



**rijksuniversiteit
groningen**

faculteit wiskunde en
natuurwetenschappen

informatica

Software Maintenance & Evolution

Evolution analysis of The GIMP

Avdo Hanjalic / s1553623

Erik Bakker / s2074893

November 24, 2011

Contents

1	Introduction	3
2	Basic repository investigation	4
2.1	Revision history	4
2.1.1	Revision count	4
2.1.2	First commit	5
2.1.3	Last commit	5
2.2	Source-tree analysis	6
2.2.1	Largest folders	6
2.2.2	Folder containing most source-code	6
2.2.3	Source directory (/app) statistics	9
2.3	Most active developers	10
2.3.1	Top 3 active developers	10
3	Getting a first visual overview	13
3.1	Stable development periods	13
3.2	Intense changes	13
3.3	Currently stable	14
4	Authors analysis	15
4.1	Main contributors	15
4.2	Main developers quit	16
4.3	Correlation between type of files and developers	16
4.4	Correlation between location of files and developers	18
5	Code size analysis	19
5.1	Code size evolution	21
5.2	Code size vs Total Size	24
6	Complexity analysis	25
6.1	Most complex files	25
6.2	Complexity fluctuations	26
6.3	Complexity correlations	28
7	Conclusion	31
8	Evaluation	32
8.1	Discarded repositories	32
8.1.1	Chromium	32
8.1.2	FileZilla 3 client	33
8.1.3	Octave	33
8.1.4	TortoiseSVN	34
8.1.5	XBMC	34
8.1.6	VirtualBox	34
8.2	SolidTA	34
8.2.1	Performance	34
8.2.2	Stability & reliability	35
8.3	Software evolution analysis	35
9	Appendices	36
9.1	Time consumption	36

1 Introduction

For the course Software Maintenance and Evolution we were requested to write a report on the evolution analysis of some (open source) 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."

Some software projects were proposed, however these were not ideal for analysis, as they did not satisfy all relevance and/or workability requirements. It took us quite some time to find a workable project; we discarded 6 projects before finding a workable one. For a repository to be relevant for analysis it had to satisfy these properties:

- It contains more than a few thousands of files
- It contains more than a few thousands of revisions
- It has to be written in C or C++
- It was developed (and committed) by many users.

For a repository to be workable with our tools, the complete history of (a subset of) the (SVN) repository must be retrievable within an acceptable amount of time. Therefore the following must hold:

- The repository server is not overloaded
- The amount of revisions is less than 30,000 (approximately)
- The amount of files (of the subset) is less than 15,000 (approximately)

We limited the amount of files and revisions, because the amount of data to be retrieved is of order files x revisions.

It took us quite some time to find a repository meeting all these properties: after discarding 6 projects we eventually found The GIMP, which appeared to have a workable repository. The problems of finding a workable repository are discussed in the evaluation section.

2 Basic repository investigation

The first thing to do was to check out the trunk directory of GIMP using TortoiseSVN. The checkout took about 30 minutes to complete, resulting in a code-base of approximately 100MB. In this report we will review the project from its initial commit up to the last SVN commit on 16.04.2009 (r28296). The GIMP moved to a Git before that, but the last 'back-sync' took place then.

2.1 Revision history

2.1.1 Revision count

"How many versions are in the repository?"

To find out the amount of 'versions' (commits), we go to the directory where the repository was checked out with SVN, e.g. D:\Dev\gimptrunk. On this directory we call the (Tortoise) SVN log. The repository contains 28296 commits until 16.04.2009, however the amount of commits for this particular directory (/trunk) are 25041, as shown in figure 2.1.1. Once the repository was checked out, the retrieval of this information took an insignificant amount of time, compared with the amount of time needed for reporting.

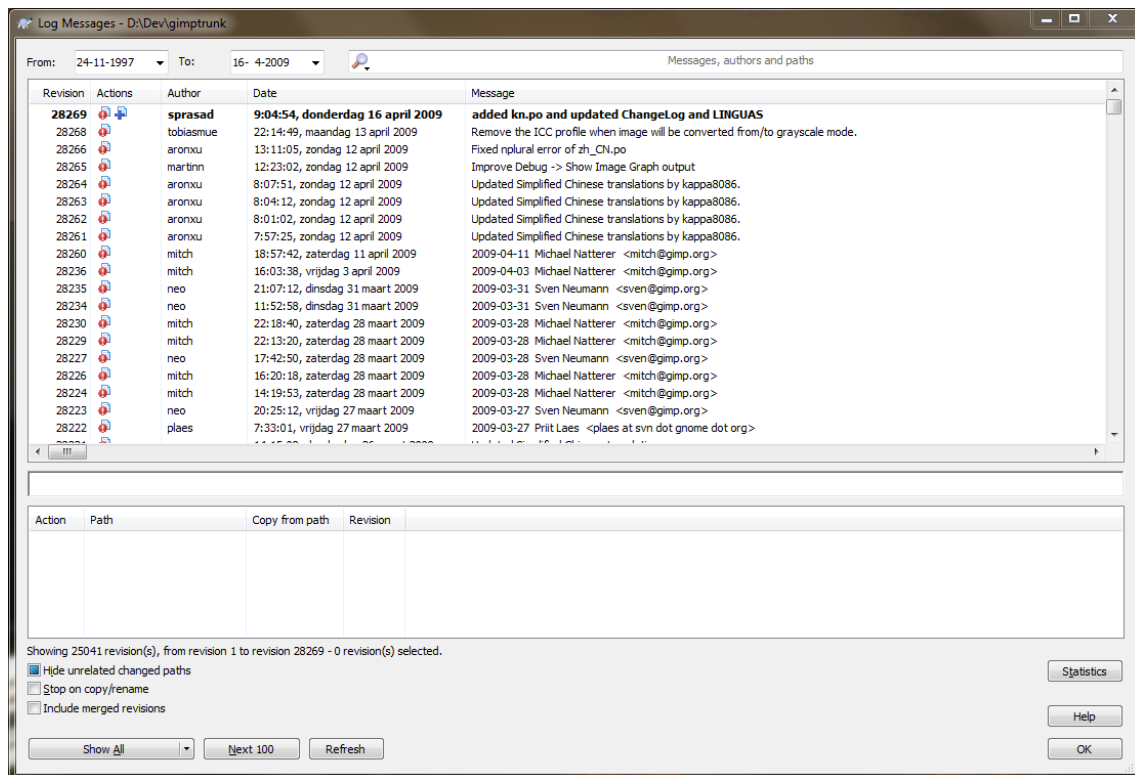


Figure 2.1.1: Last commit

2.1.2 First commit

"When was the first one committed?"

To find out when the first commit was, we scroll down to the beginning of the log by selecting a random revision record and pressing the END button. Here we see that the initial commit took place Monday November 24th 1997 at 23:05:25. Once the repository was checked out, the retrieval of this information took an insignificant amount of time, compared with the amount of time needed for reporting.

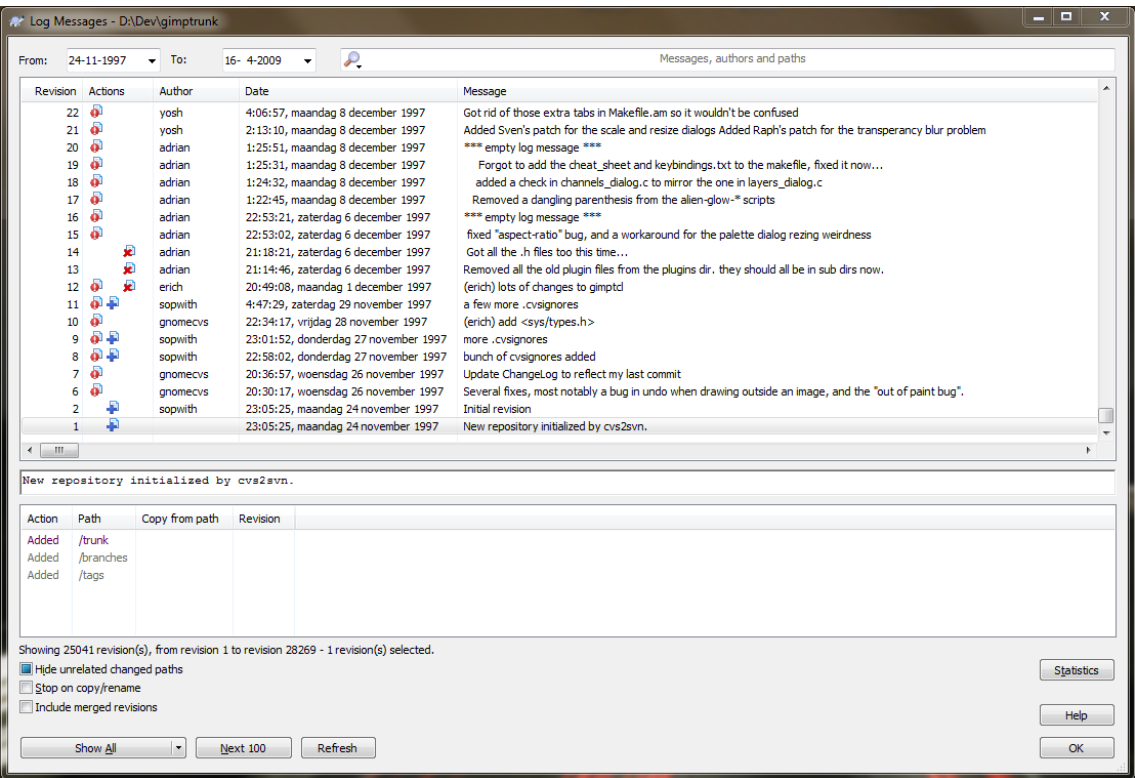


Figure 2.1.2: First commit

2.1.3 Last commit

"When was the last one committed?"

This information can be derived from figure 2.1.1: the last visible commit was Thursday April 16th 2009 at 09:04:54. This information is retrieved together with the amount of versions, therefore taking insignificant amount of time.

2.2 Source-tree analysis

2.2.1 Largest folders

"Which are the top-level largest folders in the repository (i.e. containing the most files)?"

To get an answer to this question, we have decided to first do a SVN export of the repository check-out and run a TreeSize¹ tool on it. The reason for the export was to prevent the TreeSize tool counting the files (Tortoise)SVN creates for its administration. After 1 minute to export the repository and about 5 minutes of getting familiar with the tool, we managed to plot the directory sizes, as shown below:

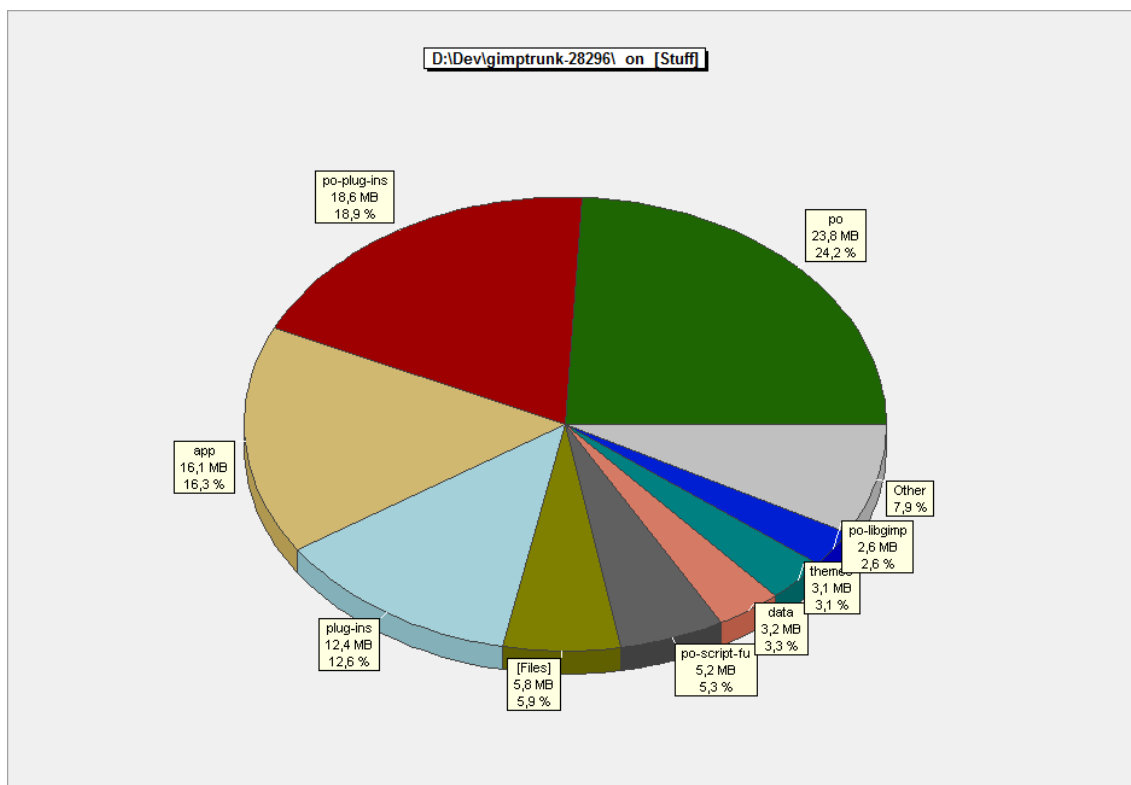


Figure 2.2.1: Contents of /

To get these results we started TreeSize Personal, chose the directory of the SVN export, selected the directory and finally disabled the showing of free (disk) space.

From figure 2.2.1 we can conclude that the top 4 largest directories are:

1. /po (24.2% of the whole source)
2. /po-plugin-ins (18.9% of whole source)
3. /app (16.3% of whole source)
4. /plug-ins (12.6% of whole source)

2.2.2 Folder containing most source-code

"Which, of these folders, are the ones where the most source code is located?"

Finding the answer to this question required some more time than for the previous one as we were not familiar with the tool, which appeared the tool was able to filter on file types. In this case we are interested in 'Software Development Files'.

¹<http://www.jam-software.com/treesize/>

Finally we check the contents of the 4 sub-directories separately, to see the amount of 'Software Development Files'. The results are shown below:

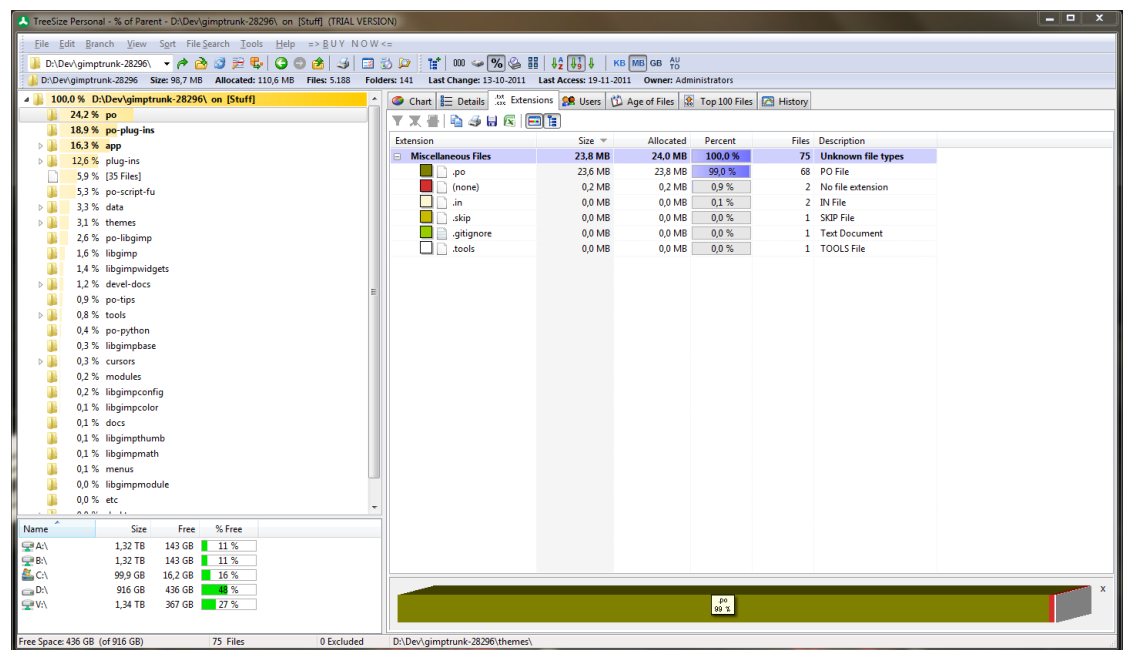


Figure 2.2.2.1: /po

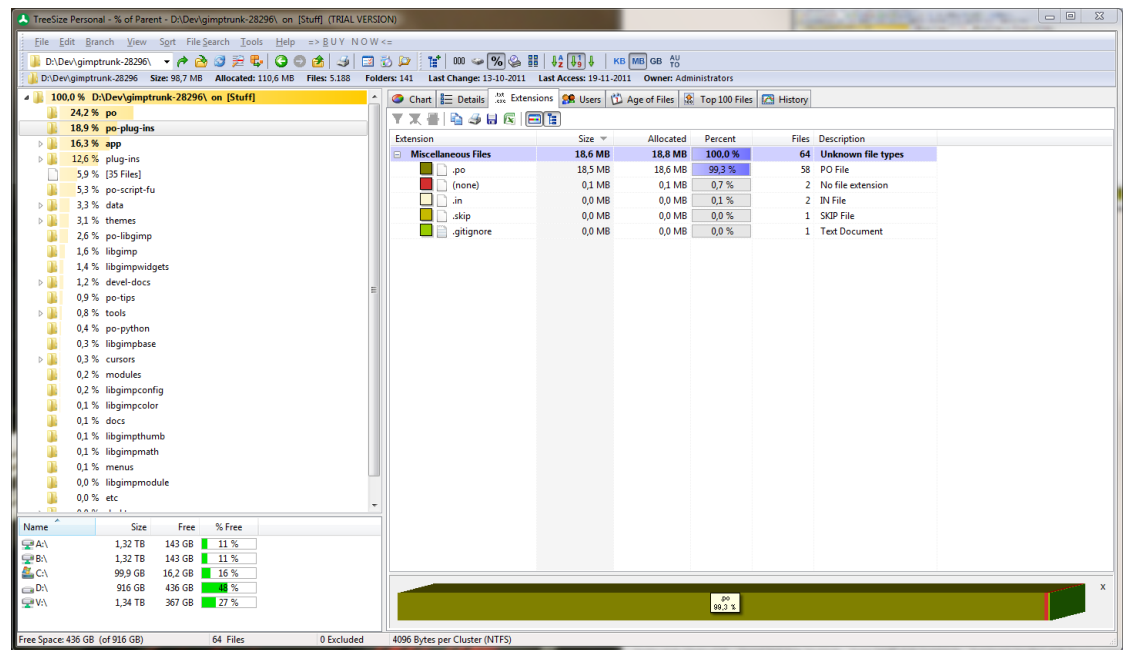


Figure 2.2.2.2: /po-plugin-ins

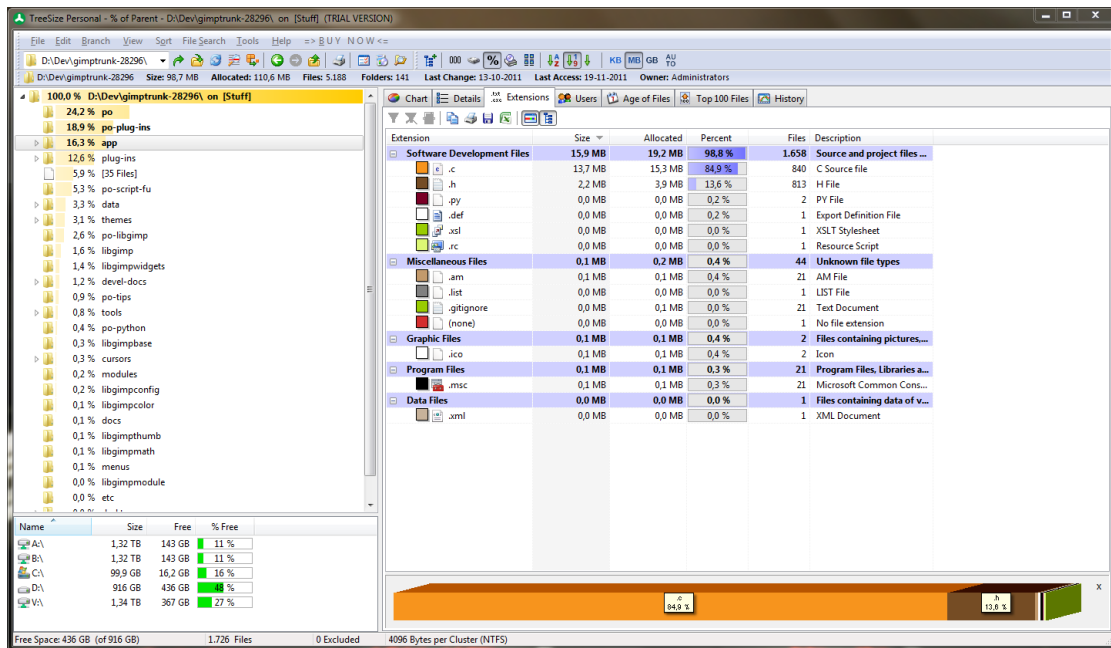


Figure 2.2.2.3: /app

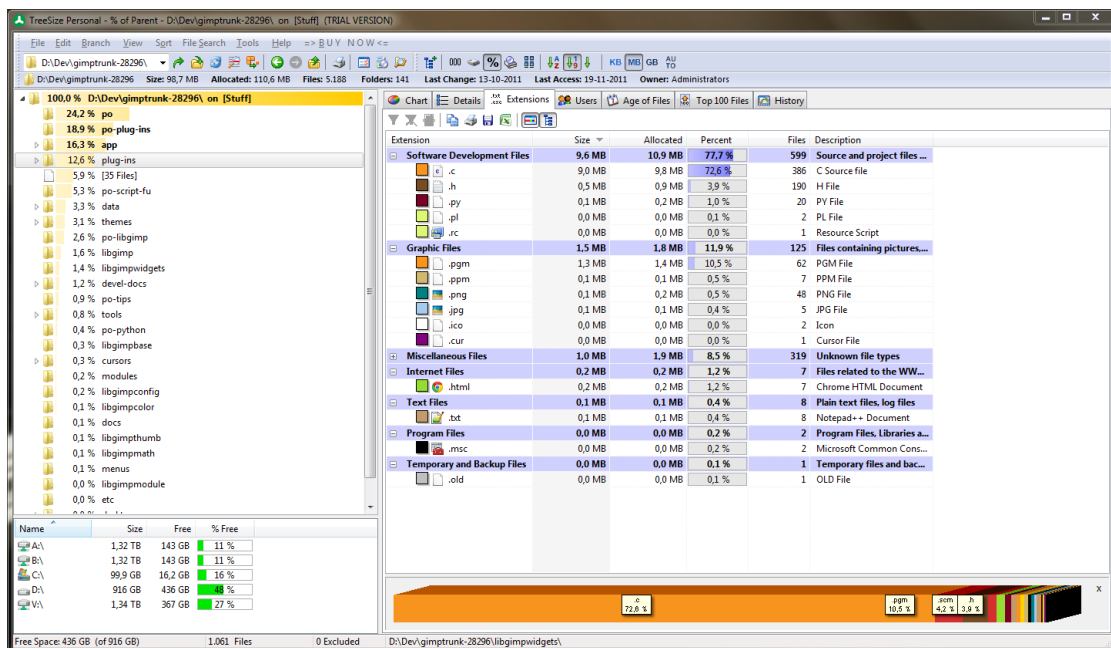


Figure 2.2.2.4: /plug-ins

Files placed in the po* directories are translation files:

"The PO file type is primarily associated with 'GNU Gettext' by Free Software Foundation. The GNU gettext utilities are a set of tools that provides a framework to help other GNU packages produce multi-lingual messages. PO files are meant to be read and edited by humans, and associate each original, translatable string of a given package with its translation in a particular target language."²

Depending on the interpretation of 'source files' one could interpret the directories '/po' or '/app' to contain most source files. Even though translation files certainly are part of the source-code, we prefer to interpret the C-files, containing program logic, as actual source files.

²<http://filext.com/file-extension/PO>

2.2.3 Source directory (/app) statistics

As the previous analysis did not generate exciting results we have decided to look deeper into the /app directory. An overview of this folder is shown below:

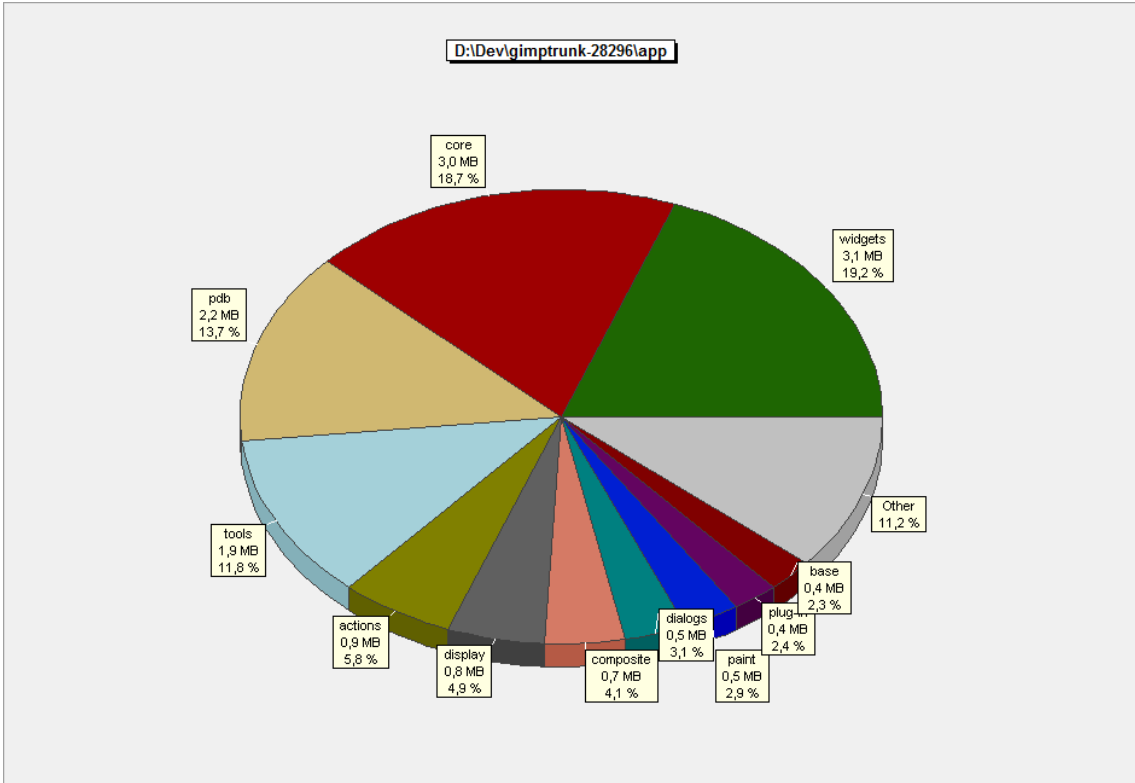


Figure 2.2.3.1: /app contents

The folders /app/core and /app/widgets appear to contain the most lines of code (expressed in disk space utilization).

2.3 Most active developers

To get the result the following steps are taken:

- Download the latest version of StatSVN from <http://www.statsvn.org/>
- Unzip the downloaded zip file
- Open a terminal
- Change into the svn directory and type 'svn log --xml -v > svn.log'
- Change back to the <statsvn folder>
- Type: 'java -jar statsvn.jar <repository dir>/svn.log <repository dir>'
- Wait for a couple of hours and open <statsvn folder>/index.html

Author	Author Id	Changes	Lines of Code	Lines per Change
neo	neo	58268 (29.4%)	3566987 (27.5%)	61.2
mitch	mitch	74328 (37.6%)	1814025 (14.0%)	24.4
yosh	yosh	15775 (8.0%)	1275984 (9.8%)	80.8
sopwith	sopwith	2522 (1.3%)	607420 (4.7%)	240.8
kmaraas	kmaraas	315 (0.2%)	180433 (1.4%)	572.8
adamw	adamw	519 (0.3%)	147980 (1.1%)	285.1
marcoc	marcoc	834 (0.4%)	145023 (1.1%)	173.8
danilo	danilo	311 (0.2%)	139042 (1.1%)	447.0
Helvetix	Helvetix	538 (0.3%)	116349 (0.9%)	216.2
pcq	pcq	3736 (1.9%)	100790 (0.8%)	26.9
dnloreto	dnloreto	192 (0.1%)	88168 (0.7%)	459.2
utx	utx	576 (0.3%)	82869 (0.6%)	143.8
simon	simon	1847 (0.9%)	80101 (0.6%)	43.3
mitr	mitr	304 (0.2%)	78226 (0.6%)	257.3
pablo	pablo	145 (0.1%)	74744 (0.6%)	515.4
plaes	plaes	109 (0.1%)	72674 (0.6%)	666.7
adriqhem	adriqhem	125 (0.1%)	71598 (0.6%)	572.7
dooteo	dooteo	91 (0.0%)	70305 (0.5%)	772.5
menthos	menthos	231 (0.1%)	68734 (0.5%)	297.5
dijhed	dijhed	106 (0.1%)	66861 (0.5%)	630.7
redfox	redfox	185 (0.1%)	65752 (0.5%)	355.4
qrakic	qrakic	68 (0.0%)	65282 (0.5%)	960.0
baddog	baddog	148 (0.1%)	65272 (0.5%)	441.0
minmax	minmax	41 (0.0%)	64809 (0.5%)	1580.7
stano	stano	128 (0.1%)	64668 (0.5%)	505.2
simos	simos	35 (0.0%)	64153 (0.5%)	1832.9
dindinx	dindinx	1590 (0.8%)	61105 (0.5%)	38.4
olau	olau	208 (0.1%)	60086 (0.5%)	288.8
alt	alt	1208 (0.6%)	59064 (0.5%)	48.8
jimmac	jimmac	1014 (0.5%)	58910 (0.5%)	58.0
aihana	aihana	149 (0.1%)	58635 (0.5%)	393.5
zyqis	zyqis	160 (0.1%)	57536 (0.4%)	359.6
jordim	jordim	46 (0.0%)	56152 (0.4%)	1220.6
al_shopov	al_shopov	80 (0.0%)	56130 (0.4%)	701.6
siquird	siquird	53 (0.0%)	54512 (0.4%)	1028.5

Figure 2.3.1: Contribution of developers

For finding the most active developers StatSVN is used. This tool is not yet in version 1.0, but works very good. The output took about 6 hours to generate. The following users are the most active users during the whole project. It is sorted on the most lines of code committed to the project. The table header changes are the amount of commits done by a particular user. This table is based on the whole project.

Neo and mitch are the users who contributed the most for the project. Together they are responsible for 67% of the commits of the project. That is 41.5% lines of code of the whole project.

2.3.1 Top 3 active developers

"Which are the three most active developers in the first half of the project?"

The three most active developers in the first half of the project, that is from 1997-11-24 to 2003-08-04, are neo, yosh and mitch. To get this information the following steps were taken.

- Download the latest version of StatSVN from <http://www.statsvn.org/>
- Unzip the downloaded zip file
- Open a terminal

- Change into the svn directory and type 'svn log -xml -v -r {1997-11-24}:{2003-08-04} > svn.log'
- Change back to the <statsvn folder>
- Type: 'java -jar statsvn.jar <repository dir>/svn.log <repository dir>'
- Wait for a couple of hours and open <statsvn folder>/index.html

Author	Author Id	Changes	Lines of Code	Lines per Change
neo	neo	24190 (21.7%)	1782866 (26.8%)	73.7
yosh	yosh	14181 (12.7%)	1321537 (19.9%)	93.1
mitch	mitch	43362 (39.0%)	1297726 (19.5%)	29.9
kmaraas	kmaraas	279 (0.3%)	159069 (2.4%)	570.1
pablo	pablo	131 (0.1%)	128758 (1.9%)	982.8
pcq	pcq	3885 (3.5%)	119854 (1.8%)	30.8
utx	utx	574 (0.5%)	82639 (1.2%)	143.9
menthos	menthos	227 (0.2%)	74965 (1.1%)	330.2
alt	alt	1210 (1.1%)	72864 (1.1%)	60.2
sopwith	sopwith	2523 (2.3%)	59785 (0.9%)	23.6
egger	egger	559 (0.5%)	53912 (0.8%)	96.4
baddog	baddog	100 (0.1%)	49459 (0.7%)	494.5
adrian	adrian	1578 (1.4%)	49205 (0.7%)	31.1
dnloreto	dnloreto	91 (0.1%)	45114 (0.7%)	495.7
lark	lark	78 (0.1%)	43610 (0.7%)	559.1
tml	tml	2283 (2.1%)	43048 (0.6%)	18.8
rockwalru	rockwalru	762 (0.7%)	41809 (0.6%)	54.8
chyla	chyla	112 (0.1%)	41687 (0.6%)	372.2
adam	adam	1146 (1.0%)	40522 (0.6%)	35.3
frob	frob	197 (0.2%)	38063 (0.6%)	193.2
kabalak	kabalak	47 (0.0%)	36533 (0.5%)	777.2
nether	nether	627 (0.6%)	34997 (0.5%)	55.8
javcox	javcox	1343 (1.2%)	34961 (0.5%)	26.0
rasta	rasta	188 (0.2%)	34285 (0.5%)	182.3
olau	olau	133 (0.1%)	33238 (0.5%)	249.9
vasuhiro	vasuhiro	354 (0.3%)	33128 (0.5%)	93.5
docsbr	docsbr	63 (0.1%)	32144 (0.5%)	510.2
minmax	minmax	36 (0.0%)	31391 (0.5%)	871.9
kenneth	kenneth	70 (0.1%)	28790 (0.4%)	411.2
jamesh	jamesh	141 (0.1%)	28755 (0.4%)	203.9
simos	simos	18 (0.0%)	27144 (0.4%)	1508.0
redfox	redfox	109 (0.1%)	26771 (0.4%)	245.6
austin	austin	801 (0.7%)	26607 (0.4%)	33.2
pablodc	pablodc	131 (0.1%)	25799 (0.4%)	196.9
stric	stric	300 (0.3%)	25409 (0.4%)	84.6

Figure 2.3.2: Contribution of developers first half project

As displayed in the image. The user mitch committed more often(changes), but neo and yosh committed more lines of code

"Which are the three most active developers for the last half of the project?"

For the second part of the project do the same steps as described in chapter 2.3.1, but change the line 'svn log -xml -v -r {1997-11-24}:{2003-08-04} > svn.log' to 'svn log -xml -v -r {2003-08-04}:{2009-04-16} > svn.log'. The following image will be created.

Author	Author Id	Changes	Lines of Code	Lines per Change
neo	neo	35705 (37.7%)	2450860 (35.3%)	68.6
mitch	mitch	35142 (37.1%)	740208 (10.7%)	21.0
adamw	adamw	525 (0.6%)	151790 (2.2%)	289.1
marcoc	marcoc	847 (0.9%)	150770 (2.2%)	178.0
danilo	danilo	297 (0.3%)	139584 (2.0%)	469.9
Helvetix	Helvetix	445 (0.5%)	112126 (1.6%)	251.9
plaes	plaes	114 (0.1%)	72674 (1.0%)	637.4
yosh	yosh	2077 (2.2%)	71602 (1.0%)	34.4
simon	simon	1519 (1.6%)	70009 (1.0%)	46.0
dooteo	dooteo	97 (0.1%)	68347 (1.0%)	704.6
dijhed	dijhed	112 (0.1%)	66861 (1.0%)	596.9
dindinx	dindinx	1411 (1.5%)	66654 (1.0%)	47.2
al_shopov	al_shopov	85 (0.1%)	64264 (0.9%)	756.0
grakic	grakic	84 (0.1%)	63963 (0.9%)	761.4
mitr	mitr	221 (0.2%)	63315 (0.9%)	286.4
aihana	aihana	145 (0.2%)	62993 (0.9%)	434.4
zyqis	zyqis	166 (0.2%)	58781 (0.8%)	354.1
jordim	jordim	46 (0.0%)	56152 (0.8%)	1220.6
adriqhem	adriqhem	112 (0.1%)	55610 (0.8%)	496.5
siqurd	siqurd	53 (0.1%)	54296 (0.8%)	1024.4
weskagqs	weskagqs	2054 (2.2%)	54269 (0.8%)	26.4
stano	stano	89 (0.1%)	52278 (0.8%)	587.3
pachimho	pachimho	51 (0.1%)	51420 (0.7%)	1008.2
dnylande	dnylande	172 (0.2%)	49578 (0.7%)	288.2
aman	aman	45 (0.0%)	47827 (0.7%)	1062.8
serrador	serrador	155 (0.2%)	46760 (0.7%)	301.6
hebra	hebra	168 (0.2%)	45281 (0.7%)	269.5
ifriedl	ifriedl	445 (0.5%)	44736 (0.6%)	100.5
nshmyrev	nshmyrev	101 (0.1%)	44202 (0.6%)	437.6
ituohela	ituohela	106 (0.1%)	43954 (0.6%)	414.6
redfox	redfox	85 (0.1%)	42107 (0.6%)	495.3
ica	ica	61 (0.1%)	41822 (0.6%)	685.6
dnlloreto	dnlloreto	109 (0.1%)	41062 (0.6%)	376.7
clyties	clyties	82 (0.1%)	40461 (0.6%)	493.4
pgevleg	pgevleg	42 (0.0%)	40285 (0.6%)	959.1

Figure 2.3.2: Contribution of the developers second half project

The amount of commits are almost the same for the users neo and mitch, but mitch committed more then 20% more lines of code. These two users are committing a lot in respect to the other users in the second part of the project.

3 Getting a first visual overview

In this chapter the program SolidTA will be used to get a visual overview of the subversion repository.

3.1 Stable development periods

Stable development periods can be found by looking for periods within which the code base does not change and/or grow significantly. The files in SolidTA can be sorted by creation time. The yellow dots are commits(changes in the code). The following image shows this.

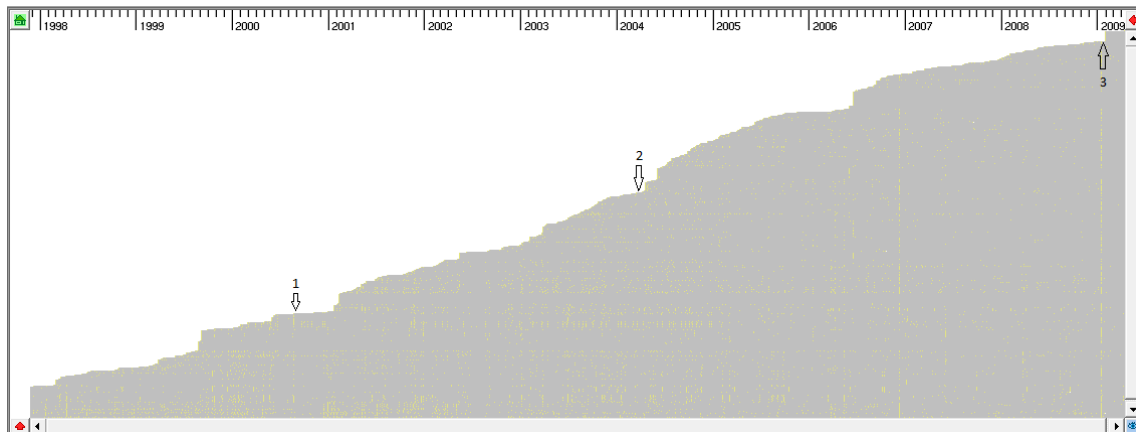


Figure 3.1.1: Stable development periods

Stable releases can be found by looking at time periods where there are no or a few yellow dots (changes made in the code). A stable release could be halfway 2000 (arrow 1), because in that period there are a few dots and the amount of files did not change. At the end of 2000 many new files were created and many commits took place. Maybe new features were installed or the code was refactored.

Before arrow 2 heavy development took place. At and after arrow 2 there is a period the development almost stopped and continued when a few files were created. At arrow 3 the license was changed according to the SVN log. After this change there is very little development. A few dots appear. This could be the latest stable release point.

3.2 Intense changes

The image in chapter 3.1 gives us also the periods when there were many changes committed. At the beginning of 2000 there were intense changes taking place. There are lots of yellow dots which indicates commits. Half way 2000 a stable release of the project was released. In that period the development almost stopped for a month.

At 11 o'clock in the evening the users of the project committed the most. The following image displays the amount of commits during the hours of the day. It could be that the users started heavy developing at around 20:00 and committed the changes at 23:00.

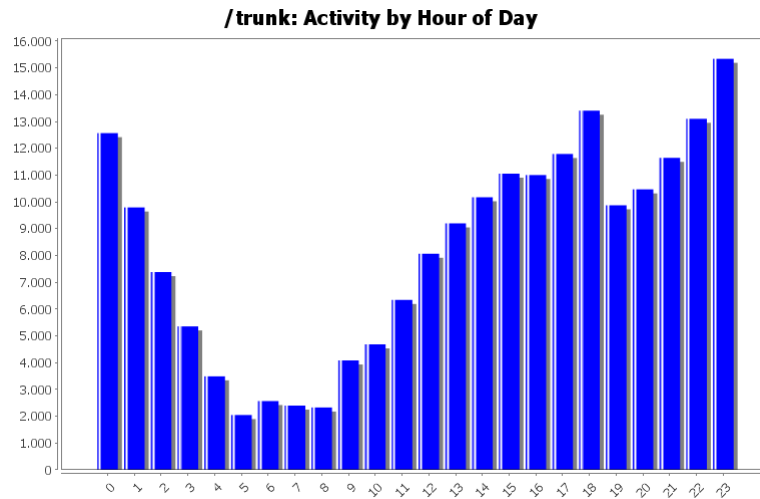


Figure 3.2.1: Activity during the hours of the day

The following image shows the amount of commits during the days of the week.

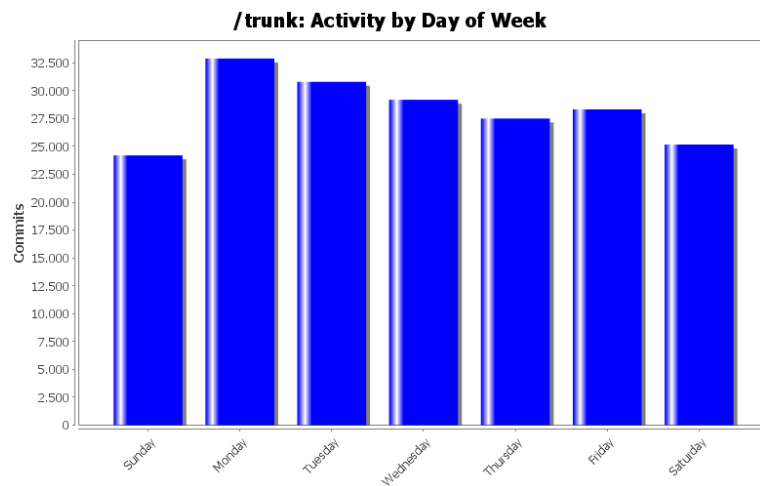


Figure 3.2.2: Activity during the days of the week

As the image shows the most activity is on Monday and then decreases a little bit by day, except Friday.

3.3 Currently stable

The image in chapter 3.1 shows us that in the beginning of 2009 a stable release could be released. From January 2009 there are a few commits and the growth rate of the files is stable. It does not grow anymore. This continues for about 3 months. So there could be a stable release released. This is also because the project moved to a GIT revision control system. The vertical yellow dots line near arrow 3 indicates that almost every file was changed. According to the log the following was changed in every file:

- * all files with a GPL header and all COPYING files:
Change license to GPLv3 (and to LGPLv3 for libgimp).
Cleaned up some copyright headers and regenerated the parsers in the ImageMap

4 Authors analysis

The aim of this chapter is to perform several analyses regarding the authors of GIMP, i.e. persons who commit changes in the repository.

4.1 Main contributors

The steps taken to retrieve the following image were:

- In the Metrics tab check the authors box.
- Select all authors.
- Open the evolution trend view.
- The display type of the evolution trend view set to flow.
- Set Commit count.

The following image will appear.

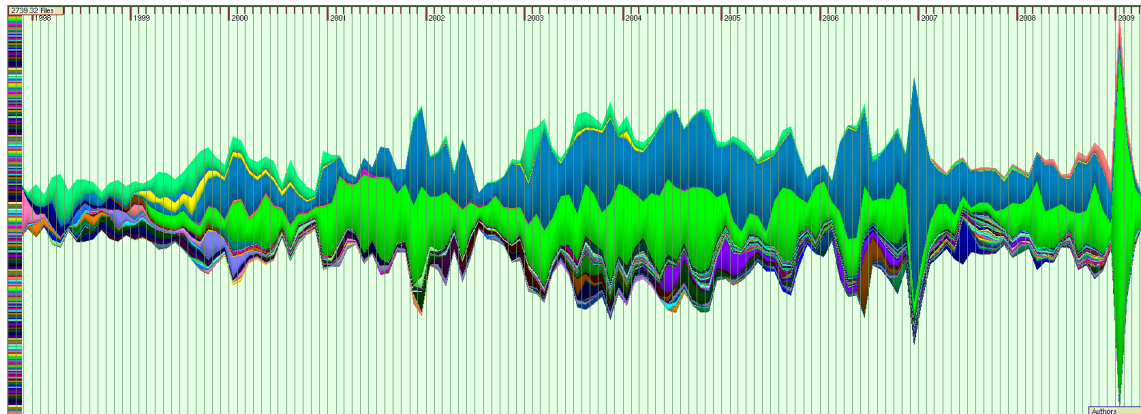


Figure 4.1.1: Contribution of developers

In the x-as the time is shown. The y-as represent the user(a color). The greater the amplitude the more changes the user committed in that period of time.

As the above image shows there are two users who committed a lot during the project. These users are mitch(green) and neo(blue). What is not visible in the image is in which file/folder the users mitch and neo are committing. The following images show where the users mitch and neo were developing:

Activity in Directories			
Directory	Changes	Lines of Code	Lines per Change
plug-ins/common/	3471 (4.7%)	184285 (10.2%)	53.0
app/widgets/	7360 (9.9%)	171452 (9.5%)	23.2
app/core/	7436 (10.0%)	148359 (8.2%)	19.9
/	5195 (7.0%)	144681 (8.0%)	27.8
app/	6031 (8.1%)	123483 (6.8%)	20.4
po/	315 (0.4%)	116894 (6.4%)	371.0
app/tools/	7431 (10.0%)	111073 (6.1%)	14.9
app/ui/	4732 (6.4%)	101341 (5.6%)	21.4
app/pdb/	3395 (4.6%)	93139 (5.1%)	27.4
libgimp/	1984 (2.7%)	48987 (2.7%)	24.6
po-plug-ins/	122 (0.2%)	48840 (2.7%)	400.3
app/actions/	2678 (3.6%)	46908 (2.6%)	17.5
plug-ins/print/	138 (0.2%)	39375 (2.2%)	285.3
app/display/	2111 (2.8%)	37345 (2.1%)	17.6
app/plug-in/	2347 (3.2%)	35068 (1.9%)	14.9
app/dialogs/	1660 (2.2%)	33689 (1.9%)	20.2
libgimpwidgets/	1183 (1.6%)	29089 (1.6%)	24.5
app/print/	1482 (2.0%)	28965 (1.6%)	19.5
po-script-fu/	76 (0.1%)	25081 (1.4%)	330.0
app/eval/	479 (0.6%)	12665 (0.7%)	26.4
app/menus/	595 (0.8%)	12209 (0.7%)	20.5
modules/	220 (0.3%)	11287 (0.6%)	51.3
plug-ins/FractalExplorer/	112 (0.2%)	9451 (0.5%)	84.3
devel-docs/libgimp/	102 (0.1%)	8731 (0.5%)	85.5
plug-ins/ipea/	191 (0.3%)	8307 (0.5%)	43.4
devel-docs/app/	92 (0.1%)	6690 (0.4%)	72.7
plug-ins/script-fu/	298 (0.4%)	6479 (0.4%)	21.7
devel-docs/libgimp/tmpl/	356 (0.5%)	6103 (0.3%)	17.1
app/xcf/	239 (0.3%)	5822 (0.3%)	24.3
app/basel/	537 (0.7%)	5782 (0.3%)	10.3
plug-ins/alpha/	102 (0.1%)	5596 (0.3%)	54.8
plug-ins/map-object/	42 (0.1%)	5279 (0.3%)	125.6
libgimpbase/	269 (0.4%)	5137 (0.3%)	19.0

Figure 4.1.2.1: Contribution of Mitch

Activity in Directories			
Directory	Changes	Lines of Code	Lines per Change
po/	2171 (3.7%)	1299827 (36.4%)	598.7
po-plug-ins/	1782 (3.1%)	956011 (26.8%)	536.4
/	8674 (14.9%)	171358 (4.8%)	19.7
po-script-fu/	1553 (2.7%)	120591 (3.4%)	77.6
plug-ins/common/	3806 (6.5%)	91622 (2.6%)	24.0
app/pdb/	973 (1.7%)	83229 (2.3%)	85.5
po-libgimp/	1569 (2.7%)	67497 (1.9%)	43.0
app/core/	3363 (5.8%)	54043 (1.5%)	16.0
plug-ins/imagemap/	534 (0.9%)	44289 (1.2%)	82.9
app/	2562 (4.4%)	42872 (1.2%)	16.7
app/widgets/	2906 (5.0%)	42391 (1.2%)	14.5
app/tools/	3051 (5.2%)	41896 (1.2%)	13.7
libgimp/	1579 (2.7%)	36375 (1.0%)	23.0
libgimpwidgets/	1285 (2.2%)	31023 (0.9%)	24.1
po-perl/	59 (0.1%)	27712 (0.8%)	469.6
app/display/	1233 (2.1%)	20139 (0.6%)	16.3
app/print/	1487 (2.6%)	19848 (0.6%)	13.3
po-tips/	98 (0.2%)	18547 (0.5%)	189.2
app/hase/	805 (1.4%)	18168 (0.5%)	22.5
app/config/	884 (1.5%)	16210 (0.5%)	18.3
devel-docs/libgimp/	212 (0.4%)	15788 (0.4%)	74.4
app/print-funcs/	215 (0.4%)	14913 (0.4%)	69.3
app/dialogs/	1121 (1.9%)	14636 (0.4%)	13.0
devel-docs/libgimp/tmpl/	396 (0.7%)	14603 (0.4%)	36.8
plug-ins/print/	330 (0.6%)	13726 (0.4%)	41.5
libgimp/	572 (1.0%)	13434 (0.4%)	23.4
plug-ins/app/	303 (0.5%)	12573 (0.4%)	41.4
devel-docs/app/	249 (0.4%)	12518 (0.4%)	50.2
app/plug-in/	1059 (1.8%)	12463 (0.3%)	11.7
plug-ins/ipea/	345 (0.6%)	12390 (0.3%)	35.9
plug-ins/psd/	31 (0.1%)	11630 (0.3%)	375.1
plug-ins/script-fu/	494 (0.8%)	10887 (0.3%)	22.0
plug-ins/script-fu/scripts/	1408 (2.4%)	9836 (0.3%)	6.9

Figure 4.1.2.2: Contribution of Neo

Based on this information the user mitch was developing on more places in the project while neo mainly developed in the po folder. This is why mitch can be named as the chief developer and second neo.

4.2 Main developers quit

In the previous chapter the users mitch and neo were considered to be the chief developers. What happens if the chief developers quit the project. In the image the users mitch and neo are left out. They are gray. The other users remain in color. As the image shows there is a lot of gray, except in the beginning of the project. The user yosh(light green) is then very active.

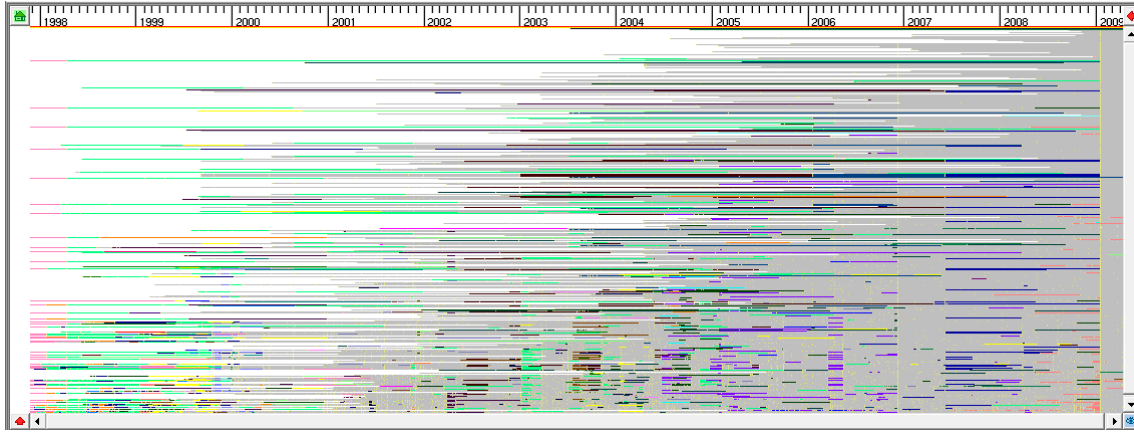


Figure 4.2.1: Developer contribution when neo and mitch left out

If mitch and neo stop working on the project then it would be very hard to continue creating a better application. It would take some time to get a new chief developer. Muks(dark blue) together with martinn(pink) would be good candidates for the new chief developers. They committed about 25% of the project together. This could be a problem for further development in the other 75% of the project.

4.3 Correlation between type of files and developers

Almost the whole project consists of .c and .h files. The files in toxic-green are .po files. This are translation files for the application. The files in the image are grouped on file type and sorted by creation time.

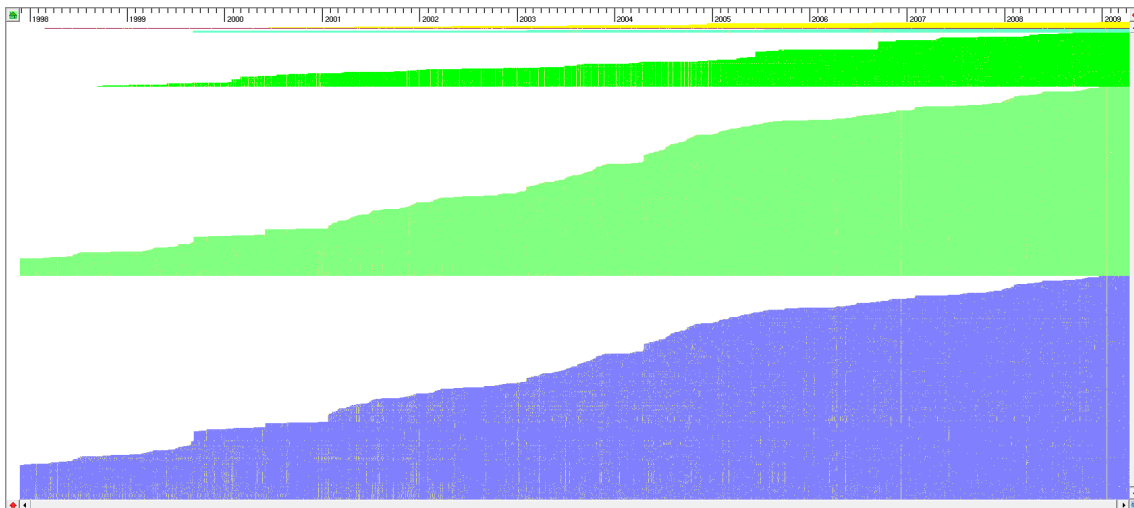


Figure 4.3.1: Files (from top to bottom: .sh, .py, .txt, .po, .h, .c)

In the next image the horizontal bars are still the file types, but the colors are different authors.

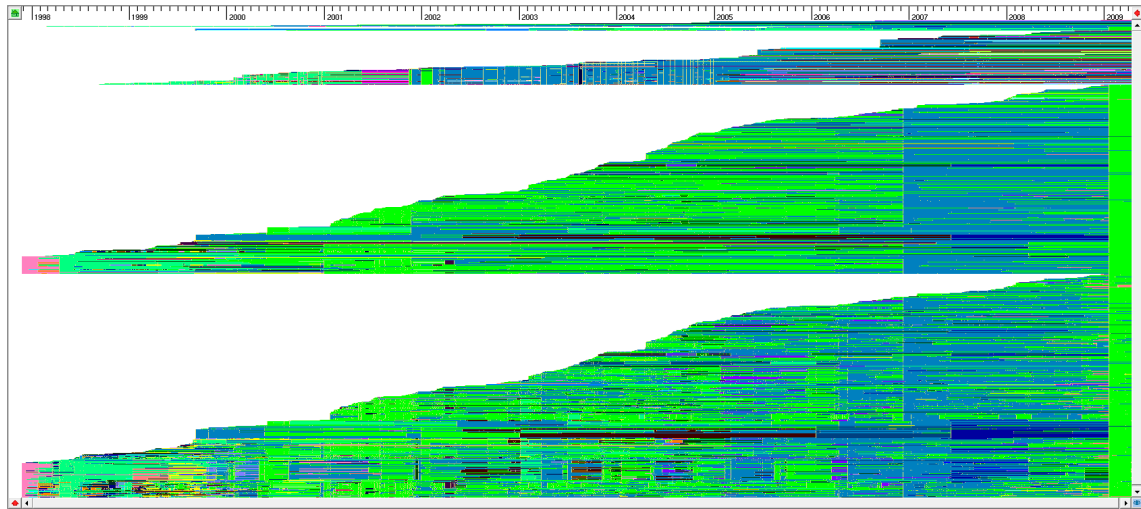


Figure 4.3.2: Developers that worked on the file-types

As the image shows there is no correlation between the type of files and the developers. This is because there are only .c and .h files. The only thing to mention is that the .po files are created and changed mostly by the same user. See the following image.

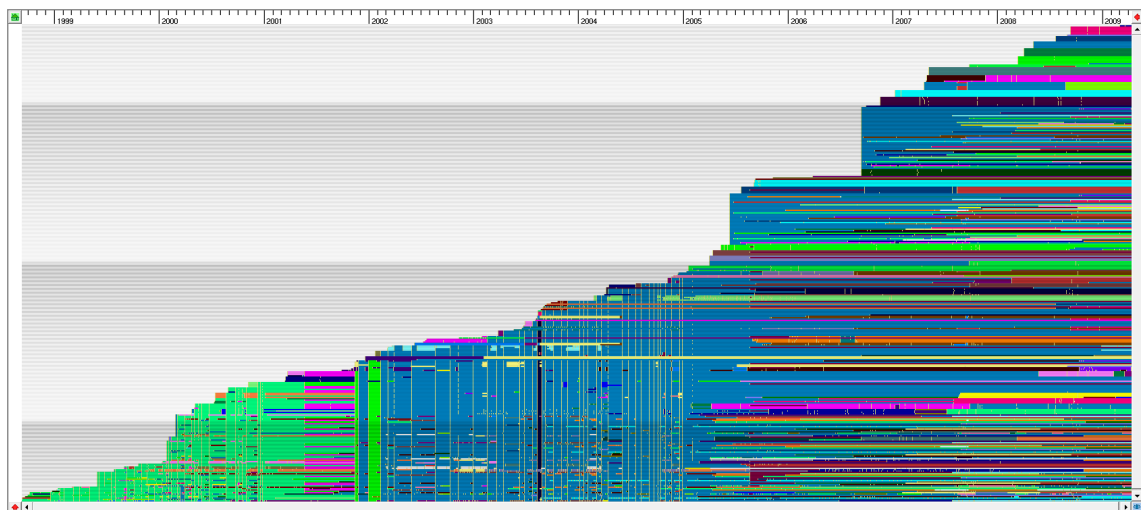


Figure 4.3.3:

4.4 Correlation between location of files and developers

The following steps are taken to get the following image:

- In the Metric tab select folders filter
- Select all folders
- File arrange is sort alphabetically

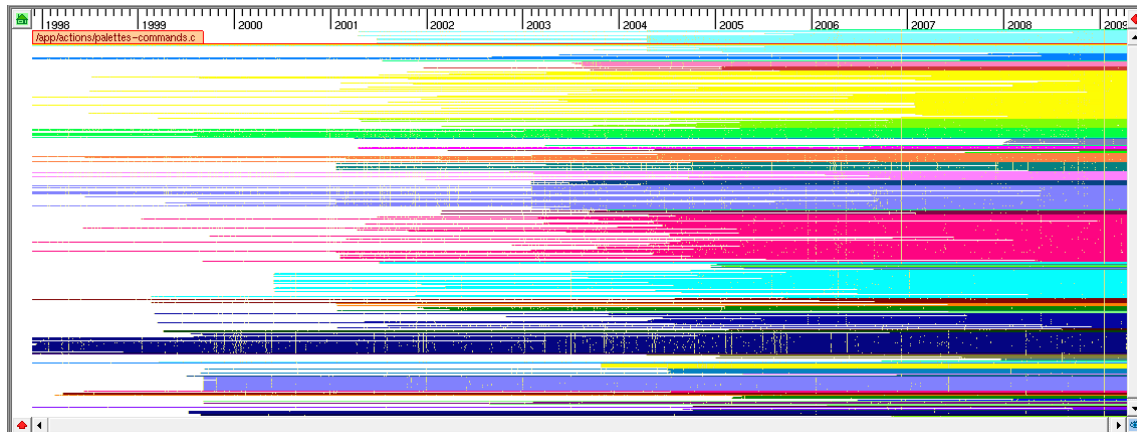


Figure 4.4.1: Directories of the repository

Every different color is a different folder. There is no significant correlation between the location of the files and the developers. There is a developer who did one thing in the project. The user maurits did a lot in the /plug-ins/imagemap folder. The dark-brown color is the user maurits. See the following screen-shot.

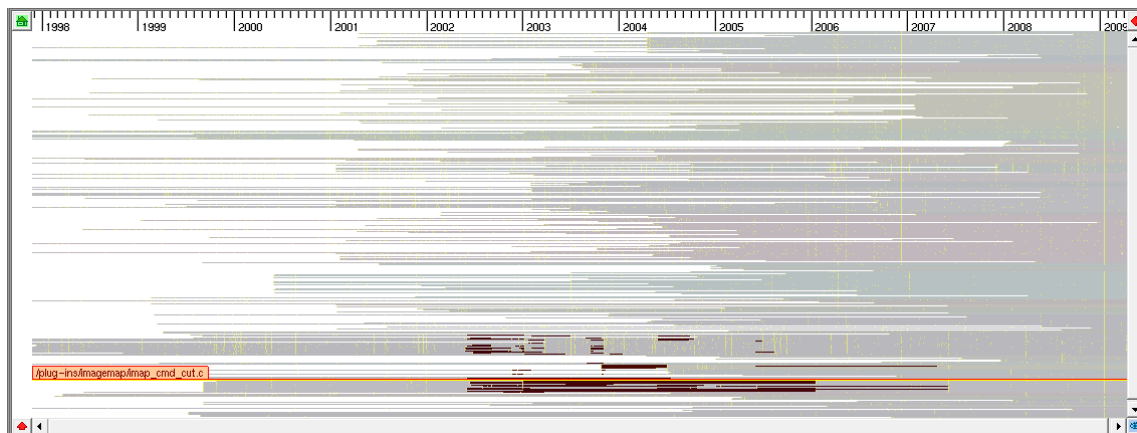


Figure 4.4.2: Developer contribution to directories

If we take a closer look at the two chief developers then we see that neo did a lot of work in the po and the po-plug-ins folder. The images in chapter 4.1 show where the users were developing. Take a look at the mitch user shows that the user was developing mostly in the app folder.

5 Code size analysis

"First, use the "Lines of text counter" calculator to compute the lines-of-code (LOC) metric for the source code files in the repository. This should generate the 'Code size' metric. Be forewarned, this plugin might take quite some time to execute, as it needs to actually access the contents of the source files. After the 'Code size' metric is available, answer the following questions."

First thing we need to do is to get the file contents of the files. To prevent retrieving the contents of files which we are not going to use anyway, we first create a view called '**sources**'. These files are needed for calculation of the lines-of-code (LOC) attribute.

1. In the metrics tab we check 'File type'.
2. We filter on the types '.h, .c, .py, .sh, .txt', which we consider as the main source file types.
3. In the evolution view we group by type.
4. We zoom in in the evolution view and select the colored files.
5. Yet we have created a view, which we rename to '**sources**'

Now a view is created of the files which contents we want to retrieve. This is achieved by clicking the 'Update contents' button in the 'Projects' tab. This command is very time consuming; it took us 4 days (96hours) to retrieve the data.

When the contents of the source files were retrieved, to analyze the size of the code, we still needed to count the lines-of-text value for each file. This we achieved by means of the 'Lines of text counter' plug-in in SolidTA. We expected this to be an I/O intensive operation, however the benefits of a (last generation) SSD are huge when doing this: the total run-time of the lines of text counter is about 1 minute.

However, the line of text counter generated a 'Lines of text' metric in SolidTA, when we were expecting the 'Code size' metric. After a 7 hour lasting execution of the CCCC calculator we indeed acquired the 'Code size' metric, where LOC (lines of code) appeared one of the selectable attributes.

Before continuing further analysis, first we show a comparison of 'lines of text' and the 'lines of code' metrics.

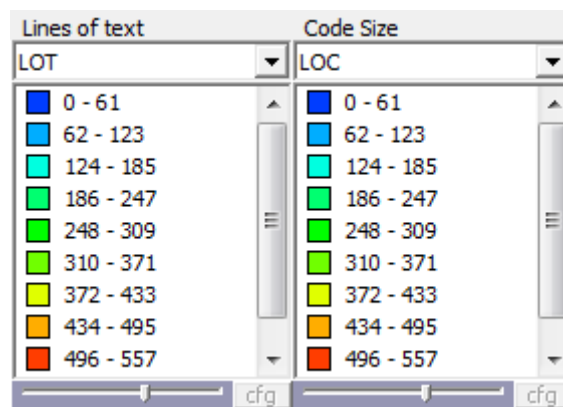


Figure 5.0.1: Legend

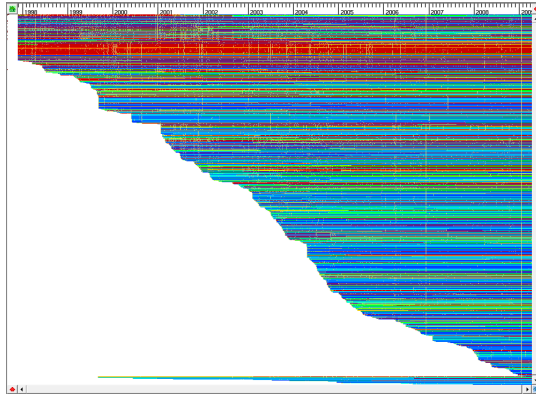


Figure 5.0.2: Lines-of-Text

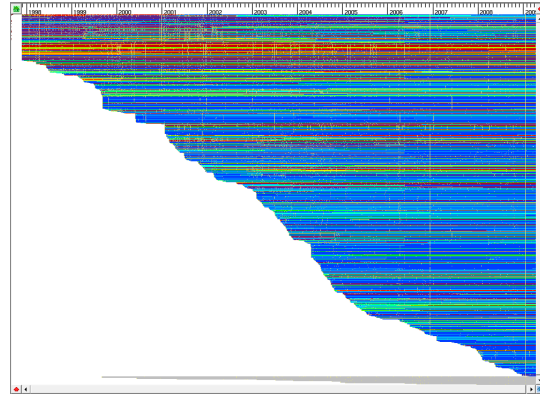


Figure 5.0.3: Lines-of-Code

The CCCC calculator was able to calculate metrics for only a subset of the files, for which we have a LOT calculation. The 'Code Size' metric only supports a sub-set of source-code files; *.txt and *.py files are not supported. Fortunately the amount of these files is relatively small (gray lines in 5.0.3).

From the figures 5.0.2 and 5.0.3 it becomes clear that the amount of LOT is greater than the amount of LOC. Of course this is to be expected as LOC is a subset of LOT; from all LOT some lines are code, while others are white-space, or other non-code lines.

The rest of the code is analyzed using a selection of this file subset. This selection can be created by grouping by 'Code size', zooming in and finally selecting all colored lines. We call the selection **'sources_measurable'**.

5.1 Code size evolution

"How is the size of code files evolving in the project? Are source code files growing or shrinking on the average?"

To analyze the file size evolution we first take a look at the global project evolution. For this we:

1. Select the Code Size (LOC) metric.
2. Select all attributes in the LOC menu.
3. Right-click on the selected attributes and select 'show evolution'.
4. If the trend view was not enabled already, this one has to be enabled too by pressing the 'bar' button in the top-right area of SolidTA.

The initial view shows us the average file size (in terms of LOC):

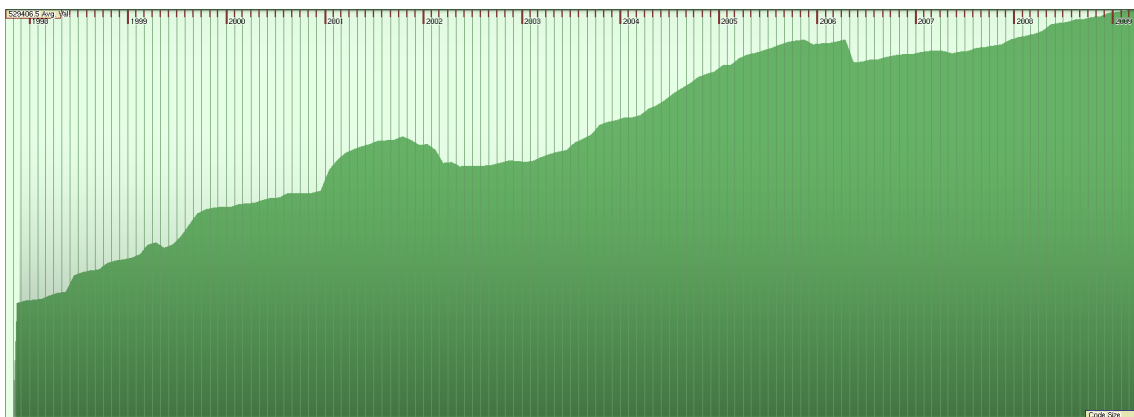


Figure 5.1.1: Average code size (LOC) vs Time

For a better overview of the groups we have also taken a look at the code size evolution in terms of amount of files per size group (groups as defined in 5.0.1). This was achieved by right-clicking the evolution view, selecting the 'Flow' as display type and once again right-clicking the evolution view, this time choosing 'File count'. The result is shown in figure 5.1.2.

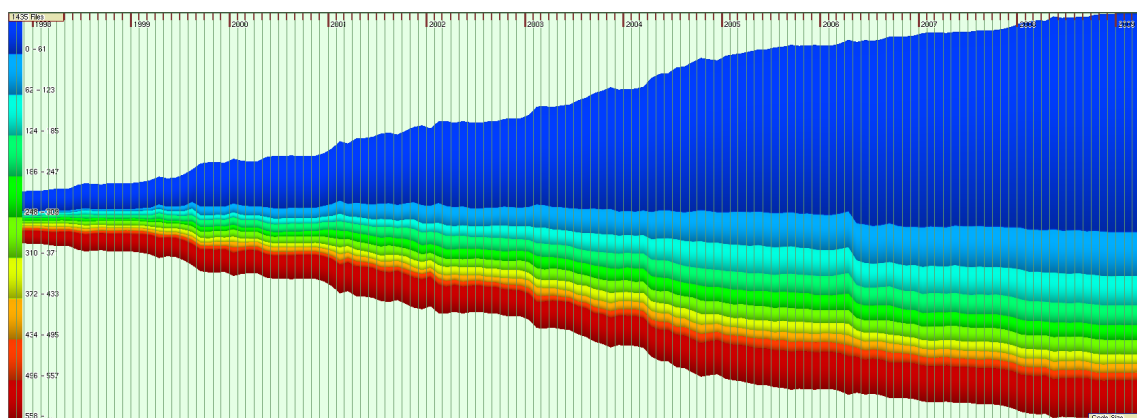


Figure 5.1.2: Amounts of files (Code size (LOC) in colors) vs Time

From 5.1.2. it is clearly visible that the relative amount of small files increases overtime; in the beginning the ratio of small files is about 40% of all files, while this ratio increases to more than 50%. The total amount of large files does increase overtime, however the ratio, initially around 30%, appears to decrease to around 10%.

Figure 5.1.1 indicates that the average code size increases over time. However, it is not possible to say whether old source files are growing or shrinking on the average: by adding new small files, the

average code-size will be influenced to appear smaller. This does not indicate anything on the already present files though. We think the average size of old source files will not grow in the same way as in 5.1.1. For this every file's evolution should be analyzed separately. As an sample we take the files from the initial commit and analyze their evolution for a better picture.

For this we sort all files by their creation time, zoom in at file level and select all files that are present from the beginning. This selection is called '**sources_initial**'. On this selection we perform the same steps as for 5.1.2. The result is shown below:

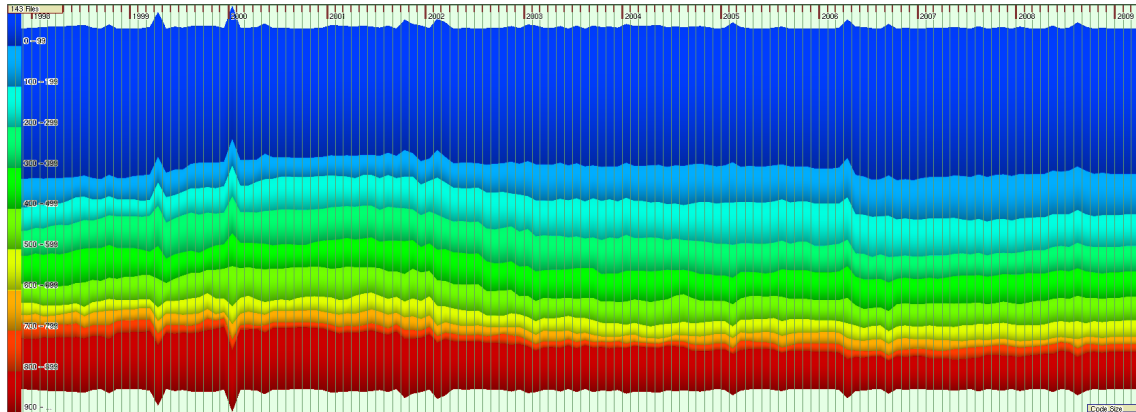


Figure 5.1.3: Code size (LOC) evolution of initially committed files

Figure 5.1.3 confirms our assumption: files that were large initially do not decrease in size, or show a slight size decrease at best.

In figures 5.1.1 - 5.1.3 a sudden decrease of complexity is visible in the first half of 2006. The code complexity is increasing, to decrease suddenly (the 'hitch' in light-blue to yellow layers). Combining this with the information from the global history overview, where no significant change is visible in the amount of source-code files, we conclude that some re-factoring took place here.

"Which are the fastest growing files? Which are the files that shrink the most?"

To find the files that grew/shrunk the most, we take the following steps:

1. Take the '**sources_measurable**' selection.
2. Now we want to filter out the largest files only, as these files could show the greatest fluctuation in size. This is done by selecting the bottom item in the selection menu (5.0.1) of Code Size (LOC).
3. We group the evolution view by Code Size. Now the largest files are grouped together, we create a new selection of them called '**sources_largest_size**'.
4. Continuing from this selection, we now select the smallest files only, group these by Code Size and make a new selection of these files called '**sources_changed_size**'.

The files that remain in the last selection have been in both the 'largest' and smallest state at some time. To show the changes (from small-to-large or large-to-small), we enable both the smallest and largest LOC item. The intermediate items are left out to emphasize the changes. Finally we sort the whole evolution by time for the clearest overview. The results are shown below in figure 5.1.4.

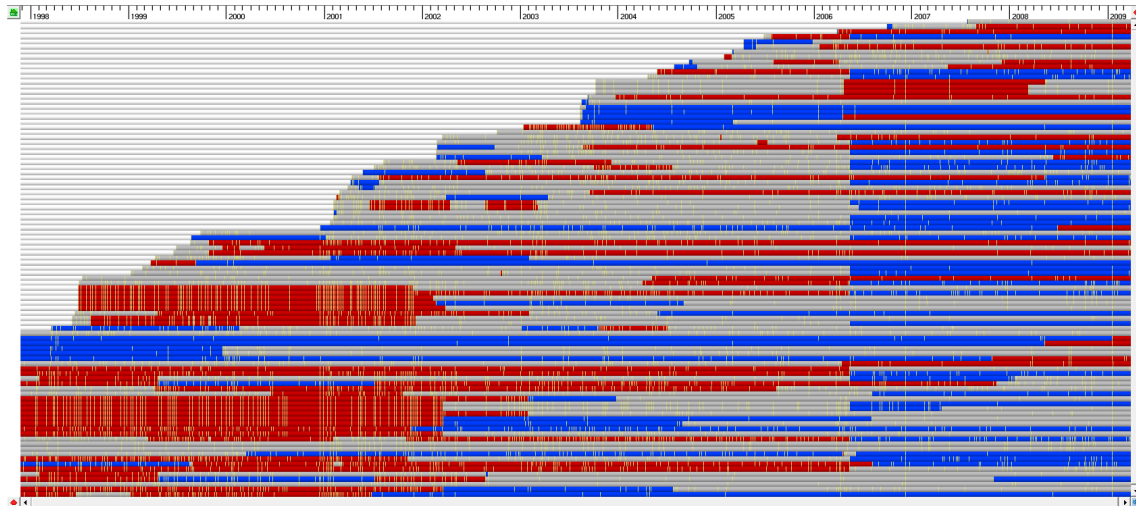


Figure 5.1.4: Files with greatest size fluctuation.

Files that decrease in complexity start red and evolve to blue. For files that increase in size over time, the opposite is true: these files start blue and end red. The less gray space (gradual evolvment) between the red and blue, the faster the files have grown/shrunk. We think the names of the actual files are not relevant for this analysis, though they can be retrieved by opening the '**sources_changed_size**' selection, contained in the SolidTA data.

To show this being just a fraction of all files that have been large at some point, we show the result of enabling small file indication in the selection '**sources_largest_size**':

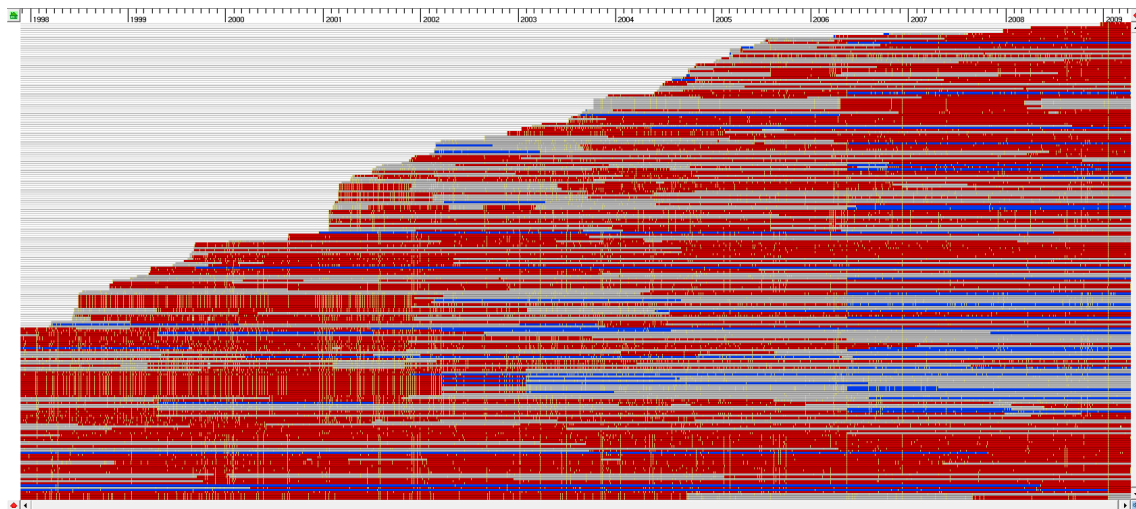


Figure 5.1.5: Evolution of largest files (overall)

Figure 5.1.5 confirms our assumption again: files that were large initially do not decrease in size, or show a slight size decrease at best.

5.2 Code size vs Total Size

"Group all files, in the evolution view, based on the 'Code size' attribute (right-click in the evolution view, and then 'Group selected'). How much is source code, in terms of percents, from the total project size (in terms of files)?"

To show the ratio of source files over the total amount of files, based on the 'code size' attribute, we perform the following steps:

1. Enabled the Code Size metric (LOC)
2. Right-click the evolution view, sorted on creation time
3. Right-click the evolution view and group by Code Size.

The result of this is shown below:

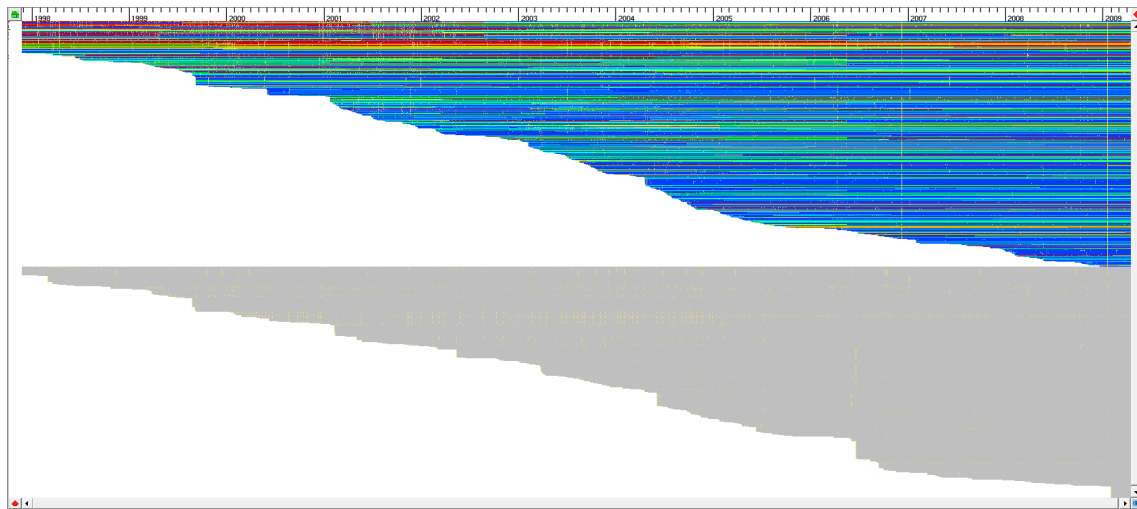


Figure 5.2.1: Code size as part of total size

Roughly a half of the amount of the repository's files are source-code files (as interpreted by the CCCC calculator). This should not be confused with the amounts in section 2.2, where we talk about total file sizes, instead of file counts.

6 Complexity analysis

6.1 Most complex files

Which are the most complex source files in the entire project?

To find the most complex files we need to undertake the steps listed below. Assumed is that the CCCC calculator was executed before (as in section 5).

1. Enable the **sources_measurable** selection.
2. Go to the Metrics tab and select Complexity.
3. Select 'Total complexity' in the drop-down menu that just appeared.
4. Move the slider to the right until the screen shows less red.
5. Select the bottom menu-item (in this case 162 - ...).
6. Sort the evolution view on creation time for better overview.
7. Group the evolution view on complexity. If the overview contains more than 10% of red lines, repeat from step 4 until this is not the case anymore.

These steps result in figure 6.1.1.

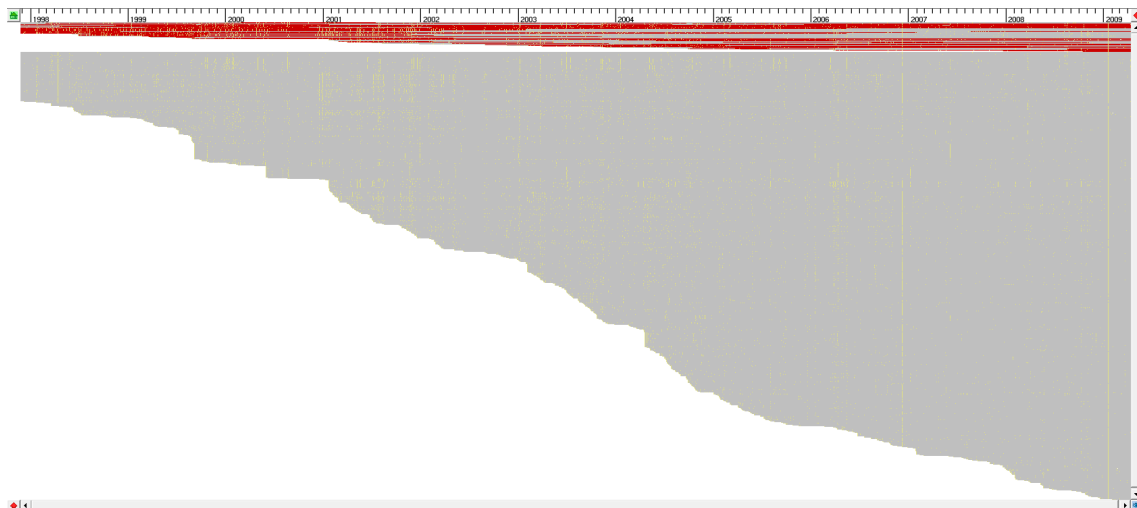


Figure 6.1.1: Most complex files

As we need to look at this set of source files in the next section, we create a selection of these files in the same way we did in previous sections. We call this selection '**sources_largest_complexity**'.

6.2 Complexity fluctuations

Are there files on which the complexity decreases significantly in time? Which are these?

Are there files in which the complexity increases significantly in time? Which are these?

To find which files' complexity decreases in time significantly, if any, we use the same approach as in 5.1: within the most complex files we look for files which at some time were not complex at all.

1. We start off loading selection '**sources_largest_complexity**'.
2. In the metric configuration we select only the top menu item for complexity (in this case 0 - 15). All files that were not complex at some point show up in blue.
3. We group the evolution view by complexity.
4. From these files we create a new view called '**sources_changed_complexity**'
5. Finally selected the bottom item from the complexity menu, while keeping the top item selected (hold CTRL). For a better overview sort the view on creation time.

These steps result in the overview shown in figure 6.2.1.

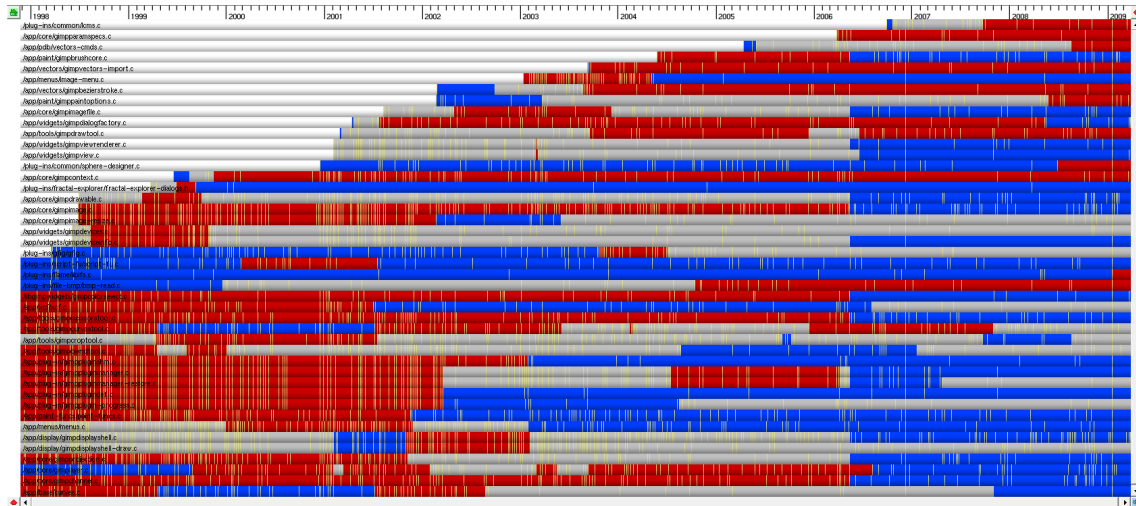


Figure 6.2.1: Files with greatest complexity fluctuation.

For this analysis the same holds as in section 5.1: files that decrease in complexity start red and evolve to blue. For files that increase in complexity over time, the opposite is true: these files start blue and end red. The less gray space (gradual evolution) between the red and blue, the faster the files have grown/shrunk. We think the names of the actual files are not relevant for this analysis, though they can be retrieved by opening the '**sources_changed_complexity**' selection, contained in the SolidTA data.

To show this being just a fraction of all files that have been highly complex at some point, we show the result of enabling low-complexity indication in the selection '**sources_largest_complexity**':

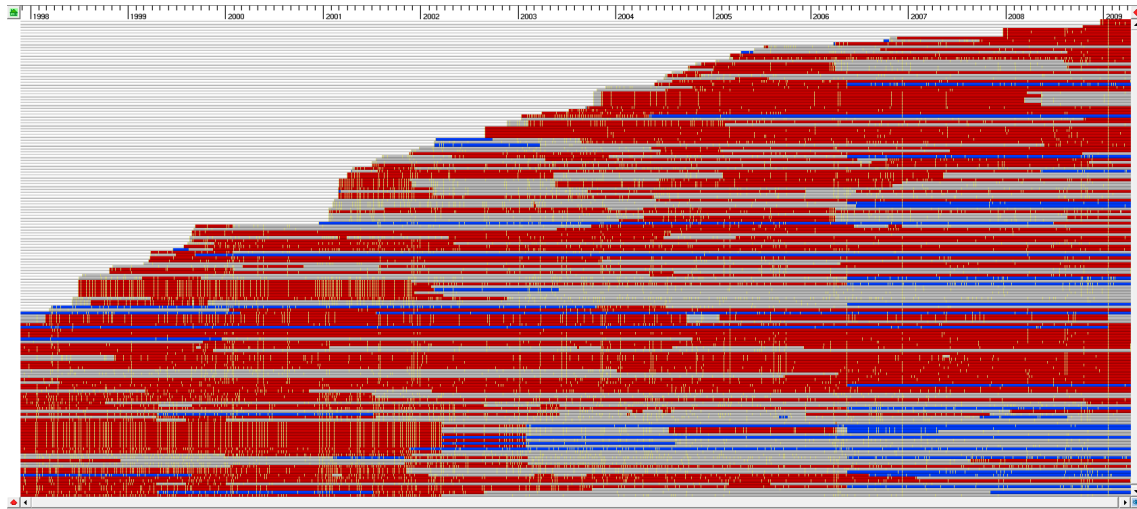


Figure 6.2.2: Evolution of most complex files (overall)

6.3 Complexity correlations

"For the above files (high complexity and/or complexity rate of variation), are these highly active files (with many changes), or not?"

To find out whether a correlation exists between the rate of variation (amount of commits) and complexity we undertake the following steps:

1. We open the **sources_measurable** selection.
2. In the metrics tab we enable Complexity, choose Total complexity and finally select the bottom item (144 - ...).
3. Finally we sort the evolution view on activity.

In figure 6.3.1. the result of these steps is shown. The amount of commits is shown on the left side.

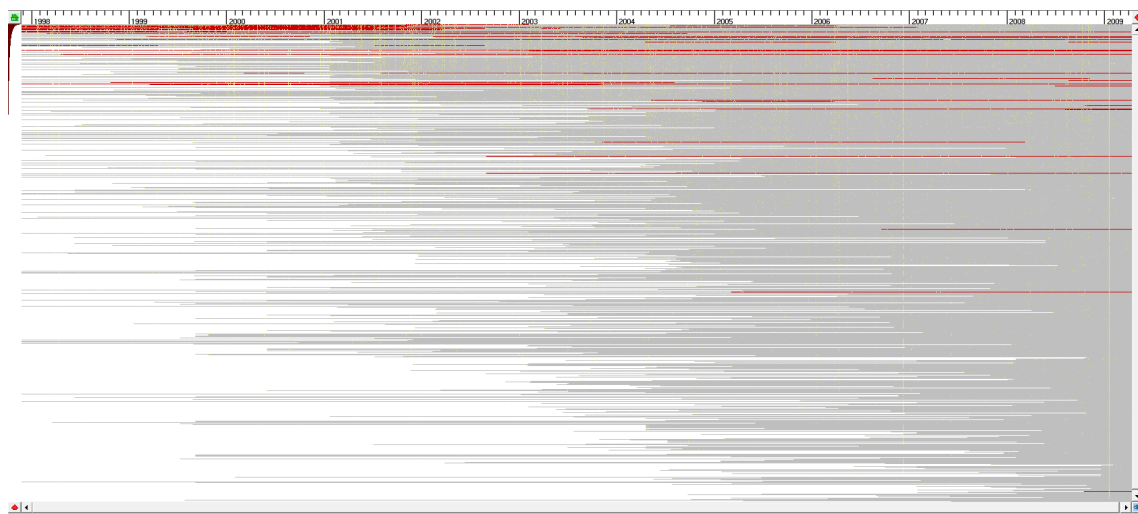


Figure 6.3.1: Source files sorted on activity, complex files in red.

Indeed, the most complex files are mainly on top of the image. From this we conclude a correlation between complexity and activity exists:

1. Files with high activity are likely to be complex.

"Is there some correlation between the highly complex files and the file size (measured in lines of code, as done in Step 4)? Can you find a direct correlation? Or an inverse correlation? Or is there no visible correlation to be found?"

To find out whether a correlation exists between the complexity and size of files we will first compare the fluctuation of these aspects on the set of most size-fluctuating files. For this we need to:

1. Open the **sources_changed_size** selection.
2. In the metrics tab enable Complexity and select the largest and smallest items.
3. In the metrics tab enable Code size and select the largest and smallest items.
4. Change the colors for large code size to black and small code size to yellow. This will yield a better distinction between the two properties.

When switching observer between the two properties, it will look as if the red changes into black, while the blue changes into yellow. This is shown in figures 6.3.2.1 and 6.3.2.2.

To show that the same is the case for all large files, steps 2 - 4 are performed on the selection **sources_largest_size**. The results of this are shown in figures Figure 6.3.2.3 and Figure 6.3.2.4.

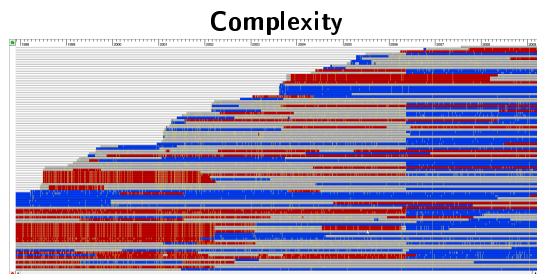


Figure 6.3.2.1: Most size-fluctuating files

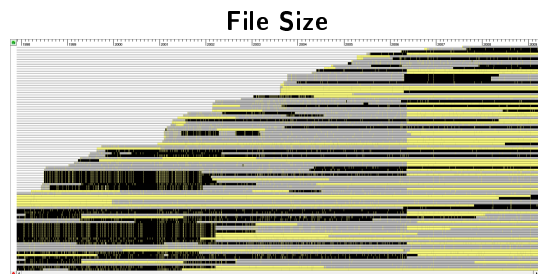


Figure 6.3.2.2: Most size-fluctuating files

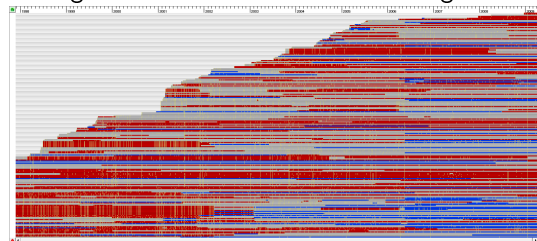


Figure 6.3.2.3: Largest files

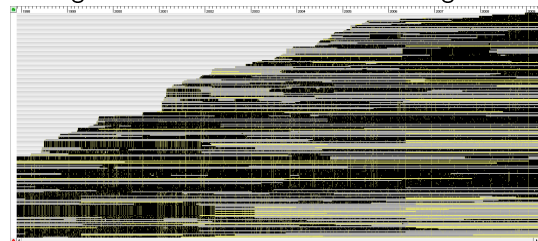


Figure 6.3.2.4: Largest files

From this it seems plausible that file size and complexity are directly correlated. We perform a trend check on both aspects to see whether they really evolve in the same way. This is achieved by enabling evolution on both the Complexity and Code size metrics. The graph visible in the trend view is shown in figure 6.3.2.5.

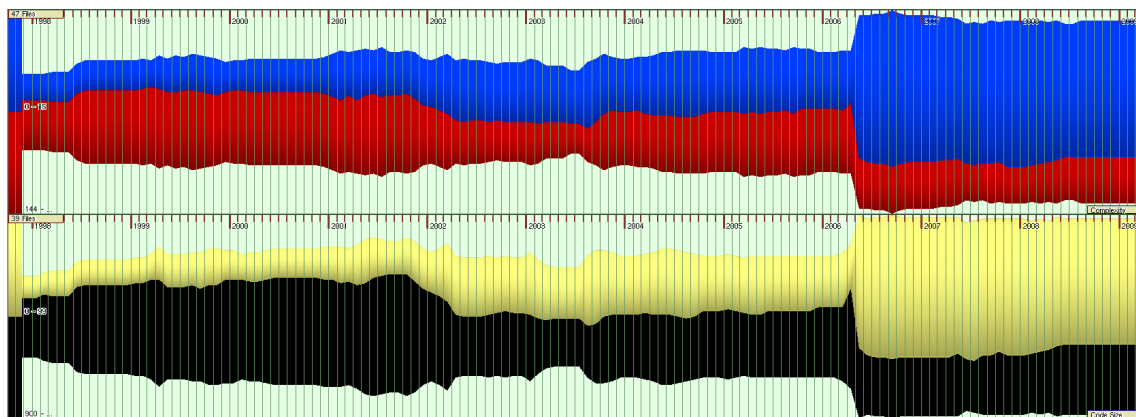
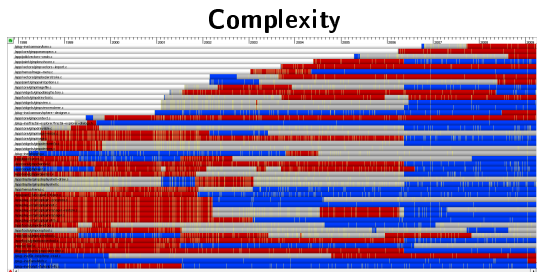
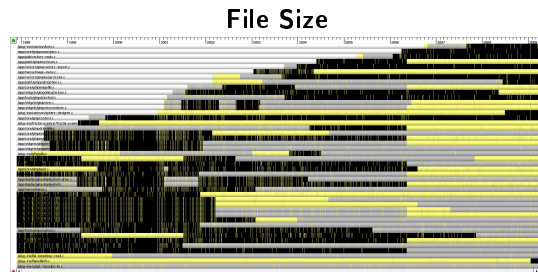


Figure 6.3.2.5: Trends of complexity and file size of most size-fluctuating files

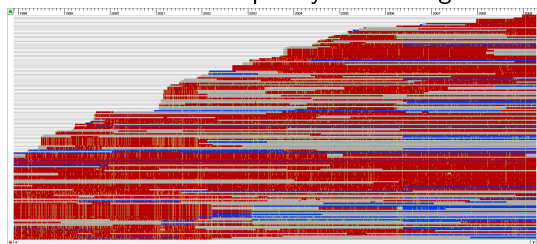
We can conclude that the complexity of files is indeed significantly correlated to the file size, though the reverse is not necessarily true. To find out whether also file size is significantly correlated to complexity, we perform the previous steps on the selections **sources_changed_complexity** and **sources_largest_complexity**. The results are shown in figures 6.3.3.1 - 6.3.3.5.



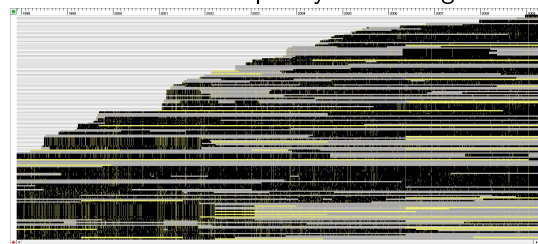
6.3.3.1: Most complexity-fluctuating files



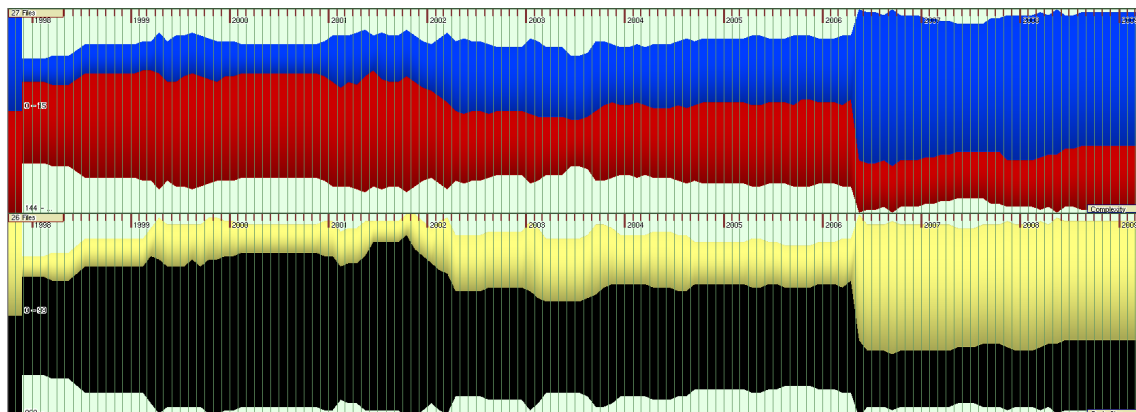
6.3.3.2: Most complexity-fluctuating files



6.3.3.3: Most complex files



6.3.3.4: Most complex files



6.3.3.5: Trends of complexity and file size of most complexity-fluctuating files

As the same relation appears from the other side we can conclude complexity and file size are correlated one-to-one;

1. If file size increases, complexity will increase.
2. If complexity increases, file size will increase.

7 Conclusion

"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."

Complexity

From section 6 we can conclude that the complexity of the project has not increased drastically over time. As in section 6 a one-to-one correlation was determined between complexity and file size, we can use file size to say things about the complexity.

While the total amount of files has grown, the relative amount of complex files has decreased (figure 5.1.2). Furthermore, the files that were committed initially do not change in complexity, or even show a slight decrease. This indicates that new functionality is added to the code in a well organized way.

Figures 5.1.1 - 5.1.3 all show a hitch in complexity in 2006. When we look at figures 5.1.4 and 5.1.5 we see that at this same point many complex files show a huge decrease in complexity. Some thorough refactoring took place there, reducing the average complexity significantly. Besides, this was not the only point of refactoring, as small sudden decreases are visible over the whole time-line.

We conclude that the complexity of The GIMP is well maintained.

Modularity

According to the analysis in section 2, The GIMP appears to be designed for modularity as it contains an /app and a /plug-ins directory. When looking inside the /app and /plug-ins folder, they both appear to be split up again in many sub-folders. This and a maintainable complexity are the strongest high-level signs that modularity is applied.

Maintainability

Maintainability is not a directly measurable aspect of the source-code. We can however derive whether a project is well maintainable by looking at its performance in complexity and modularity. Complexity has an inverse effect on the project's maintainability, while modularity should effect the maintainability in positive way.

As the project has a low complexity and a high modularity, we conclude that it's maintainability is quite good.

Overall quality of software and development

From the previous points we derive that the high-level quality of the software appears to be good.

The software development appears to be dominated by neo and mitch

It is good that the project has multiple main developers, who all are familiar with each others code (according to figure 4.3.2). On the other side, a few developers being responsible for the largest part of the code-base forms a risk for the project. For example if one of them would leave, the project could bleed to death.

Whether this happened in the beginning of 2009, when the GPL3 license was introduced, is not entirely clear. It seems that neo - and many other developers together with him - stopped contributing to the project once the new license was added. This is visible in the low amount of new releases over the last 2 years. The road-map of The GIMP shows new major releases, though we assume it is still an active project.

8 Evaluation

In this section we discuss our experience of carrying out the work for this report. We start with the discarded repositories, followed by a short discussion on SolidTA, to finish with a global evaluation.

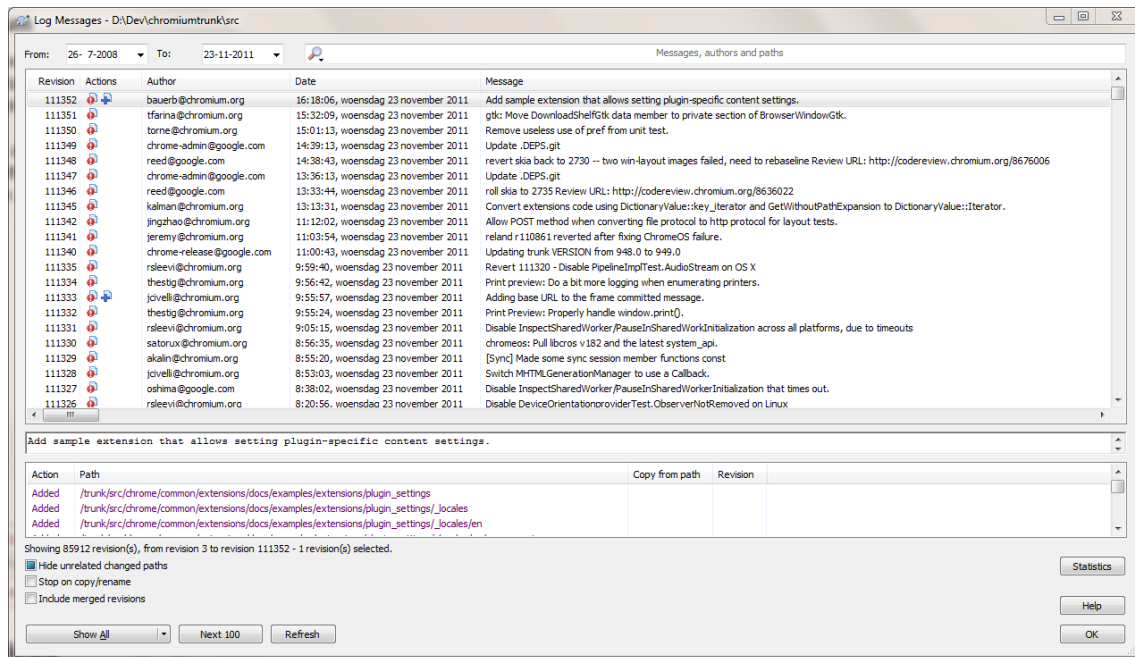
8.1 Discarded repositories

Finding a project with workable properties (as listed in the introduction) was a tough job. In this (sub)section we will list the repositories we have attempted to use. For every repository the source of the (blocking) issue will be discussed.

8.1.1 Chromium

The first project we have given a try was Chromium. The link for this repository is <http://src.chromium.org/svn/trunk>. The repository of this project was created in 2008 and has been extremely active ever since.

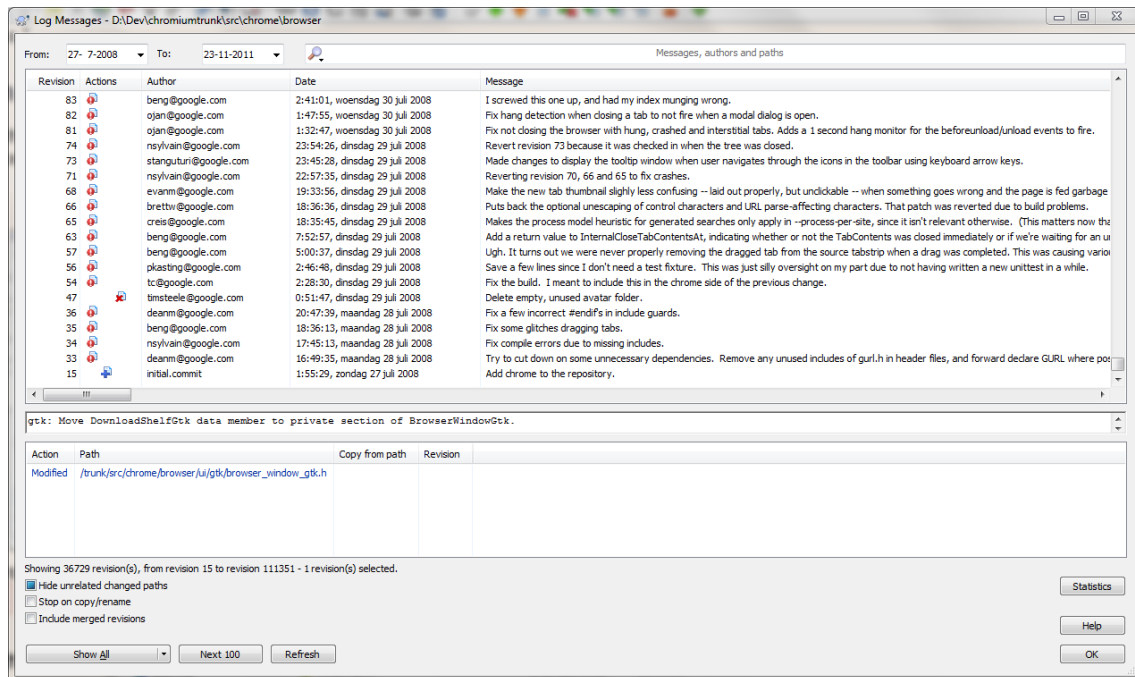
Naively we started with checking out the whole (/trunk) of the repository, which resulted in a code-base of about 80,000 files, 7.4GB in total. As retrieving this large amount of files lasted one night, we looked for a subset of the repository to analyze; we tried to analyze /src directory (30,000 files, 460MB) with SolidTA, not realizing that the application would need to retrieve all revisions.



8.1.1: Log of Chromium repository /src directory

A few hours after the initiation of file-list retrieval, we were able to see the file-list in SolidTA. However after a few days of updating version information, we began to wonder why it was taking so long without any visible progress, soon to find out that this repository is not only huge in size, but also in the amount of revisions (more than 100,000 in total, from which 85,000 relevant for this directory). We had mistaken the total amount of revisions to be around 10,000 initially.

We figured that taking a smaller subset of files could resolve the problem. Therefore we decided to analyze the source-code of the browser only, contained in /src/chrome/browser. This directory contains about 6,200 files, altogether about 55MB in size. The amount of relevant revisions was about 36,000.



8.1.2: Log of Chromium repository /src/chrome/browser directory

Retrieval of the file-list took about 2 minutes, version information required 4 hours, but getting the contents appeared impossible. SolidTA would take 2 days to retrieve about 25% of the contents, to stall thereafter. Because we were not familiar with SolidTA and therefore did not know how to continue this process, we repeated the content-retrieval 2 more times. As we were suspicious that this might not work at all, we started looking for other projects. After a week of still not having workable data we discarded the Chromium project.

8.1.2 FileZilla 3 client

A small project we have tried to obtain the data of was FileZilla 3. The link for this project is <http://svn.filezilla-project.org/svn/FileZilla3/trunk>. The repository of this project was created in 2004.

Again the first thing to do was checking out the repository with SVN. This did not take a long time. The check-out resulted in a code-base of 17.8MB in size, containing 928 files. The amount of revisions was 4280.

We were actually successful in acquiring all data with SolidTA, which was done to test whether we are able to retrieve the data of a small project at all.

Sadly the projects changes were only committed by three guys, of which one had 1 commit in total. Besides that the project was too small for a relevant analysis, according to the requirements listed in the introduction.

To satisfy all requirements we had to look further.

8.1.3 Octave

Octave is an open-source version of MatLab. The link to its repository is <https://octave.svn.sourceforge.net/svnroot/octave/trunk>. The repository of this project was created in 2001.

Again we started off with checking-out the repository, which progressed faster than what we had seen before.

The trunk of Octave contains 52.1MB of data, in a little more than 9,500 files. The amount of revisions was about 8734, which altogether was of acceptable size for a relevant analysis.

The reason for us to drop this project were the source-files, primarily existing of .m (MatLab) files; the CCC metric calculator is not able to process for code complexity analysis in SolidTA.

8.1.4 TortoiseSVN

Another project we tried was TortoiseSVN. The link to its repository is <http://tortoisesvn.googlecode.com/svn/trunk>. The repository was created in 2003

The project exists of 145MB of sources, contained in almost 7,000 files. The amount of revisions was with 22087 a bit high, but considering the speed of the check-out we would give it a try.

Sadly our hope vaporized shortly, when - for unknown reasons - SolidTA appeared unable to acquire the file-list.

8.1.5 XBMC

Xbox Media Center, which is the original name of the project, seemed interesting as one of us was familiar with compiling the software. The link to the repository is <https://xbmc.svn.sourceforge.net/svnroot/xbmc/trunk>. The repository was migrated to Git in the beginning of 2011.

The speed of the check-out was a bit disappointing. The repository contained 365MB of data in a bit more than 19,000 files. The project is subdivided in many components, enabling us to take a subset of files. Therefore we decided to give it a try.

Within some time SolidTA was able to retrieve the file-list. However, retrieving the version info would make SolidTA crash for unknown reasons, rendering XBMC unworkable.

8.1.6 VirtualBox

The last project that we tried before we came to The GIMP was VirtualBox. The link to its repository is <http://www.virtualbox.org/svn/vbox/trunk/src>.

The project was started in 2007, and has had an active development as it reached almost 40,000 commits in those years. The speed of this repository was amazing, resulting in excitement on our side. The total size of the data was 209MB in 13,350 files.

After the check-out, which took a very small amount of time, we were very disappointed when it appeared that the only 'user' that had ever committed was 'vboxsync'. This led us again to discard a project that almost qualified all requirements.

8.2 SolidTA

The opinions about SolidTA are divided. Some really like the tool, while others hate it. We think SolidTA is a powerful tool for gathering insight into a project's evolution. However, it does have some serious performance and stability issues, which we discuss next.

8.2.1 Performance

First thing we noticed is that the speed, with which data is retrieved, is simply horrible. Where a simple SVN tool, like TortoiseSVN, takes seconds to retrieve the whole file-list, SolidTA can take up to tens of minutes. The reason for this is not evident, but the consequences are.

Sadly, the minutes lost for file-list retrieval are acceptable compared with the time it takes to retrieve version info and/or file contents.

We are not completely aware of the way this is implemented, as we do not have access to the source-code of SolidTA. Though it is very clear that SolidTA retrieves the information file-by-file, instead of doing this in the same way as TortoiseSVN. The latter requests a complete revision log, which usually contains changes on multiple files. This results in TortoiseSVN showing the full revision history within minutes in worst case, where this takes hours in SolidTA. As far as we have been able to determine, there is no difference between information of the revision history and the version info.

Finally, the implementation of the last step - the retrieval of all contents in SolidTA - is questionable. For this information SolidTA requests the contents per file and per revision. We assume that an SVN server orders its files as received; files of revision 1 are written together, followed by files of revision 2, etc. By requesting each revision for each file it is likely that the server will have to search the hard-disk on every request. This could be prevented by requesting all files of a whole revision at a time. Although we do not have proof for this, we are convinced a lot of performance can be gained here.

8.2.2 Stability & reliability

SolidTA for some reason it skips the retrieval of some files every now and then. The reason for this could be a server-time out on the specific request, however it is not corrected by sending the request again.

SolidTA also has some strange stability bugs. We have not been able to find the exact reason for SolidTA refusing to retrieve the file-list and/or file-contents for some projects, or even crashing at this point. However, as these issues occurred on more than one machine, while the repositories were checked-out properly with TortoiseSVN, we think that the issue lies within SolidTA.

Furthermore, at some point one of us was not able to start SolidTA anymore, after installing SVN (command line executable), and adding its directory to the PATH system variable. Removing this directory from PATH rendered SolidTA functional again.

8.3 Software evolution analysis

Despite the problems we encountered when choosing a project and retrieving its data, we found the analysis quite interesting. Though SolidTA provides just some of the techniques discussed in the Software Maintenance and Evolution course, these techniques provide a powerful method to analyze and learn from the evolution of a project.

The assignment was set up to be completed using SolidTA, which provides primarily file-level analysis of a project. Other tools exist that can analyze on syntax-level, line-level, dependencies and many more. Though, for analysis of an unfamiliar project, this approach is probably best, as other analyses would require one to have far-going knowledge of the source-code.

Altogether the course and assignment have raised our interest in the relatively new field of software evolution analysis.

9 Appendices

9.1 Time consumption

In this section the time consumption of the most time consuming tasks are set out. We have not measured the time for reporting as this requires a complete hour registration, e.g. for checking and improving of section. Furthermore the main responsible persons are noted for every section/task.

Activity	Time spent	Author
----------	------------	--------

Finding a project

Investigating ³ Chromium repository	2 days	Avdo & Erik
Retrieving Chromium contents (aborted)s	2 weeks	Avdo
Investigating FileZilla repository	3 hours	Avdo
Retrieving & Processing FileZilla contents	2 days	Avdo
Investigating Octave repository	2 hours	Avdo
Retrieving Octave contents	1.5 days	Avdo
Investigating TortoiseSVN repository	2 hours	Avdo
Investigating XBMC repository	2 hours	Avdo
Investigating VirtualBox repository	2 hours	Avdo

Investigating GIMP repository

SVN check-out	30 minutes	Avdo
SolidTA file list	1 minute	Avdo
SolidTA version info	2 hours	Avdo

Retrieving & Processing GIMP contents

SolidTA retrieve contents	4 days	Avdo
Compute CCCC metrics	2 x 7 hours	Avdo
Compute StatSVN statistics	6 hours	Erik

Reporting

Report Section 1		Avdo
Report Section 2 on Chromium (discarded)		Avdo & Erik
Report Section 2.1, 2.2		Avdo
Report Section 2.3		Erik
Report Section 3		Erik
Report Section 4		Erik
Report Section 5		Avdo
Report Section 6		Avdo
Report Section 7		Avdo
Report Section 8		Avdo
Report Section 9		Avdo
Essay		Erik



**rijksuniversiteit
 groningen**

faculteit wiskunde en
 natuurwetenschappen

informatica

Software Maintenance & Evolution Essay Dependency Evolution Analysis

Avdo Hanjalic / s1553623

Erik Bakker / s2074893

November 24, 2011

Contents

1	Introduction	3
2	Data modeling	3
2.1	Assumptions	3
2.2	Graphs	3
2.3	Data model	4
2.3.1	Nodes	4
2.3.2	Edges	5
2.4	Optimization	5
3	Dependency data visualization	6
3.1	Mockup	7
3.2	Graph types	7
3.3	Advantage knowing particular graph structure	8

1 Introduction

Almost every software project nowadays uses a revision control system such as subversion. Project leaders are more and more interested in what kind of information the revision control systems can give them. By default the revision control system subversion give no more information then the commit logs. When analyzing the revisions it can give more detailed information about the development of the project. Here is where the visualization program comes in place. With dependency evolution analysis the information of the revisions can be used to display relevant information about the project.

2 Data modeling

2.1 Assumptions

Before making the data model, we made a assumption that the programming language of the code should be java. The revision control system should be subversion.

2.2 Graphs

The structure of a data model depends on what kind of data has to be stored and what kind of output has to be created. In this case the output are graphs. The following graphs are taken into account when creating the data model:

- A call graph
This graph shows the calls between functions in the entire project.
- A class inheritance graph
The inheritance graph displays the inheritance relations between different classes.
- A containment graph
A containment graph shows the relations between methods, classes, files, folder. E.g. method-in-class-in-file-in-folder.
- A build dependency graph
This graph is responsible for showing the dependencies between files. E.g. when file X is changed, then files Y and Z need to be rebuilt.

2.3 Data model

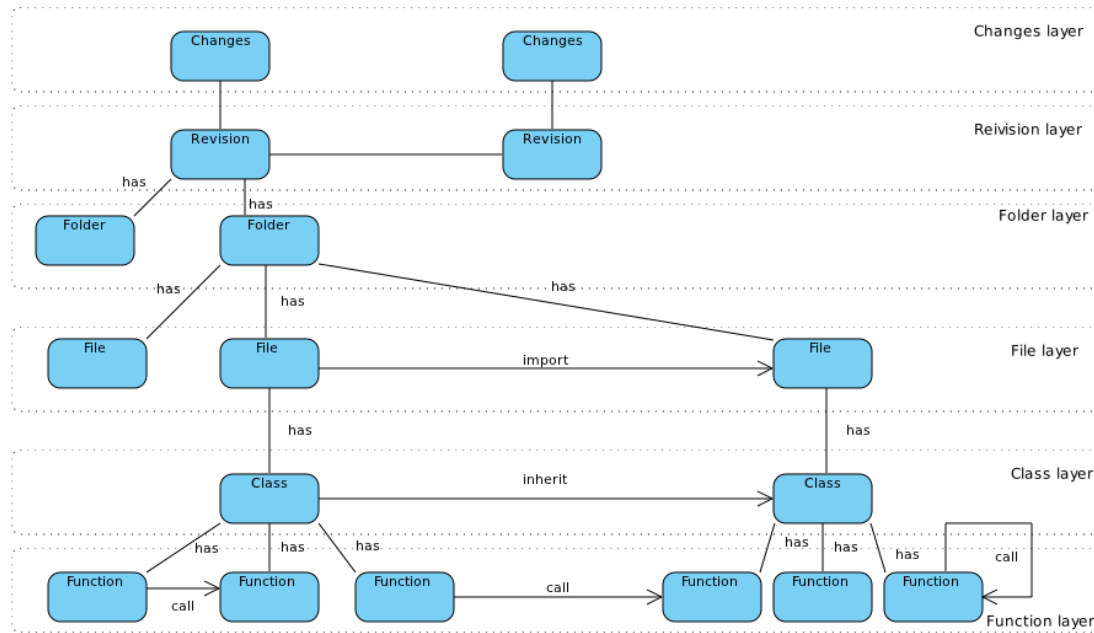


Figure 1: UML diagram data model

Figure 1.1 is the representation of the data model. First of all the content of the nodes (Revision, Folder, File, etc.) will be discussed and then the relations (edges) between the nodes.

2.3.1 Nodes

Content of node Revision:

- Revision number

Content of node Changes:

- Output 'svn diff'. Difference between current and previous revision.
- Changes of correspondences number

Content of node Folder:

- Folder name
- Correspondences number

Content of node File:

- File name
- LOC (Lines of Code in the file)
- Correspondences number

Content of node Class:

- Class name
- Correspondences number

Content of node Function:

- Function name
- Correspondences number

Every node except for the revision node has a Correspondences number. This number is equal for the function in all different revisions. Function *getName()* in revision 23 gets correspondences number F546346, then the same function in revision 24 gets the same correspondences number. It is possible that during the development the name of the function changes. When the function name changes the correspondences number will not change. To determine if the function name is changed the changed node can be used. Every change compared to the previous revision of the project code is stored in this node. This way changing the function name will be noticed and the same Correspondences number will be given to the function.

2.3.2 Edges

Every edge (relation) can be accessed from the corresponding nodes. Node Revision 2 can go to Node Revision 1 and Node revision 3. Every Revision node has a relation with the Changes node and the several Folder nodes. Every Folder node is linked with File nodes.

Every File node can have two different relations. A File node has a class inside it and a File node can import another file Node. There can be several relations between different files and a file can have several classes in it.

Every Class node can have several has function relations. A class can also have a inherit relation with another class.

The Function nodes can only have call relations. This call relations can be with another Function node or with itself (recursion).

2.4 Optimization

To optimise the performance of the visualization program an implementation of a data model per graph is required. This way only the first time the visualization program starts it takes a long time to generate the data information. Once this is complete, the graphs can be easily and quickly created.

A call graph only needs the function calls. The following data-model would be implemented to optimise the visualization program:

Function	
correspondencesNumber	varchar(255)
name	varchar(255)
functionCalls	varchar(99999999)
classCorrespondencesNumber	varchar(255)
fileCorrespondencesNumber	varchar(255)
folderCorrespondencesNumber	varchar(255)
revision	integer(10)

Figure 2: data model call graph

All functions will be inserted in this table:

- correspondencesNumber: this is the number of the function.
- name: this is the name of the function.
- functionCalls: this is a comma separate string with the correspondencesNumber of the functions which the current function calls.
- classCorrespondencesNumber: correspondencesNumber of the class where the function belongs to.
- fileCorrespondencesNumber: correspondencesNumber of the file where the function belongs to.
- folderCorrespondencesNumber: correspondencesNumber of the file where the function belongs to.
- revision: revision number.

For optimising the class inheritance graph the following table will be used:

Class	
correspondencesNumber	varchar(255)
name	varchar(255)
inheritCorrespondencesNumber	varchar(255)
fileCorrespondencesNumber	varchar(255)
folderCorrespondencesNumber	varchar(255)
revision	integer(10)

Figure 3: data model inheritance class

All classes will be inserted in this table:

- correspondencesNumber: this is the correspondences number of the class.
- name: the name of the class.
- inheritCorrespondencesNumber: this the correspondencesNumber of the class that is inherited.
- fileCorrespondencesNumber: the correspondencesNumber of the file the class belongs to.
- folderCorrespondencesNumber: the correspondencesNumber of the folder the class belongs to.
- revision: number of the revision.

For the containment graph the data model from figure 1 can be used. The hierarchy is clear in this data model.

Last but not least the build dependency graph. The table in figure 2 can be used to calculate dependencies on function level. The table in figure 3 can be used to calculate dependencies on class level. The last dependency is on file level. The following table will be created for this.

File	
correspondencesNumber	varchar(255)
name	varchar(255)
importCorrespondencesNumber	varchar(255)
folderCorrespondencesNumber	varchar(255)
revision	integer(10)

Figure 4: data model dependency graph

All files will be inserted in this table:

- correspondencesNumber: this is the number of the file.
- name: this is the name of the file.
- importCorrespondencesNumber: comma separated file which files are imported.
- folderCorrespondencesNumber: number of the folder this file belongs to.
- revision: number of the revision this file belongs to.

What happens if a new revision is committed?

All the files in the new revision will be analyzed and inserted in the data model in figure 1. The output of 'svn diff' will be inserted in the Changes node. Where needed the correspondences numbers will be altered. When the user once to see the graphs then the Changes node will be read and the changes will be executed on the relevant tables.

A disadvantage of this model is that it consumes a lot of disk space, but nowadays disk space is very cheap. A other disadvantage is that it consumes a lot of time the first time it has to check out all the revisions at the same time.

3 Dependency data visualization

Visualizing the dependency data is a challenging problem. The dataset can be highly complex, intertwined and quite large.

3.1 Mockup

The following image is a mockup of the interface it could look like.

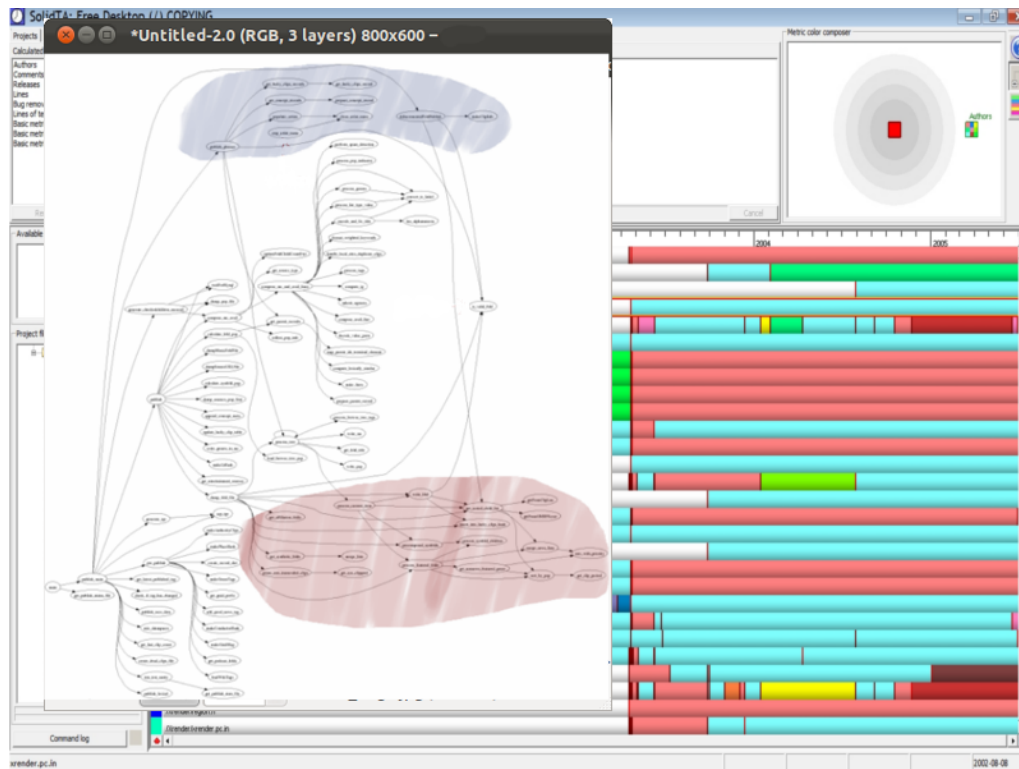


Figure 5: mockup user interface

Before getting this popup with a call graph the user has to select an amount of files which the user wants to use in the call graph. Once the user selected the files a right mouse click and then press 'gen call graph' is enough to create the popup shown in the mockup in figure 5. Every line between the circles are function calls. The blue and purple shading is a marking for all the functions within one single file or class. The name of the file or class is included in the legend (not implemented in the mockup) The function calls start at the left side in the popup.

With the use of the popup and selecting files the user can limit the amount of functions and function calls in the popup. The user can for example select all the files in one folder to see the dependencies in that particular folder. This way the visualization is easier to follow. A downside of this method is if the user selects all files with hundreds maybe thousand of functions and function calls the graph will be in the worst case unreadable for the human eye.

There is also a zoom function to zoom out to class level. This way the user is able to see the dependencies between the different classes.

3.2 Graph types

The graphs discussed (call, inheritance, containment and build) are different in respect to each other. The graphs can be categorized in three different types: a tree, a directed acyclic graph and a general (cyclic) graph.

The call graph could be a general graph. It is not a general cyclic graph, because it can have an edge with degree other than 2. A function does not by default call another function.

The inheritance graph could be a directed acyclic graph. It is not possible to inherit the class from the same class. A class inherit another class. The direction of the inherit must be clear in the graph.

The containment graph could be a tree. A function is always part of a class, a class is part of a file, a file is part of a folder.

The build dependency graph could be a general graph. File A can be imported by file B in such a way

that File B has to be rebuild if file A changes. The same applies to the inheriting of classes and the function calls.

3.3 Advantage knowing particular graph structure

The advantage in visualization design of a tree rather than a general graph is that a tree can be created from top to bottom. The structure of a tree is far more clear than the structure of a general graph. A tree is a connected graph and has no cycles. A general graph is a more complex graph then the tree. The advantage of knowing the dependency visualization graph is that the data model can be created according to the graph. A data model for a tree is very different then a data model for a general graph.