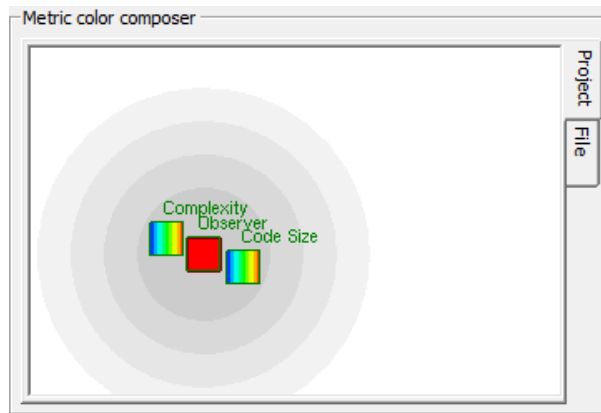


Figure 41: Metric Color Composer - Complexity & Code Size

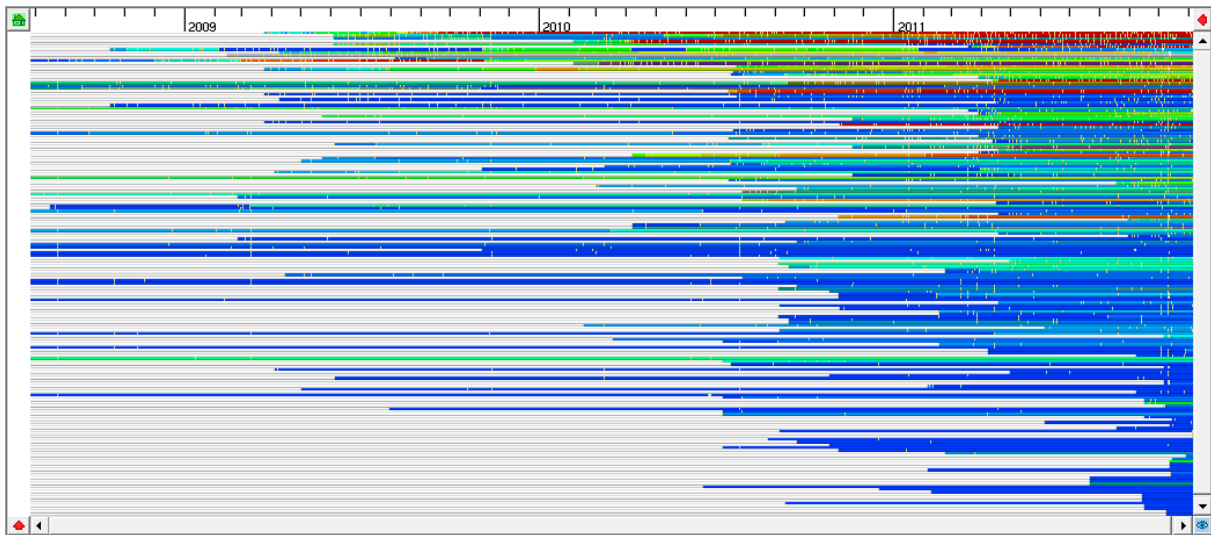


Figure 42: Complexity and Code Size - Evolution View

Figure 43 and Figure 44 show the evolution view with the files colored corresponding to complexity and code size.

As exemplified in these tree Figures, the color of the files remains roughly the same. The same files that have high complexity due to the presence of red color in Figure 43, the same files, in 95% of the occasions, have large size, again due to the presence of red color in Figure 44.

The same applies for the opposite, with low complexity and small sized files. Although this is exactly what is happening in Figure 42 both metrics are applied (red colored files remain red and blue colored files remain blue).

In conclusion there is a correlation between highly complex files and the file size . Highly complex files are often large.

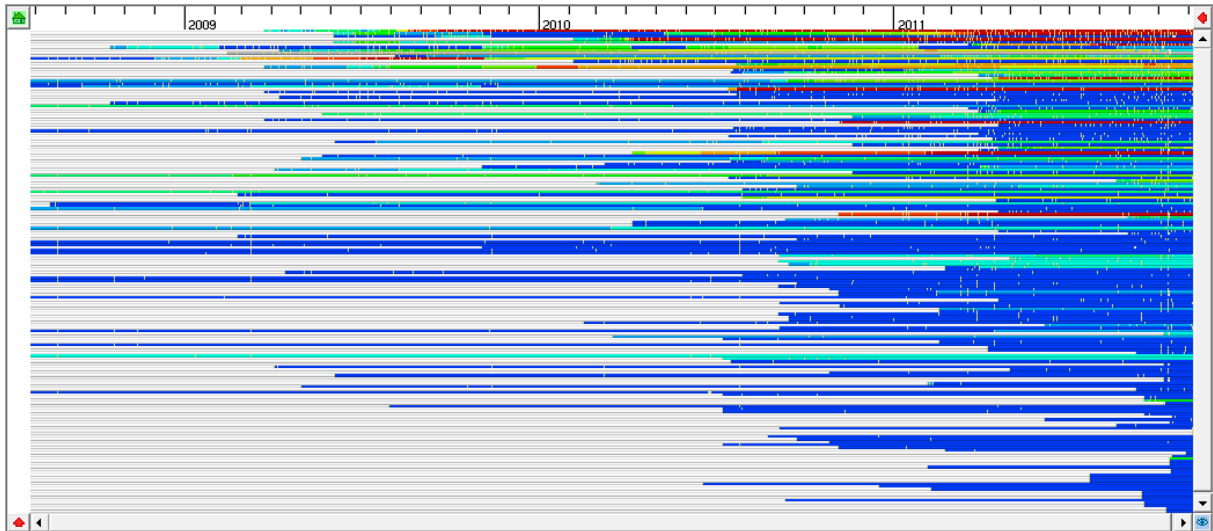


Figure 43: Complexity - Evolution View

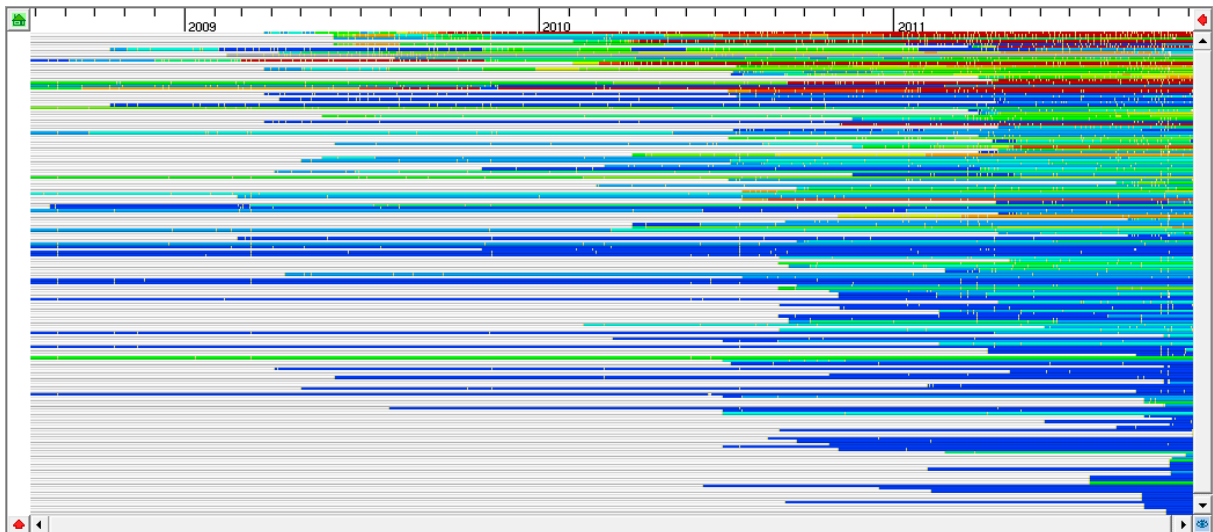


Figure 44: Code Size - Evolution View

In summary, the previous investigation on complexity show that this part of the project has a relatively low complexity, which supports the initial hypothesis.

This part of the project does not require refactoring in order to reduce complexity. Although there are some large files with high complexity, these files are also very active which probably means that they are also very important for the project and it would be relatively difficult to refactor them.

7 Dependency Analysis - Explore a Scenario

7.1 Introduction

The design and development of a software analysis tool involves a lot of problems that needs to be addressed. This includes gathering and processing of data as well as an efficient and meaningful visualization of the data.

Software analysis tools often rely on version management systems like Subversion or Git as a source of information. These repositories are valuable sources of information because they provide the possibility to get previous versions of the source code and thus allow to analyze the evolution of the project over time.

For every single version v_i of a project it is possible to extract a graph G_i . For instance a call graph, build dependency graph or control flow graph. These graphs can be either utilized to analyze the qualities of a particular version or to analyze the evolution of a quality in time. The evolution of a project is especially interesting because it shows the development process, interesting phases and future trends of the qualities. However, the data gained from the analyzation is complex and the visualization is not trivial.

The goal of this essay is to explore the design and development problem space of a tool that is able to visualize the evolution of the previously mentioned graphs $G_1, G_2, G_3 \dots G_n$

7.1.1 Problem

In every case of analyzation and visualization, a critical step needs to be accomplished and this is the definition of a data model. These data models store valuable information extracted basically with code parsing techniques. In general, this information includes sets of graphs G_i as well as the correspondences between their nodes and edges.

Graphs which are being stored in the Data Model described in Section 7.2 can be one of the following types:

- Call graphs

A call graph is a directed cyclic graph that represents calling relationships between sub-routines in pieces of codes. Its nodes represent function definitions and its edges functions calls.

- Containment graphs

A containment graph is a tree, which vertices represent software entities e.g. methods, classes, namespaces, files, folders, packages and its edges represent containment relations.

- Class Inheritance graphs

A class inheritance graph is a directed acyclic graph that basically describes inheritance of classes. Its nodes represent class declarations and its edges represent class inheritance relations.

- Build Dependency graphs

A build dependency graph is a directed acyclic graph and it basically describes compilation dependencies between files in a repository. Its nodes represent files and its edges the compilation dependencies.

After creating a data model and all the extracted graph has been stored. Analyzation and visualization techniques can be applied to it. One possible reason for doing this could be the interest to understand how the detailed structure of source code changes during the evolution of a software project¹⁴.

7.2 Data Model

In order to store the different kind of graphs, a Data Model is proposed that allows to capture all kinds of graphs whether the graph is cyclic, acyclic or a tree. The Data Model is illustrated in Figure 45.

The central entity of the Data Model is the Repository, which holds the address of the Version Control System (e.g. a URL or File path) and the account. In general all the information, which is necessary, to connect to a repository.

The Repository has a list of versions. A version can be seen as a change commit of source code into the repository. Hence, it has a date, a number and an author. The number should be unique and sequential.

Since the Data Model should be capable of storing more than one type of graph, each version can hold a list of arbitrary graphs, e.g. call graph or build dependency graph. The type of graph is determined by the type property of the graph entity. A graph itself consists of edges and vertices.

An adjacency list is used to store the edges between vertices. Each edge refers to exactly two vertices within a graph (from, to). Furthermore, an edge can be directed or undirected and has a type property. A vertex can have a name, a hash and also a type property. Both, vertices and edges have a meta attribute, which can be used to store graph specific properties. For instance, the build impact and build costs, as illustrated in Section 7.5.

In general, this Data Model is able to capture all kinds of graphs, since the meaning of the graph, vertices and edges can be flexibly defined via the edge type. Furthermore, it does not constrain or enforce any properties of a graph, which means that data can be stored very flexibly. Nevertheless, this flexibility is also a tradeoff, since the developer has to implement constraints of the heterogeneous graph types on his own.

¹⁴Structural Analysis and Visualization of C++ Code Evolution using Syntax Trees by F. Chevalier et. al.

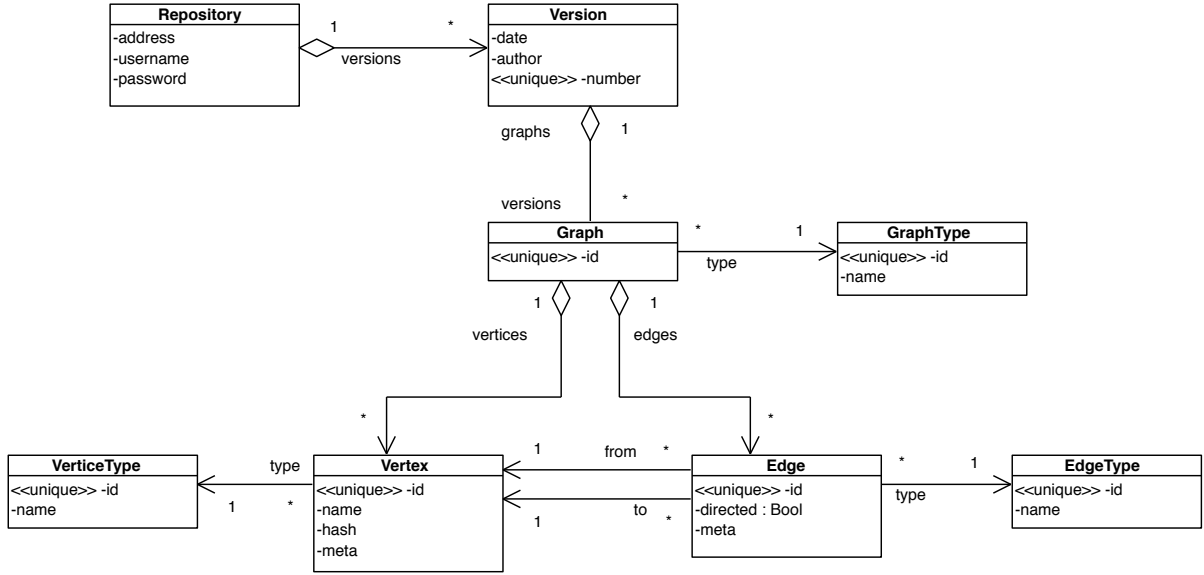


Figure 45: Data Model

7.3 Correspondence

A problem in the visualization of the evolution of a graph is to identify the same vertices and edges across versions ($vertex_i = vertex_{i+1}$). It is necessary to store some kind of correspondence information in the data model.

However, the correspondence information depends on the type of graph which has been extracted. The identity of a vertex in a call graph could be described by its fully qualified name. In case of Java this would be namespace.class.method-name.method-signature. However, this simple scheme is not possible for all types of graphs. A control flow graph for example uses chunks of code as vertices, which cannot be addressed by a name.

Finding correspondent vertices is not a trivial problem. The mentioned solution for call graph does not work, if a method has been renamed or moved. It has to be decided if a method is identified by its name, content or both.

In general a hashing mechanism can be used to calculate a unique identifier for an identity. The hashing mechanism must ensure that the same value will return the same value for entities regardless of their version. However, the concrete implementation depends on the graph type and could range from a simple filepath to a complex clone detection mechanism that allows some tolerance in similarity.

The Data Model requires that the hash value is calculated when the vertex is inserted into the model. It will be stored as an attribute of a vertex. The correspondence information is not stored explicitly in the Data Model but can be calculated during runtime. The pseudo-code for finding correspondent vertices in a graph of type x in two consecutive versions i and j is illustrated in Listing 7

Listing 7: Pseudocode to find correspondent vertices

```

1
2 Graph  gi = repository.getVersion( i ). getGraphByType( x)
3 Graph  gj = repository.getVersion( j ). getGraphByType( x)
4
5 foreach ( Vertex: vi in gi) {
6     vj = gj.findVertexWithHash( v. hash)
7
8     if ( v2 != null)
9         storeCorrespondance( vi , vj)
10 }

```

A tradeoff of the data model is, that the correspondence information needs to be calculated on request. This could cause a performance issue. An alternative would be to explicitly store correspondence information in a separate adjacency list. However, the advantage of the chosen method is, that it allows to skip versions. e.g. correspondence of vertices between v_i and $v_i + 5$. Another trade off is that all graphs of a certain type have to be recalculated if the hashing functionality of a graph changes.

7.4 Updating

Inserting a new version into the Data Model would require to extract the desired graphs and store them in the Data Model. Furthermore, the hashing function has to be executed on each vertex per graph.

7.4.1 Implementation

A software project can be very large with a lot of files and versions. Additionally, the extracted graphs can be even larger, depending on the level of detail.

Because of this, it would be necessary to implement the data model in a performant relational database. An additional advantage of a database is the possibility to define an index on the hash values and thus optimize the findVertexBy hash function.

Furthermore, concurrency is important when updating the model. It would be possible to extract multiple graphs at the same time, since a single graph does not share information with another graph.

In order to optimize the usability of the program, the calculation of the correspondence information can be shifted to the analyzation phase.

7.5 Visualization

After the creation of the data model, visualization techniques are needed in order for effectively analyses to be applied in a project. Throughout this section a technique of visualizing the evolution of a build dependency graph is being presented. The same technique is being used by *SolidTA*. Basically the same *2D layout* as *Solid Trend Analyzer's* main evolution view is being used. It has to be mentioned that the x axis represents time and the y axis represents files, whereby every file version is drawn as a rectangle, as shown in Figures 49 and 50.

7.5.1 Evolution of Build Dependency graphs

Build Dependency graphs are being extracted and stored in the data model. What is needed for the visualization of evolution of those graphs is the computation of the correspondences between files in each version. Actually what is needed is the calculation of the build impact and build cost metric for each file in each version. By having the dependencies as edges and files as nodes of each version in the Data Model, the build impacts and build costs can be computed. The following concrete example illustrates this visualization technique.

The representation of the evolution has been divided into three parts as shown in Figures 46, 47 and 48 . For each file in these three versions of the graph, the building cost (bc) and/or the building impact (bi) has to be calculated with a technique which has been showed during the Software Maintenance and Evolution lectures.

For the calculation of bc , the number of $.h$ files that need to be recompiled during a rebuild is taken into consideration. For example in Figure 46 file $e.so$ has $bc = 2$, which is the number of files that needs to be recompiled in case of a rebuild, $c.h$ and $a.h$.

For the calculation of bi , the bc of files that are being affected due to this file's compilation in case of a rebuild are taken into consideration. For example in Figure 47 file $c.h$ has $bi = 5$ because the addition of all the bc of files that will be affected of a possible rebuild is 5, files $d.h$ and $e.h$.

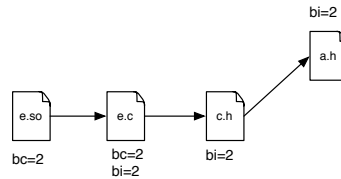


Figure 46: Build Graph (1)

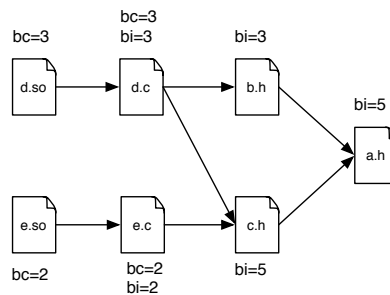


Figure 47: Build Graph (2)

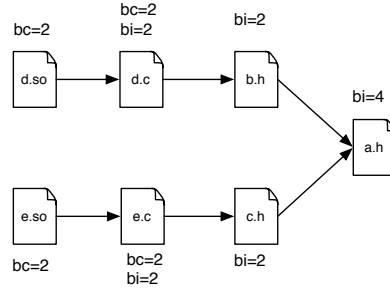


Figure 48: Build Graph (3)

The three graphs in Figure 46,47,48 show the build dependency graph in three different moments of time annotated with bi and bc. These moments in time are reflected in Figure 49 and 50 with black arrows.

• Building Impact

Figure 49 shows that throughout the period before the first arrow, the building impact for files *a.h* and *c.h* is zero. When file *e.c* is added the building impact of files *a.h* and *c.h* increases to 2 as also depicted in Figure 46.

Considerable increases take place in time slice 2 when the file *d.c* is added in the project after the addition of file *b.h*. The building impact of all files increases except from file *e.c*, as also depicted in Figure 47. Especially the building impact of the first 2 added files is increased to 5.

In time slice 3 a deletion of the dependency between files *d.c* and *c.h* is illustrated, as also in Figure 48. It is obvious that building impacts of all files are decreased.

• Building Cost

Visualization technique works approximately the same for showing the evolution of building costs of files. We can also see that 3 time slices are illustrated when building costs are either increasing or decreasing depending on the specific occasion.

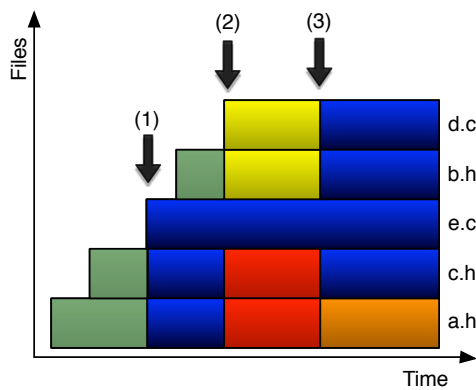


Figure 49: Evolution of Build Impacts

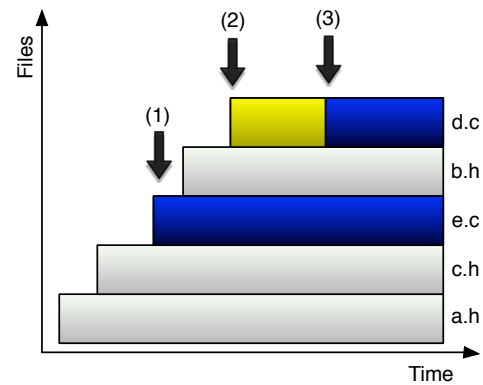


Figure 50: Evolution of Build Costs



Figure 51: Colors - Building Impact, Building Cost

The proposed visualization technique produces no cluttering as the color pattern that is used offers clear results and clear visualization. Even for large numbers of files of a complex and large dataset this technique would offer an efficient degree of visualization.

Appendix

A1 Release dates

Table 1: Release dates

Version	Date
1	Dec 1, 2008
2	Jan, 2009
3	May 28, 2009
4	Sep 22, 2009
5	Jan 26, 2010
6	May, 2010
7	Aug 17, 2010
8	Oct 7, 2010
9	Oct 23, 2010
10	Dec 3, 2010
11	Jan 28, 2011
12	Mar 11, 2011
13	Apr 26, 2011
14	Jun 2, 2011
15	Jul 28, 2011
16	Sep 10, 2011
17	Oct 19, 2011

A2 Command-line output

b) Results Revision Analysis

Listing 8: Information about the latest revision

```
1 Path: svn
2 URL: http:// src. chromium. org/ svn
3 Repository Root: http:// src. chromium. org/ svn
4 Repository UUID: 4 ff67af0 -8 c30 -449 e-8 e8b- ad334ec8d88c
5 Revision: 108459
6 Node Kind: directory
7 Last Changed Author: rogerta@chromium. org
8 Last Changed Rev: 108459
9 Last Changed Date: 2011-11-03 15:09:56 +0100 ( Thu, 03 Nov 2011)
```

Listing 9: Information about the first revision

```

1 Path:  svn
2 URL:  http:// src. chromium. org/ svn
3 Repository Root:  http:// src. chromium. org/ svn
4 Repository  UUID:  4 ff67af0 -8 c30 -449 e-8 e8b- ad334ec8d88c
5 Revision:  1
6 Node Kind:  directory
7 Last  Changed Author:  initial. commit
8 Last  Changed Rev:  1
9 Last  Changed Date:  2008-07-25 23:13:22 +0200 ( Fri , 25  Jul 2008)

```

A3 Time-tracking

Task	hh:mm
Investigating release dates and creating chart	2:00
Getting revision information for whole repository	0:15
Basic Investigation of chrome/renderer	4:10
Basic Investigation of chrome	8:00
SolidTA Content Analysis	7:00
StatSVN Analysis	5:25
Checkout Chromium Repository	3:50
Getting a first visual overview	4:30
Code Size Analysis	8:00
Complexity Analysis	10:00
Reading Articles	4:00
Authors Analysis	6:45
Familiarizing with SolidTA	5:00
Writing Essay	12:00
Review	6:40
Writing Introduction	2:00