

RIJKSUNIVERSITEIT GRONINGEN

RESEARCH PROJECT REPORT

Visualization of the Evolution of Software Quality Metrics on Open Source Projects

Eduardo F. Vernier - s3012875

supervised by
Dr. Alexandru C. TELEA
Renato R.O. da SILVA, M.Sc.
Dr. João L.D. COMBA

July 5, 2016

Contents

1	Introduction	2
1.1	Time-dependent Multivariate Datasets	2
1.2	Research Goal	3
2	Metric Collection	4
2.1	Version Control System	4
2.2	Software Quality Metrics Tools	5
2.3	Metric Extraction	8
2.4	Dataset Filtering and Normalization	10
3	Visualization Techniques	10
3.1	Projection	11
3.2	Glyphs	13
3.3	Treemapping	15
3.4	Colormap	19
3.5	Flow Graph	20
4	Visualization of Google ExoPlayer Project	21
5	Future Work	24
6	Conclusion	26
	References	27

1 Introduction

Data visualization is the presentation of data in graphic format, making it easier for people to understand, compare and extract useful information from collected data. Expressing information through graphics, pictures and animations facilitates the interpretation process in comparison to when the same data is presented in sheets.

When the data dealt with has hundreds of points with tens of attributes that evolve during thousands of time steps, this task becomes even more complex.

On this project, we will deal with the problem of visualizing the evolution of software quality metrics on Open Source projects. The techniques used, challenges faced and the reasoning behind design choices are detailed in this report.

Section 2 discusses how a Version Control System and software quality metric extractor tool are used to generate a time-dependent multivariate dataset. Section 3 details the visualization techniques used to provide insight into the data. Section 4 provides an analysis of the Google ExoPlayer project’s evolution using the aforementioned techniques. Section 5 suggests some possible future directions for the project and Section 6 concludes the report.

1.1 Time-dependent Multivariate Datasets

Imagine that a medical research laboratory is tracking the development of a community of infants in order to get insight on the symptoms of an epidemic disease (e.g. Zika virus). Every week, tests are run and attributes such as weight, blood composition, MRI scan data, among others are collected for each child.

Understanding the evolution and tracking interesting patterns or correlations on such multidimensional dataset for large values of n (number of infants), m (number of collected attributes) and T (number of time moments) is a very challenging task.

Several techniques have been proposed to visualize similarity and change on temporal high-dimensional data. According to Aigner et al. [1], current techniques can be categorized as abstract or spatial, univariate or multivariate, linear or cyclic, instantaneous or interval-based, static or dynamic, and two or three-dimensional.

In this project, we will use a dimensionality reduction technique that provides a scalable alternative to creating projections that evolve smoothly

over time eliminating unnecessary temporal variability [2]. The technique is briefly summarized in section 3.1.

1.2 Research Goal

The main goal of this project is to provide a tool that assists in the task of understanding the evolution of software entities. With such tool, developers and project managers might be able to make more knowledgeable decisions about refactoring, maintenance and delivery of software projects.

To present the tool’s requirements, we will divide it in two: an extractor tool that explores repositories and extract metrics from a given number of revisions; and a visualization tool that takes the generated datasets and present them in a coherent visual manner.

It would be interesting to have at our disposal a technique that given two software entities, returns a numerical value between 0 and 1 telling how similar they are, this way, understanding the dynamics of the implicit structure would be much easier. Given that such technique doesn’t exists, we will use an approach similar to the one used to measure image similarity, in which features extraction is used to estimate the distance (or similarity) between two images [3]. In our context, instead of using techniques such as edge detection, blob detection, template matching and thresholding to build a feature vector, we will use a set of software quality metrics (e.g. cyclomatic complexity, lines of code, lack of cohersion).

The metric extractor tool must be relatively fast and scalable up to hundreds of thousand lines of code, it must be fully automated (i.e. require no user interaction when analysing a new revision) and easy to set up. It must also be able to analyse Java code and produce a reasonable amount of metrics. Java was chosen as the main language because it has a large collection of relevant open source projects developed using it, and for simplicity’s sake, it would be easier to fix one language as a requirement than to make the choice generic.

The visualization tool must provide insight into both the explicit structure of the project, captured by its physical or logical hierarchy and dependencies, and implicit structure, i.e. aspects of the recorded data which create groups of highly-related entities. Therefore it is indispensable that both intra and inter-file metrics be extracted (see Section 2.2).

The visualization tool must provide real time performance, it should be developed using multi-platform technology and contain sensible interaction mechanisms.

Even though software metrics are the main research subject, the tool

is built to accept any time-dependent multivariate dataset, hence, nothing stops it from being effective at analysing the crime rate evolution on a metropolitan area, for example.

2 Metric Collection

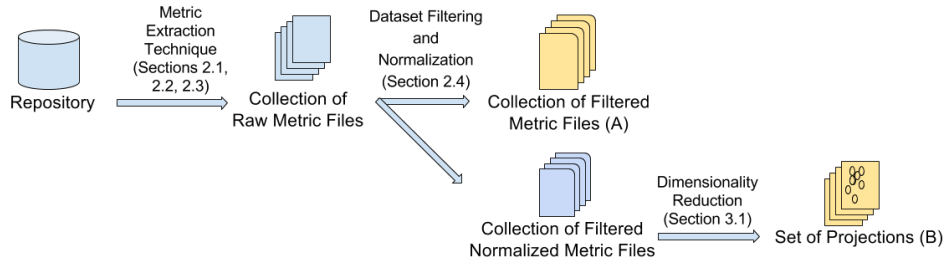


Figure 1: Data extraction pipeline

Figure 1 illustrates the pipeline that takes a software repository and outputs the two datasets (A and B) that are used in a visualization.

The first step to collecting data from repositories is choosing what Version Control Systems to use. There are many options such as CVS, Subversion, Perforce, Bazaar and Git. To choose one over the others, aspects such as popularity for Open Source projects, efficiency and simplicity to “walk” between revisions are crucial. More details on the discussion are presented on section 2.1

The choice of Metric Extraction tool is discussed in section 2.2.

The method developed to take revision files and generate a collection of raw metric tables scalably is featured on Section 2.3.

The need for filtering and normalization of the acquired datasets is explained on Section 2.4.

The dimensionality reduction technique used to generate the projections is presented on Section 3.1.

2.1 Version Control System

According to GitHub co-founder Scott Chacon on the book Pro Git [4], the major difference between Git and any other VCS is the way Git “thinks” about its data. Conceptually, most other systems (e.g. SVN, Mercurial and CVS) store information as a list of file-based changes, meaning that in a source file with a hundred lines of code, if three new lines are added, only

these three lines with their meta data are going to be stored in the new commit, whilst Git stores it as a series of snapshots of the file system, which mean that the hundred and three lines of code are redundantly (even though compressed) saved into the repository when the changes are committed.

But according to the Git Wiki [5], Git is much faster than Subversion since all operations (except for push and fetch) are local and there is no network latency. Git's repositories are also much smaller than Subversions (for the Mozilla project, 30x smaller) and Git repository clones act as full repository backups.

One of the reasons for the smaller repo size is that an SVN working directory always contains two copies of each file: one for the user to actually work with and another hidden in `.svn/` to aid operations such as status, diff and commit. In contrast a Git working directory requires only one small index file that stores about 100 bytes of data per tracked file. On projects with a large number of files this can be a substantial difference in the disk space required per working copy.

Subversion has the advantage of having a simpler way to walk through revision as it uses sequential revision numbers (1,2,3,...), while Git uses unpredictable SHA-1 hashes. Walking backwards in Git is easy using the `^^` syntax, but there is no easy way to walk forward.

Every time a commit is made in Git, the differences are not recorded, instead, it saves all modified files integrally and inserts their references to the commit tree. To be efficient, if files have not changed, they are not redundantly stored in disk, but a link to the previous identical file it has already stored is created. Therefore, Git thinks about its data dynamics as a stream of snapshots. The organization of a repository tree is depicted on Figure 2.

Having clones act as full repository backups with integral files and the performance advantage over SVN were the main reason that lead us to choose Git as VCS. Also the well documented and maintained library libgit2 for C/C++ was important to the choice of VCS.

2.2 Software Quality Metrics Tools

Our requirements for the Software Quality Metrics Extractor Tool regarded the quality and relevance of the metrics, strictness of input acceptance, i.e how it dealt with unbuildable code and missing references, project activity, overall performance and ease of set-up and integration. It is important for the understanding of the project dynamics to extract both intra-file metrics, i.e. those which can be computed simply by looking at a single file

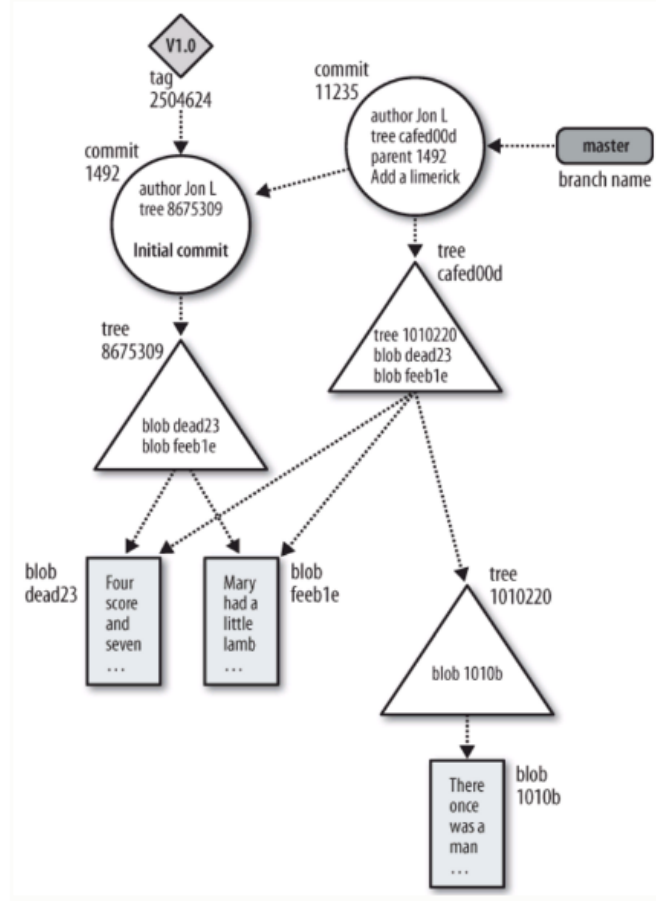


Figure 2: Git tree organization (image from <https://www.hackerearth.com/>)

in isolation (e.g. LOC, average method complexity, average class cohesion) and inter-file metrics, which are metrics that look at the relations between entries in different files (e.g. depth of a class hierarchy, average function-call-path length, metrics on the number of uses of a given symbol). The later are significantly more complex to compute.

Eight tools were tested in the metric extractor tools analysis phase of the study, out of which, four - CCCC, SourceMeter, iPlasma and CppCheck - were swiftly discarded due to design or performance misalignment with our goals. The remaining four tools - SonarQube, Analizo, CodePro Analytix and SciTools Understand - were subjected to a series of tests that would grade the suitability of each for our purposes.

SonarQube is a very professional tool dedicated to continuous analysis and measurement of source code. It has complicated initial set-up and proved to be hard to integrate in our software. The quality of its outputted metrics wasn't on par with the other three tools and its performance wasn't satisfactory either. Analizo, differently from the other three tools, is an academic project. It is easy to use and outputs a good amount of relevant metrics. Unfortunately, for complicated projects with many external or missing references, it struggled, taking over 30 minutes to analyse medium size repositories, rendering it unsuitable for our study.

CodePro Analytix and SciTools Understand fit all our requirements. Both are easy to use and integrate, flexible regarding the repositories they take as input, customizable in the parameterization of the metrics, fast, well-documented professional grade tools. Both output a good set of inter and intra-file metrics in various levels of granularity and in easy to work formats (xml and csv). We chose to work with Understand over Analytix in reason of the performance advantage one has over the other and the number of class level metrics they output (43 for Understand against Analytix's 17). The full metric reference document provided by SciTool Understand is available at [6].

Table 1 shows the performance comparison between the two tools for nine Open Source projects.

Table 1: Understand and Analytix analysis time for a single revision

Project (KLOC)	Analytix Time(s)	Understand Time(s)
JMeter (118)	23	30
Checkstyle (95)	32	32
Gitblit (77)	31	20
JUnit (26)	17	10
JavaGame (3)	10	4
Netty (194)	70	66
Guava (243)	120	168
Zxing (42)	20	11
MPAndroidChart (20)	14	8

Table 2 shows a ranking from 0 to 5 related to requirements measured on all tools. Some cells have dashes instead of values because the tool was considerate unsuitable before the attribute was measured.

Table 2: Tool Comparison

Tool	Well main- tained	Ease to set up	Easy to integrate	Tolerance of input acceptance	Metric Quality	Perfor- mance
CCCC	0	0	-	0	-	-
Source Meter	3	0	-	-	-	1
iPlasma	-	4	0	-	1	2
CppCheck	-	-	-	-	0	-
SonarQube	5	1	3	4	2	2
Analizo	3	3	3	2	5	2
Analytix	5	4	5	5	5	5
Understand	5	4	5	5	5	5

2.3 Metric Extraction

Once we have settled on Git and Scitools Understand as VCS and metric collector, we built a tool that takes as argument a repository URL (e.g. <https://github.com/GNOME/gimp.git>) or a path to an already checked-out on disk repository and outputs a collection of files that represent the metric values for a set of commits. The number n of revisions to be extracted from the repository is also given as an argument along with an optional $T_{start} - T_{end}$ interval. The JMeter project, for example, has currently 12,647 commits. If we input 100 as the number of revisions, the first to be extracted will be the last committed revision, after that, we walk 126 steps back on the commit tree and check-out the next one. The process continues consecutively for the remaining revisions.

There are two possible approaches to perform the metric extraction. The first would be to create n directories and for each of them, check-out all source files from the i th revision. This is the easiest method but is very slow and space inefficient, as files from previous commits that have not undergone changes must be decompressed and written to disk unnecessarily.

The second approach was the one we chose to work with. First, check-out every Java source file from the last revision to a single “dump” directory, disregarding the original directory structure. Considering Git keeps all it’s past files compressed in the .git folder, this process is very fast and requires no online access to the repository — this is crucial, because the operation is repeated several times.

With all files checked-out in the dump directory, all alphanumeric SHA-1 file and directory keys (which work as identifiers) listed in this commit are

added to a Trie data structure. Understand then runs it's metric calculations and outputs an appropriately identified file to the *metric* directory.

When the metric calculation is complete, it takes the next selected commit and, comparing it's file/directory keys to the current state on the Trie tree, checks which files or groups of files are already loaded into disk. It also checks which of the files that are currently checked into disk are not present in the next revision and deletes them. The Trie tree is then updated for the current revision. The quality metrics are then extracted and this step is repeated for the remaining $n - 1$ revisions.

The *metric* directory for the JMeter project with 20 analysed revisions looks like Figure 3 where each file is CSV formatted collection of 43 attributes.






















Name	Size	Type
 JMeter.0.data	262.9 kB	Text
 JMeter.1.data	262.1 kB	Text
 JMeter.2.data	262.9 kB	Text
 JMeter.3.data	262.8 kB	Text
 JMeter.4.data	262.8 kB	Text
 JMeter.5.data	262.8 kB	Text
 JMeter.6.data	262.8 kB	Text
 JMeter.7.data	262.8 kB	Text
 JMeter.8.data	262.8 kB	Text
 JMeter.9.data	262.7 kB	Text
 JMeter.10.data	262.7 kB	Text
 JMeter.11.data	262.7 kB	Text
 JMeter.12.data	262.7 kB	Text
 JMeter.13.data	262.7 kB	Text
 JMeter.14.data	262.7 kB	Text
 JMeter.15.data	262.7 kB	Text
 JMeter.16.data	262.7 kB	Text
 JMeter.17.data	262.8 kB	Text
 JMeter.18.data	262.8 kB	Text
 JMeter.19.data	262.8 kB	Text
 JMeter.20.data	262.7 kB	Text

Figure 3: Metric output file set

More details and source code can be found at <https://github.com/EduardoVernier/metric-extractor>. The tool was implemented in C++ with the Qt framework.

2.4 Dataset Filtering and Normalization

Before the datasets are ready for the visualization two last steps are necessary. The first concerns the unsuitability of the dimensionality reduction technique to deal with dynamic datasets, i.e. collections where members are created or deleted between consecutive time moments. What this implies is that all classes that we are analysing must be present from the first selected commit up until the last one, which means that a considerable percentage of the project’s classes must be filtered from the dataset. As far as we know, no multidimensional projection technique fits this requirement.

The second step is normalization. This is necessary because some metric range from 0 to 1 (e.g comments ratio) and others don’t have a specific delimited range (e.g lines of code), therefore the dimensionality reduction technique will associated different weights to different metric value changes (which is not ideal). To perform the data normalization, we must first extract the average value and standard deviation of each attribute. Then for each sample, we subtract the corresponding average and divide the result by the standard deviation. What this guarantees is that the average for each feature is zero and the standard deviation through time is one.

3 Visualization Techniques

A metrics dataset (illustrated as A on Figure 1) is a set of revisions R where each revision $r_t \in [r_s, r_e]$ is a set of entities $E = \{e_i\} \subset \mathbb{R}^n$ that portray a feature vector of $n = 43$ real-valued quality metrics of a given software class.

For each revision $r_t \in [r_s, r_e]$, dt-SNE is used to generate a 2D projection $P_t = \{q_i\} \subset \mathbb{R}^2$ where i is the number of observations and q a point in 2D space. The projection set P depicts the class similarity throughout the project’s history and is illustrated as B on Figure 1. More details on the technique are present on Section 3.1.

Once we have collected the two time-dependent datasets, we must develop a tool to visualize it. We propose three linked views as means to get insight on the dynamics of the data. The first is the Projection View described in section 3.1 associated with custom designed Glyphs detailed in section 3.2.

The second view consist of a color mapped Treemap, discussed in sections 3.3 and 3.4.

The third has not yet been implemented, but it's concept is presented on section 3.5.

The final tool should be laid out as illustrated on Figure 4.

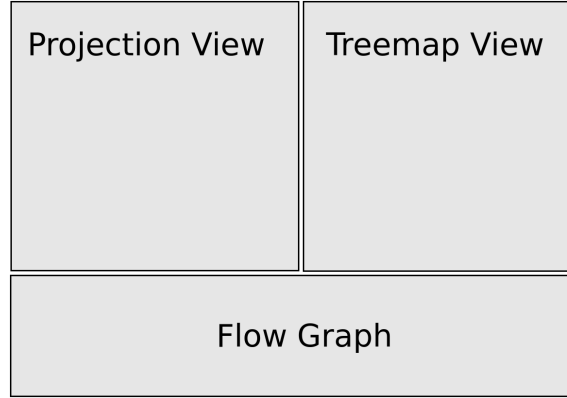


Figure 4: Tool Mock-up

The tool is available at <https://github.com/EduardoVernier/metric-view>.

3.1 Projection

Multidimensional projection (MP) techniques aim to map high-dimensional data to a p -dimensional visual space of lower order trying to preserve the original distances as much as possible. MP methods bear the intrinsic ability of generating visual representations that naturally convey the similarity relationship between instances of data.

When compared to other high-dimensional visualization techniques such as table lenses [7], evolution lines [8], evolution matrices [9], parallel coordinates [10], and scatterplot matrices [11], multidimensional projections are considerably more scalable in number of entities and dimensions it can handle, it is also specially easy to find groups of related entities.

However, MPs don't explain why a given set of entities belong to the same group. Recently, Da Silva et al. have been working to address this problem on [12] and [13].

Also, the vast majority of MPs do not handle time-dependent datasets. The recent dynamic t-SNE (dt-SNE) [2] technique is, to our knowledge, the only MP for time-dependent datasets that offers explicit and verified

guarantees in terms of spatial and temporal coherence. Trade-off between preservation of distances in the same projection vs preservation of distances across projections which are close in time is controlled by a user parameter.

Figure 5 was extracted from Rauber *et al.* [2] and illustrates on the left how the original t-SNE technique doesn't take into consideration the previous position of a group of entities and might place it in a distant locale, whilst the dt-SNE technique on the right tries to preserve the previous neighborhood layout.

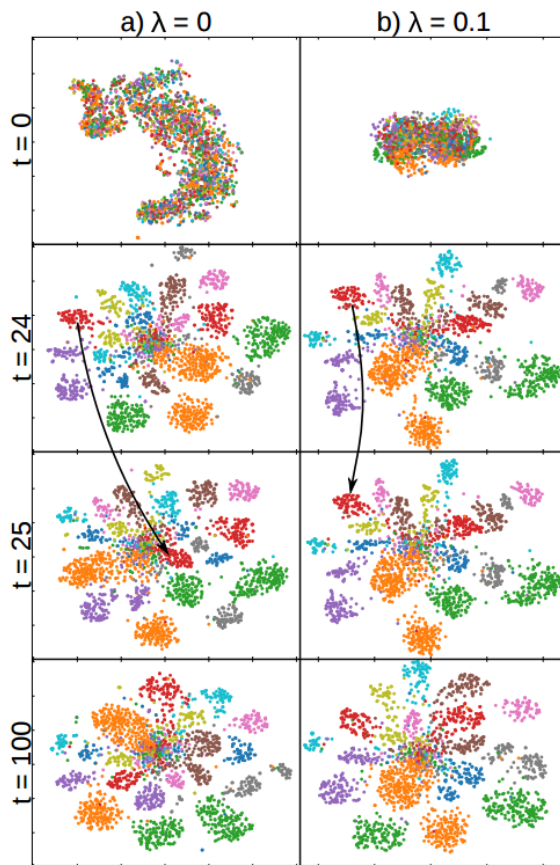


Figure 5: Dynamic t-SNE results on SVHN CNN

For each revision $r_t \in [r_s, r_e]$, dt-SNE is used to generate a 2D projection $P_t = \{q_i\} \subset \mathbb{R}^2$ such that the pairwise distance between points $\|q_i - q_j\|$ are as close as possible to $\|e_i - e_j\|$ and the distances between the same point in consecutive revisions E_{t1} and E_{t2} are preserved. This P projection set is

then visualized in the Projection View.

3.2 Glyphs

Once a projection P_t is laid out on the projection view, we instinctively start to identify clusters. But given only the point positions, it is impossible to determine which attributes might have brought them together (i.e. how are points similar to one another).

Our current approach involves adding information to these projection points q_i using glyphs and colormapping. Hence, if we select a metric such as Lines of Code, for example, we might see that some groups have similar metric values. Tracking these points’ movement dynamics (splits and merges) along with the evolution of the metric values (encoded in shape, radius and color) might provide interesting insight into the development history of the project. But note that this is a manual approach.

Da Silva *et al.* [13] have proposed an interesting set of techniques to understand which attributes might have promoted grouping of entities. They use attribute ranking to determine which are the most significant dimensions to dictate a point’s q_i resemblance to it’s neighbors. These “explaining metrics” are then used to label and color relevant clusters in the projection, as depicted on Figure 6.

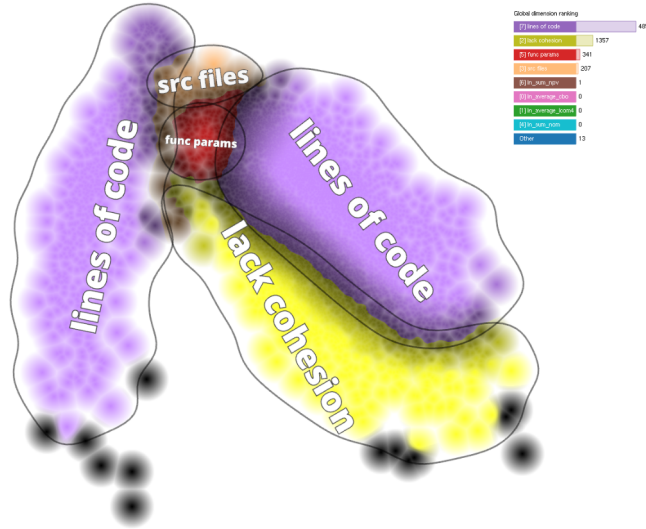


Figure 6: Identification of a cluster’s “explaining metric” using labeling and qualitative colormaps with variable confidence

We find this insight very valuable to our visualization, and even though this technique is not yet integrated on our tool, we plan to introduce the feature in the near future.

Currently, in order to add information to the projection points we use glyphs. The only restriction imposed is that no extra color should be used in the glyph’s design, as we will later color the glyphs with a number of colormaps (see Section 3.4) and we don’t want to cause “interference” on the color channel, therefore all information illustrated must be depicted in the glyph’s shape.

In Figure 7, the glyph’s radius represents the normalized value of the metric in the current revision, and the angle of the pie section amounts to the percentual change in metric value from revision T_{n-1} to T_n . If the metric increases in value, a pie section of larger radius is added to the top of the glyph, otherwise a pie section of smaller radius is placed on the bottom of the glyph representing the reduction percentage.

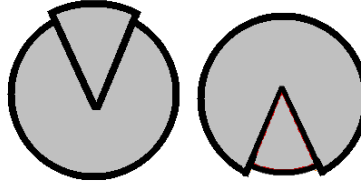


Figure 7: Glyphs portraying 20% increase and reduction on metric value

Although it’s an interesting design, it has one major flaw: 80% increase and 20% decrease in value are represented by the same form. This issue is fixed on the glyph shown on Figure 8, where the decrease is portrayed by the removal of a pie section.

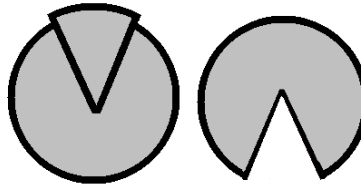


Figure 8: Revised glyph

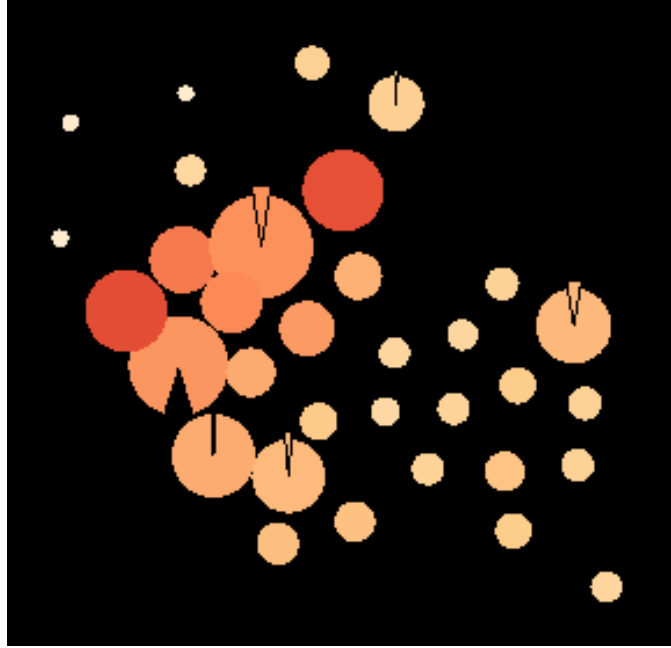


Figure 9: Set of entities with glyph's shape, radius and color representing the current value and change for the Lines of Code metric.

In the application, the pie section is only drawn if the change in metric value is higher than 1%, as shown in Figure 9.

3.3 Treemapping

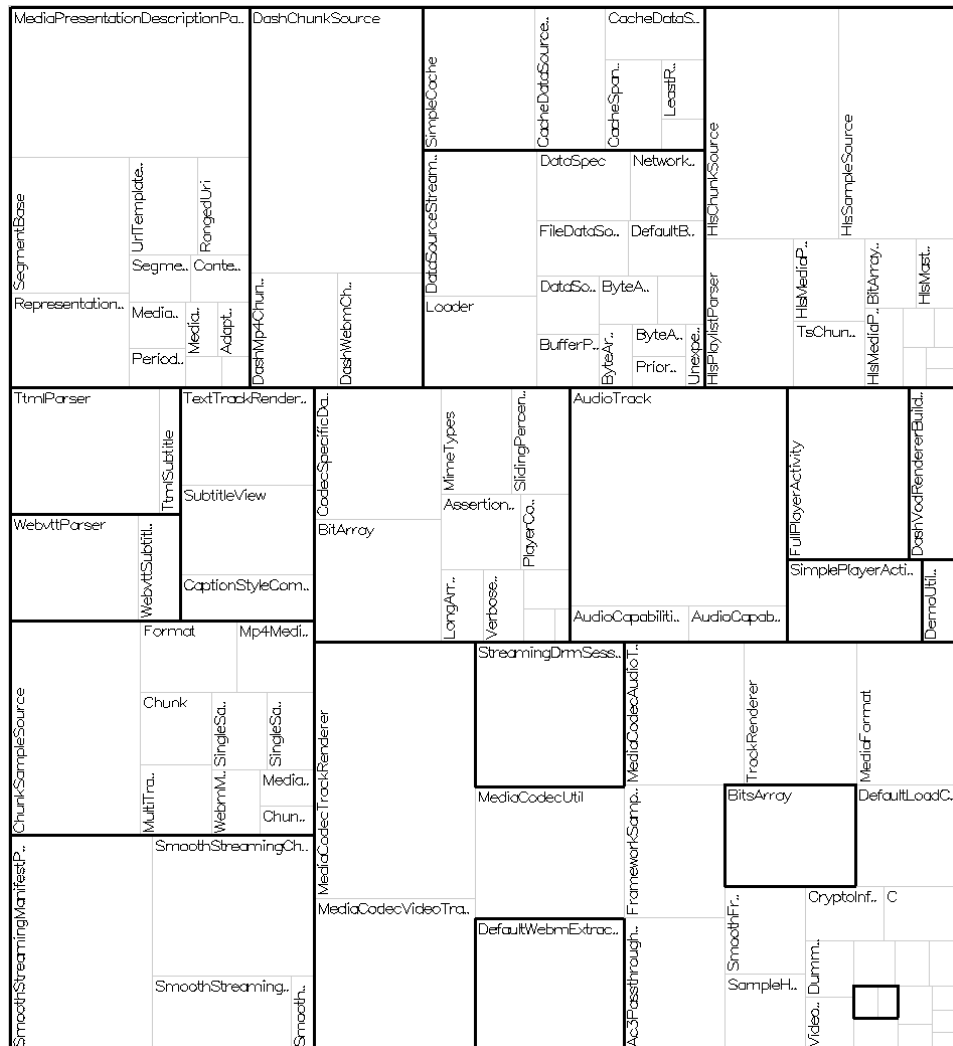
When thinking about software entities, it is very natural to think of them in a hierarchical manner. This hierarchy tends not only to refer to the organization and architecture of a project, but also to the function and behavior of specific entities.

But visualizing hierarchy on top of a projection is not a trivial task, that's why, for this initial approach, we decided to use a second linked view containing a Treemap.

A Treemap is a method to display hierarchical data using a collection of nested rectangles. On this particular project, the nesting of rectangles depicts the package/class hierarchy on the project. The area of each rectangle is defined by the number of lines of code of each class on the last analysed revision. The area of a package frame is defined by the sum of all classes' areas belonging to it or to its children packages.

There are several tiling algorithms in the literature with different characteristics (e.g. Ordered Treemaps [14], Slice and Dice Treemaps [14], Quantum Treemaps [15]). In addition, several algorithms have been proposed that use non-rectangular regions [16–18].

But all these techniques offer a trade-off between maintaining the input order and keeping low aspect-ratio. In our project, the algorithm used to plot these maps is called Squarified Treemaps [19]. It prioritizes low aspect-ratios over ordering. The results of this technique on two Open Source projects can be found on Figures 10 and 11.



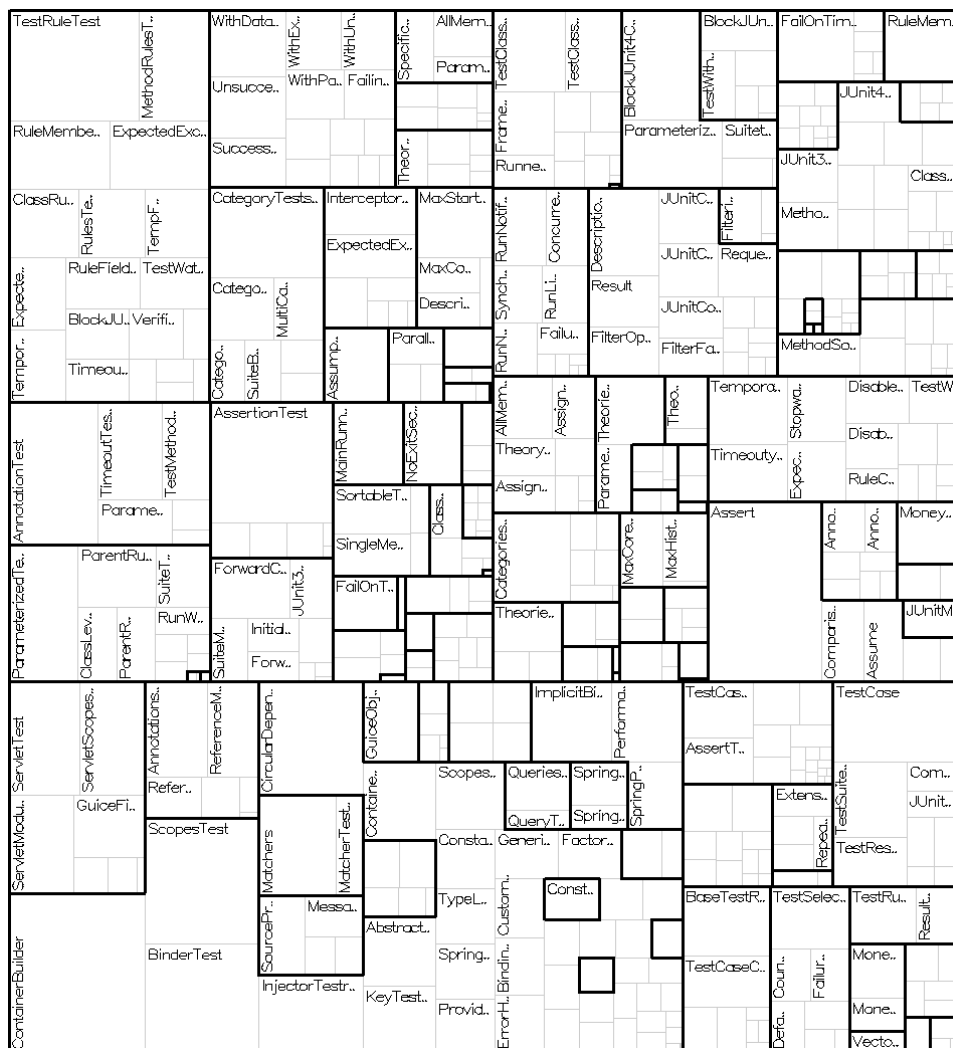


Figure 11: Google Guice 529 classes treemap

We can use the colormapping technique discussed in Section 3.4 to get interesting insight into the class entities. On Figure 12, we used the Sequential Colormap to illustrate the Comments to Code Ratio metric on the last revision. It is possible to see that the biggest classes, and arguably the most complex ones, have some of the lowest comment ratio. This might be troublesome when time comes to perform maintenance in these classes.

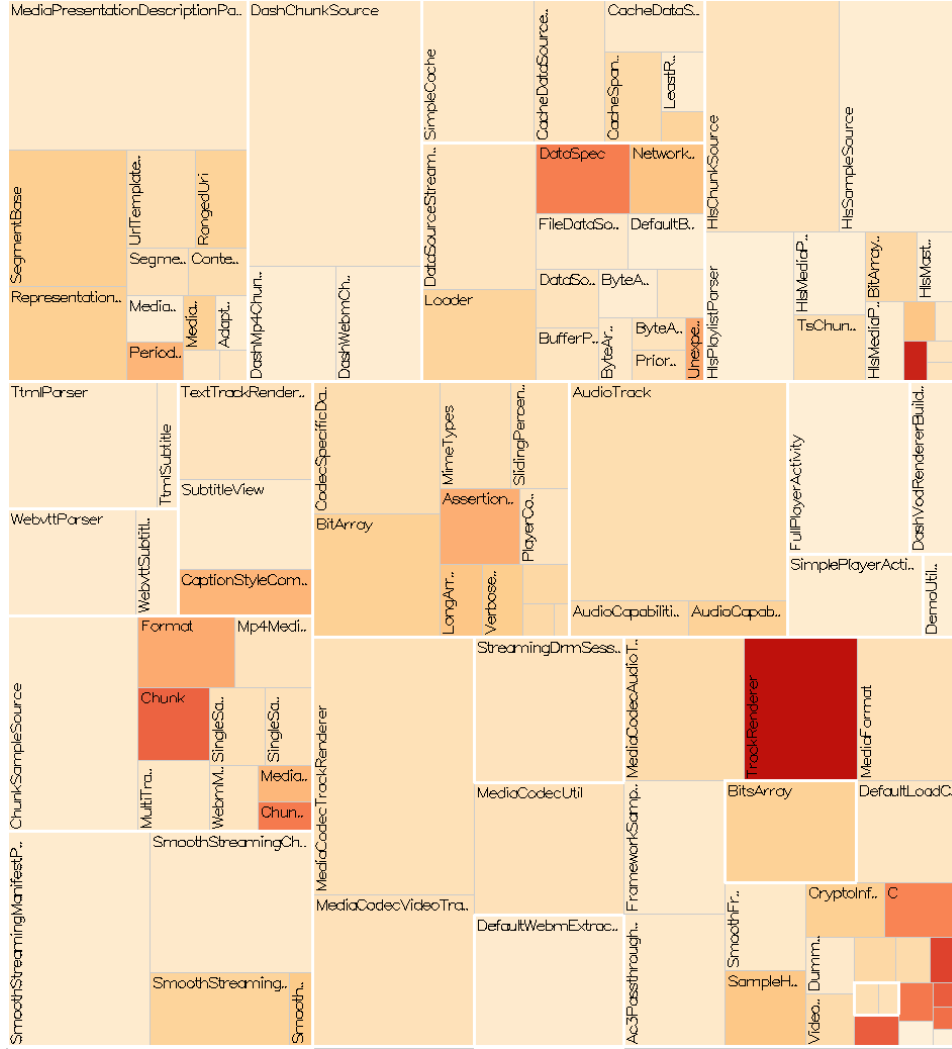


Figure 12: ExoPlayer Treemap with Comment Ratio metric

3.4 Colormap

In this tool, three different color mapping schemes were implemented. A sequential colormap was used to display the current normalized value of a chosen metric, a diverging colormap was used to show the increase/decrease of a given metric from revision T_{n-1} to T_n , and a categorical colormap was used to group classes from the same package into a single color.

All three colormaps were based on the samples presented on ColorBrewer

[20].

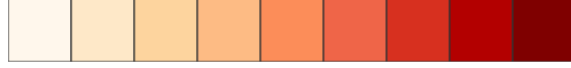


Figure 13: Sequential Colormap



Figure 14: Diverging Colormap



Figure 15: Qualitative Colormap

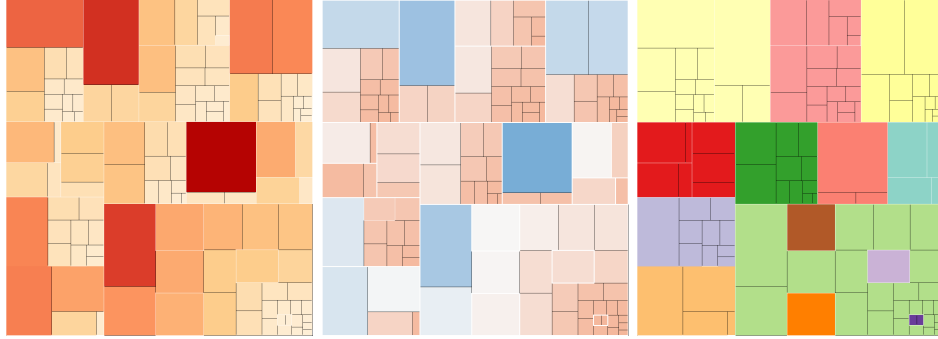


Figure 16: Color mapping on Google ExoPlayer project

The qualitative colormap in particular has room for improvement, it would be interesting to sort the packages by size before assigning a color, in this way, the likelihood of neighbor packages having the same color would be reduced.

3.5 Flow Graph

The third view of the application is a Flow Graph. Using the already implemented selection mechanisms, the user may specify a group of classes

along with a chosen metric to understand it’s evolution throughout all the project’s history.

This in itself is a very insightful technique used in other software evolution understanding tools. Figure 17 was extracted from SolidTA [7] and depicts the distribution of files for the Lines of Code metric on the Gimp project. Our approach would be different in the sense that the entities would be fixed and the metric values would vary.

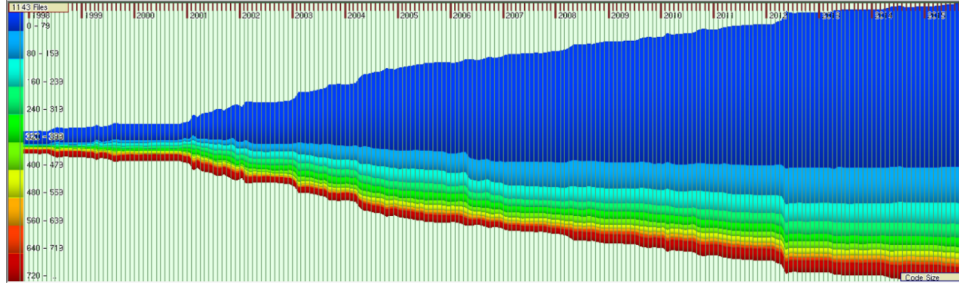


Figure 17: SolidTA Flow Graph

4 Visualization of Google ExoPlayer Project

In this section, the developed tool is used to explore the evolution of Google ExoPlayer for Android. The project currently has over 46,000 lines of code, but as discussed in Section 2.4, some classes must be discarded, leaving an average of 27,000 lines of code analysed per revision distributed over 117 classes. The period on which this analysis was run over ranges from December 2014 to February 2016. The 1609 commits that compose the project were reduced to a set of 100 revisions.

With this analysis we are trying to find interesting phenomena in the development cycle of the project, some examples might be refactoring, implementation of new features, redesign in order to accommodate for a larger user base, crucial bug fixes, among others as we are also interested in discovering the unknown.

We start with a brief overview of the life-cycle. The user is presented with a window in which he can select which metrics among the 43 he wants to illustrate on the glyph’s shape and color, or if he would rather present the hierarchical information using a qualitative colormap (as shown on Figure 16). This settings are free to change during the visualization, allowing for a interactive detailing and exploration of the dataset. The user may also

explore different time moments using the Z and X keys to advance or retreat in the history.

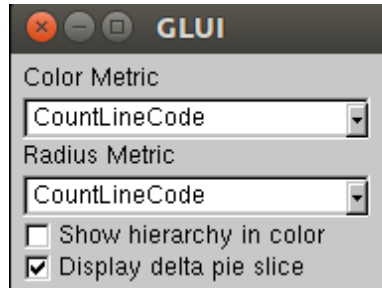


Figure 18: Setting configuration window

With the Lines of Code metric selected for both glyph radius and colormap, whilst searching for interesting patterns between revisions, it is possible to notice a significant metric value drop from T_{26} (Figure 19) to T_{27} (Figure 20).

Using the divergent colormap as in Figure 21, it is possible to identify the classes which were most affected by this possible code refactoring process.

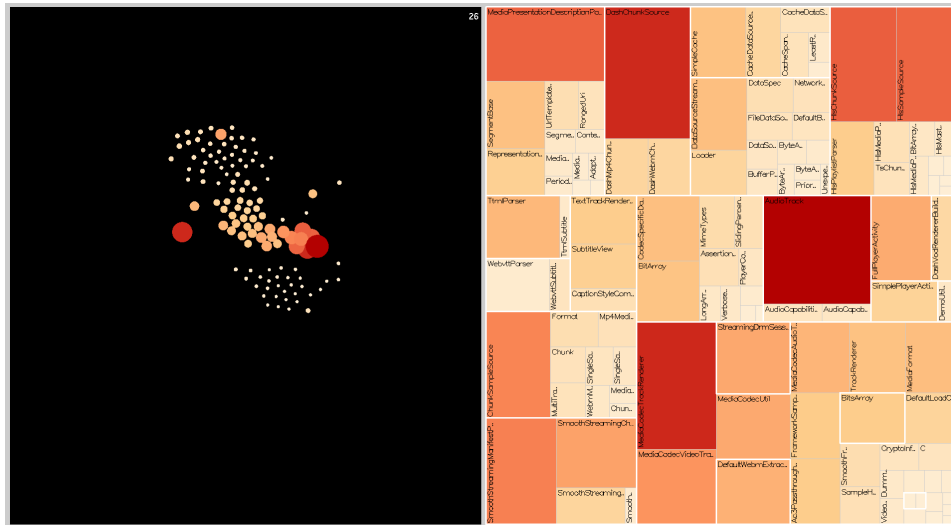


Figure 19: Revision T_{26}



Figure 20: Revision T_{27}

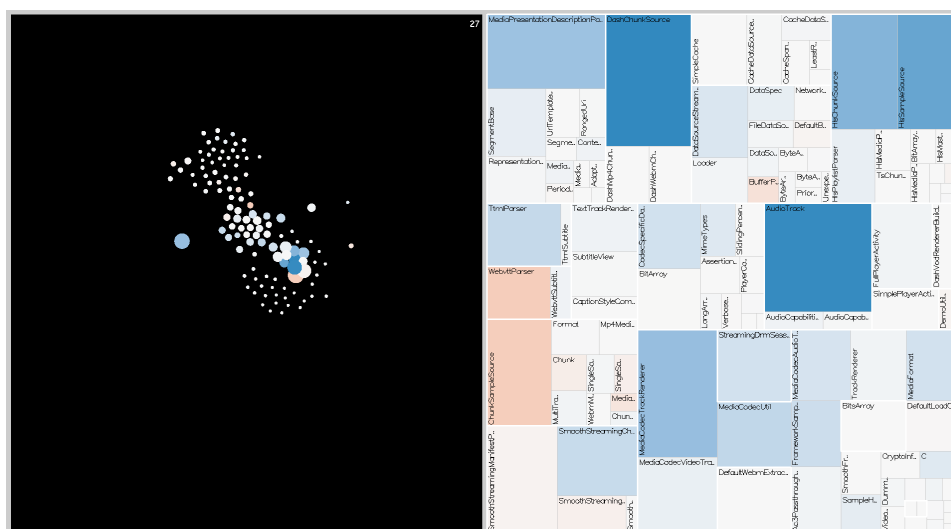


Figure 21: Revision T_{27} using divergent colormap

One possible explanation for this drop might be a refactoring period. Using the Treemap view we can see the distribution of this change on the project's hierarchy.

We can then select a set of the classes that went under the highest size

reduction and see how they relate and evolve throughout the project history. Figures 22 to 25 show these selected entities and the global evolution pattern. Interestingly, as time goes by, the the projection points tend to separate from their neighbors, which could mean that the project source is becoming more heterogeneous.

5 Future Work

There are many improvements that can be added to the current tool, starting by the Flow Graph proposed in Section 3.5, which would not only show the evolution of a given metric throughout the whole history of the project, but also present what are the actual (and not only relative) metric values. Also, as already proposed in Section 3.2, the integration of Da Silva *et al.*'s techniques would be very beneficial to the understanding of why do entities relate to each other.

Another interesting feature that would offer great insight would be to list the commit messages that are in the interval of two time steps T_{n-1} and T_n . That would allow a better understanding of **why** the metrics have changed. A new feature implementation or merge of a branch could explain an abrupt increase in the value of some metrics. A code refactoring period or the addition of a library that does part the work that was previously assigned to individual classes might express the reduction of a set of metrics. Having these messages at hand would remove the factor “I think they did this...” from the reasoning.

Additionally, a diff tool such as the one GitHub has (see Figure 26) would help get into code level details of what has happened between revisions and who was the developer responsible for it. Hence, the visualization methods we proposed would help pinpoint when and where the real changes have happened, then the user himself would be able to check the code changes that caused relevant metric behavior (e.g. redesign of an important data structure).

Lastly, it would be interesting to be able to choose between the dt-SNE projection and a simple two dimensional scatter plot, in which each axis is an independent metric. This would be a more traditional way of understanding up to four metric dynamics.

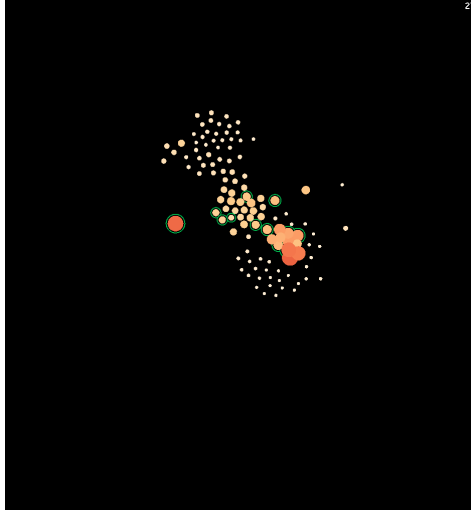


Figure 22: Projection on T_{27}

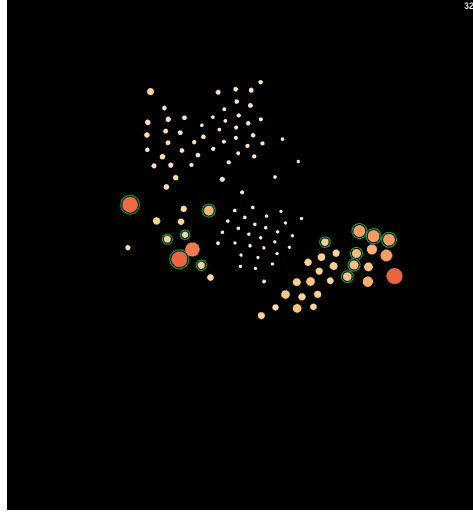


Figure 23: Projection on T_{32}

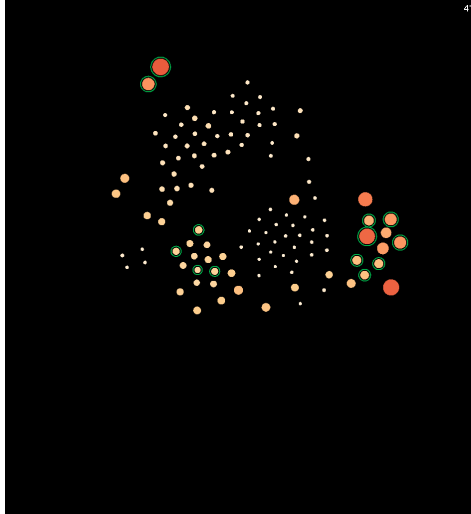


Figure 24: Projection on T_{41}

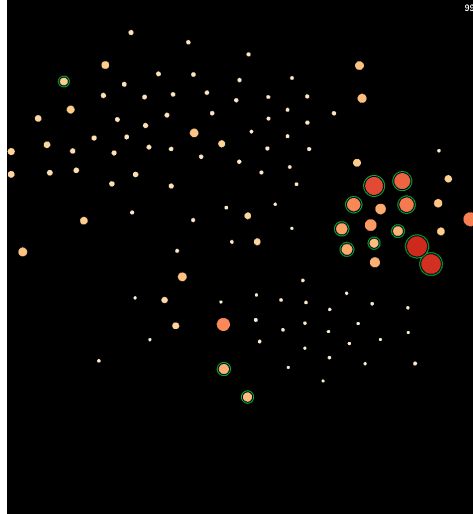


Figure 25: Projection on T_{99}

21	{	23	{
22	vector<BaseEntity*> dataCopy;	24	+ // Make a copy of data, as the original is destroyed during treemapSingledimensional computation
23	for (vector<BaseEntity*>::iterator b = data->begin() ; b != data->end(); ++b)	25	vector<BaseEntity*> dataCopy;
24	dataCopy.push_back(*b);	26	for (vector<BaseEntity*>::iterator b = data->begin() ; b != data->end(); ++b)
25		27	dataCopy.push_back(*b);
26	treemapSingledimensional(data, width, height, xOffset, yOffset);	28	
27		29	+ // Compute single level treemap
28	for (vector<BaseEntity*>::iterator b = dataCopy.begin() ; b != dataCopy.end(); ++b)	30	treemapSingledimensional(data, width, height, xOffset, yOffset);
29	{	31	
30	if ((*b)->isPackage())	32	+ // For every package from data computed above, compute treemap for their BaseEntity children
31	{	33	for (vector<BaseEntity*>::iterator b = dataCopy.begin() ; b != dataCopy.end(); ++b)
32	vector<BaseEntity*> newData;	34	{
33	((Package*)(*b))->sortEntities();	35	if ((*b)->isPackage())
34	- for (vector<BaseEntity*>::iterator child = ((Package*)(*b))->sortedEntities.begin(); child != ((Package*)(*b))->sortedEntities.end(); ++child)	36	{
35	{	37	vector<BaseEntity*> newData;
36	newData.push_back(*child);	38	((Package*)(*b))->sortEntities();
37	}	39	+ for (vector<BaseEntity*>::iterator child = ((Package*)(*b))->sortedEntities.begin(); child != ((Package*)(*b))->sortedEntities.end(); ++child)
38	-	40	{
39	- double newWidth = (*b)->coords[2] - (*b)->coords[0];	41	newData.push_back(*child);
40	- double newHeight = (*b)->coords[3] - (*b)->coords[1];	42	}
41	- double newxOff = (*b)->coords[0];	43	+ // Calculate new containers dimentions
42	- double newyOff = (*b)->coords[1];	44	+ double *coords = (*b)->getCoords();
43	treemapMultidimensional(&newData, newWidth, newHeight, newxOff, newyOff);	45	+ double newWidth = coords[2] - coords[0];
44	}	46	+ double newHeight = coords[3] - coords[1];
45	}	47	+ double newxOff = coords[0];
46	}	48	+ double newyOff = coords[1];
47		49	+ // Recursive call
48	-	50	treemapMultidimensional(&newData, newWidth, newHeight, newxOff, newyOff);
49	void Treemap::treemapSingledimensional(vector<BaseEntity*> *data, double width, double height, double xOffset, double yOffset)	51	}
		52	}
		53	}
		54	
		55	+// Wrapper for squarity algorithm
		56	void Treemap::treemapSingledimensional(vector<BaseEntity*> *data, double width, double height, double xOffset, double yOffset)

Figure 26: GitHub Diff Tool

6 Conclusion

At the current state of the project we are able to tell which entities relate to each other and how they evolve over time. However, this information alone is not sufficient to make very interesting claims or predict patterns on big Open Source projects. We need to understand why such behaviors take place on the projection and treemap views in order to investigate them and point what are their causes.

The implementation of the proposed future work would allow us to perform in-depth user studies from a top-down approach — starting from the very conceptual projection view down to code level. This might bring very interesting insight on the entity dynamics of software projects and lead to positive impact on their evolution.

References

- [1] W. Aigner, S. Miksch, H. Schumann, and C. Tominski, *Visualization of Time-Oriented Data*. Springer-Verlag London, 2011.
- [2] P. Rauber, A. Falcão, and A. Telea, “Visualizing time-dependent data using dynamic t-sne,” in *Proceedings EuroVis Short Papers*, 2016.
- [3] L. Wang, Y. Zhang, and J. Feng, “On the euclidean distance of images,” *IEEE Transactions on Pattern Analysis and Machine Intelligence* 2005.
- [4] S. Chacon, *Pro Git*. Apress, 2009.
- [5] S. Pearce, “Gitsvncomparison,” 2016 (accessed 2016-7-3). <https://git.wiki.kernel.org/index.php/GitSvnComparison>.
- [6] SciTool, “Metric implementation notes,” 2016 (accessed 2016-7-3). <https://scitools.com/documents/metricImplementationNotes.pdf>.
- [7] D. Reniers, L. Voinea, O. Ersoy, and A. Telea, “The solid* toolset for software visual analytics of program structure and metrics comprehension: From research prototype to product,” *Science of computer programming*, vol. 79, pp. 224–240, 1 2014. Relation: <http://www.rug.nl/research/jbi/> Rights: University of Groningen, Johann Bernoulli Institute for Mathematics and Computer Science.
- [8] L. Voinea, A. Telea, and J. J. van Wijk, “Cvsscan: Visualization of code evolution,” pp. 47–56, 2005.
- [9] M. Lanza, “The evolution matrix: Recovering software evolution using software visualization techniques,” pp. 37–42, 2001.
- [10] A. Inselberg and B. Dimsdale, “Parallel coordinates: A tool for visualizing multi-dimensional geometry,” in *Proceedings of the 1st Conference on Visualization '90, VIS '90*, (Los Alamitos, CA, USA), pp. 361–378, IEEE Computer Society Press, 1990.
- [11] J. Hartigan, “Printer graphics for clustering,” *Journal of Statistical Computation and Simulation*, vol. 4, no. 3, pp. 187–213, 1975.
- [12] R. R. O. da Silva, P. Rauber, R. Martins, R. Minghim, and A. Telea, “Attribute-based visual explanation of multidimensional projections,” in *Proceedings of the 2015 EuroVis Workshop on Visual Analytics*.

- [13] R. R. O. da Silva, A. Telea, E. Vernier, P. Rauber, J. Comba, and R. Minghim, “Metric evolution maps: Multidimensional attribute-driven exploration of software repositories,” in *Proceedings of the Vision, Modeling and Visualization 2016*.
- [14] B. Shneiderman and M. Wattenberg, “Ordered treemap layouts,” pp. 73–, 2001.
- [15] B. B. Bederson, B. Shneiderman, and M. Wattenberg, “Ordered and quantum treemaps: Making effective use of 2d space to display hierarchies,” *ACM Trans. Graph.*, vol. 21, pp. 833–854, Oct. 2002.
- [16] D. Auber, C. Huet, A. Lambert, B. Renoust, A. Sallaberry, and A. Saulnier, “Gospermap: Using a gosper curve for laying out hierarchical data,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 19, no. 11, pp. 1820–1832, 2013.
- [17] M. Balzer and O. Deussen, “Voronoi treemaps,” pp. 7–, 2005.
- [18] K. Onak and A. Sidiropoulos, “Circular partitions with applications to visualization and embeddings,” pp. 28–37, 2008.
- [19] M. Bruls, K. Huizing, and J. van Wijk, “Squarified treemaps,” *Data Visualization 2000: Proc. Joint Eurographics and IEEE TCVG Symp. on Visualization*, 2000.
- [20] C. Brewer, “Color brewer.” <http://colorbrewer2.org/>, 2016. Accessed: 2016-06-25.