

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

EDUARDO FACCIN VERNIER

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

Work presented in partial fulfillment
of the requirements for the degree of
Bachelor in Computer Science

Advisor: Prof. Dr. Joao L.D. Comba
Coadvisor: Prof. Dr. Alexandru C. Telea

Porto Alegre
December 2016

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Reitor: Prof. Rui Vicente Oppermann

Vice-Reitora: Prof^a. Jane Fraga Tutikian

Pró-Reitor de Graduação: Prof. Sérgio Roberto Kieling Franco

Diretora do Instituto de Informática: Prof^a. Carla Maria Dal Sasso Freitas

Coordenador do Curso de Ciência de Computação: Prof. Carlos Arthur Lang Lisbôa

Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

ABSTRACT

Computer scientists have done an incredible job at creating rich visualizations for fields such as engineering, chemistry, physics, and medicine. Yet, in modern software development, it is very hard to find visualization tools at use in design, implementation, maintenance, or testing of code. This work attempts to provide a set of visualization techniques that facilitate the understanding of the evolution of software entities and their relationships, supplying professionals with interesting insight about the development process and product.

Keywords: Software quality metrics. Metric extraction. Dimensionality reduction. Hierarchical data. Evolution.

Visualizando a evolução de métricas de qualidade de software em projetos Open-Source

RESUMO

Cientistas da computação têm feito um ótimo trabalho na criação de ricas visualizações para disciplinas como engenharias, química, física e medicina. Entretanto, no processo de desenvolvimento de software moderno, é raro encontrar ferramentas de visualização em uso no design, implementação, manutenção ou teste de código. Este trabalho tenta prover um conjunto de técnicas que visam facilitar a compreensão de entidades de software e seus relacionamentos, permitindo maior discernimento do produto e processo de desenvolvimento de software.

Palavras-chave: Métricas de qualidade de software. Redução de dimensionalidade. Dados hierárquicos. Evolução.

LIST OF FIGURES

Figure 3.1 Data extraction pipeline	15
Figure 3.2 The variable in the graph represents the global search interest relative to the highest point on the chart through time. A value of 100 is the peak popularity for the term. A value of 50 means that the term is half as popular as it once was.	16
Figure 3.3 Git tree organization (image from https://www.hackerearth.com/)	17
Figure 3.4 Metric output file set.....	21
Figure 4.1 van Wijk questions: “Botanic visualization contents of a hard disk. Useful or just a nice picture?”	24
Figure 4.2 Treemap layout comparison.....	26
Figure 4.3 Google ExoPlayer 117 classes treemap	26
Figure 4.4 Google Guice 529 classes treemap	27
Figure 4.5 Base rectangle area represents maximum LOC value and filled rectangle area encode current’s	28
Figure 4.6 Representation of hierarchy using Sunburst Diagram	28
Figure 4.7 Representation of the ExoPlayer’s project hierarchy using both Sunburst Diagram and Squarified Treemap	29
Figure 4.8 Sequential Colormap	29
Figure 4.9 Diverging Colormap	29
Figure 4.10 Qualitative Colormap.....	29
Figure 4.11	30
Figure 4.12	31
Figure 4.13 ExoPlayer Treemap with number of methods per class metric	32
Figure 4.14 Dynamic t-SNE results on SVHN CNN.....	34
Figure 4.15 LOC metric displayed in both radius and color attributes	35
Figure 4.16 LOC metric displayed in both radius and color attributes	36
Figure 4.17 Glyphs portraying 20% increase and reduction on metric value	36
Figure 4.18 Revised glyph	36
Figure 4.19 Set of entities with glyph’s shape, radius and color representing the current value and change for the Lines of Code metric.	37

Figure 4.20 Evolution of the file count on the GIMP project color coded by file size using SolidTA	38
Figure 4.21 Evolution of the LOC metric for 10 classes using the sequential colormap	38
Figure 4.22 Evolution of the LOC metric for 10 classes using the divergent colormap.	38
Figure 4.23 Filtered evolution of the LOC metric on ExoPlayer from Deceber 2014 to February 2016.....	39
Figure 4.24 Filtered evolution of the LOC metric on the RxJava project from De- cember 2013 to February 2016	39
Figure 4.25 Filtered evolution of the LOC metric on the Google Closure project from August 2011 to February 2016.....	39
Figure 4.26 Filtered evolution of the LOC metric on the Eclipse Vert.x project from January 2014 to February 2016.....	40
Figure 4.27 Filtered evolution of the LOC metric on ExoPlayer from Deceber 2014 to February 2016.....	40
Figure 4.28 Filtered evolution of the PercentLackOfCohesion metric on the Exo- Player project.	41
Figure 4.29 Filtered evolution of the PercentLackOfCohesion metric on the RxJava project.	41
Figure 4.30 Linking nodes hierarchically on a synthetic projection.....	43
Figure 4.31 Linking nodes hierarchically on the ExoPlayer project.....	43
Figure 4.32 Linking filtered packages hierarchically on the ExoPlayer project.....	44
Figure 4.33 Linking nodes by similarity on the ExoPlayer project	45
Figure 4.34 Visualization of a software system's call graph with edge bundling	45

LIST OF TABLES

Table 3.1 Understand and Analytix analysis time for a single revision.....	19
Table 3.2 Tool Comparison: Some cells have dashes instead of values because the tool was considerate unsuitable before the attribute was measured.....	20

LIST OF ABBREVIATIONS AND ACRONYMS

SMP	Symmetric Multi-Processor
NUMA	Non-Uniform Memory Access
SIMD	Single Instruction Multiple Data
SPMD	Single Program Multiple Data
ABNT	Associação Brasileira de Normas Técnicas

CONTENTS

LIST OF FIGURES	5
LIST OF TABLES	7
1 INTRODUCTION.....	10
1.1 Data	11
1.2 Research Goal.....	12
2 RELATED WORK	14
3 METRIC COLLECTION	15
3.1 Version Control System	15
3.2 Software Quality Metrics Extraction Tools	18
3.3 Metric Extraction.....	20
3.4 Dataset Filtering and Normalization.....	22
4 VISUALIZATION.....	23
4.1 Hierarchical Data.....	24
4.1.1 Squarified Treemap	25
4.1.2 Sunburst Diagram	28
4.2 Colormap	29
4.3 Projection.....	32
4.3.1 Glyphs	35
4.4 Visualizing Evolution.....	37
4.5 Linking the Views.....	41
REFERENCES.....	46
REFERENCES.....	46

1 INTRODUCTION

Modern software projects are incredibly large and complex systems, specially considering that they evolve in time.

Let us use the Eclipse IDE as an example. According to the Eclipse Neon press release, “This is the eleventh year the Eclipse community has shipped a coordinated release of multiple Eclipse projects. There are 84 projects in the Neon release, consisting of over 69 million lines of code, with contribution by 779 developers, 331 of whom are Eclipse committers” (FOUNDATION, 2016 (accessed 2016-11-9)). Another good example is the GNU Image Manipulation Program, that has been actively under development for over 20 years (TEAM, 2016 (accessed 2016-11-9)).

In this work, we will enter the realms of the Information Visualization field, and, more specifically, the Software Visualization branch. “Software visualization is concerned with the static or animated 2D or 3D visual representation of information about software systems based on their structure, history, or behavior in order to help software engineering tasks” (DIEHL, 2007).

These tasks can be performed at various levels helping to increase productivity and code quality. For instance, programmers might use these techniques in order to assist in the understanding of large code bases. Testers might find them useful to help guiding their efforts more efficiently. Architects might use visualization to compare the code being produced with the original design. Project managers and business leaders might use them in order to have an overview of long-term projects or access the quality of process and product decision.

Our goal with this project is assisting users in the understanding the evolution of software projects. Our work attempts to provide effective visualization techniques that help revealing the truth hidden behind mountains of abstract time dependent data.

According to Spence, the three fundamental ingredients for any infovis application are representation, presentation, and interaction (SPENCE, 2007). In this text, we will explain how we shaped these elements to our context in order to make it easier for users to understand, compare, correlate, and extract useful insight from software projects. The techniques used, challenges faced and the reasoning behind design choices are also detailed in this text.

Data

The Scientific Visualization field focuses on data that has physical placement. For instance, when we see the 3D rendered results of an MRI scan of a brain, the data presented can be very naturally understood, as we already have a mental map of what a brain looks like. Software source code, conversely, has no inherent physical or spacial placement. Therefore, we are faced with the added challenge of introducing appropriate visual representation for such data and relations. Additionally, the information encoded in software source files cannot be naturally mapped to a traditional scivis dataset model usually consisting of a region in \mathbb{R}^n sampled in multiple points that can have its values interpolated in this space.

On this project, we will model evolving software projects as multivariate, or multidimensional, time-dependent datasets. For each data point on a multidimensional dataset, also called a record, sample, observation, or instance, we can measure many properties of the underlying phenomenon, each of these being a different variable, attribute, or dimension. Thus, a multidimensional dataset can be thought of as a table of n rows (or observations) and m columns. When m is larger than roughly 5, this data is considered to be high-dimensional .

What is meant by time-dependent, is that our dataset is composed of a number t of observations in time, each of these being an individual multidimensional dataset.

To exemplify, imagine that a medical research laboratory is tracking the development of a community of a hundred infants in order to get insight on the symptoms of an epidemic disease (e.g. Zika virus). During the first year of infancy, tests are run weekly and ten attributes such as weight, blood composition, MRI scan data, cognitive test results, among others are collected for each child.

Understanding the evolution and tracking interesting patterns or correlations on such multidimensional time-dependent 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. (AIGNER et al., 2011), 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 (RAUBER; FALCAO; TELEA, 2016). The technique is summarized in section 4.3.

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, professionals that work with software both in the academia and industry might develop a better understanding of software dynamics, and, through visualization techniques, uncover previously unknown aspects of code bases or even confirm hypothesis, being then able to make more knowledgeable decisions about refactoring, maintenance, delivery of software projects, among others.

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 (WANG; ZHANG; FENG,). 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 cohercion) as an expression of the information present on the software entities.

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 analyzing a new revision) and easy to set up. It must also be able to analyze Java code and produce a reasonable amount of metrics. Java was chosen as the main language because of the large collection of relevant active open source projects that use it, and in order to accommodate for metric extractor tools' limitations as well as simplify the data representation, we chose to fix it as the only supported language.

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 metric extraction are supported by the extractor tool (see Section 3.2).

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

2 RELATED WORK

3 METRIC COLLECTION

Software Quality Metrics are measurements that relate to properties of a piece of software or its specifications. They can generate useful insight in various levels of granularity (e.g. project level, package level, class level, method level) and can lead to a better understanding of the code base. These insights might influence several decisions, ranging from where to place refactoring and performance optimization effort to budget planning, optimal personnel task assignments, and cost estimation.

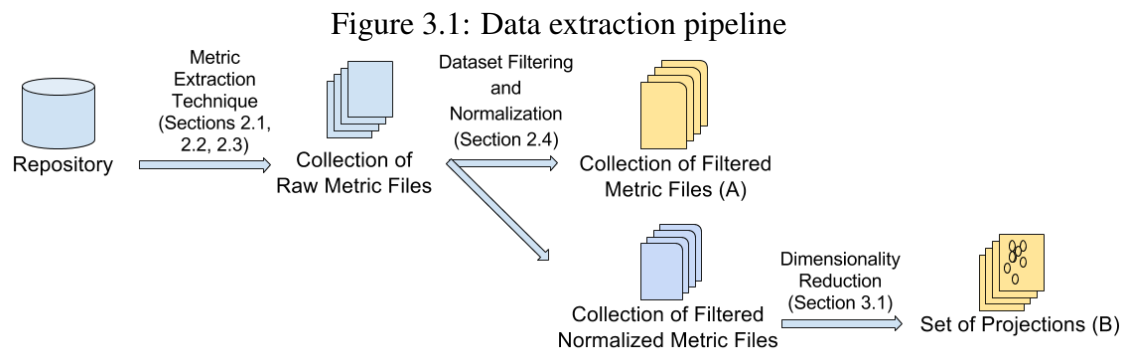


Figure 3.1 illustrates the pipeline that takes a software repository and outputs the two datasets (A and B) that are used in our visualization tool.

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, Mercurial, Bazaar and Git. To choose one over the others, aspects such as popularity for Open Source projects, efficiency in the use of resources, performance, and simplicity to “walk” between revisions, are taken into consideration. More details on the discussion are presented on section 3.1

The choice of Metric Extraction tool is discussed in section 3.2.

The method developed to take revision files and generate a collection of raw metric tables scalably is featured on Section 3.3.

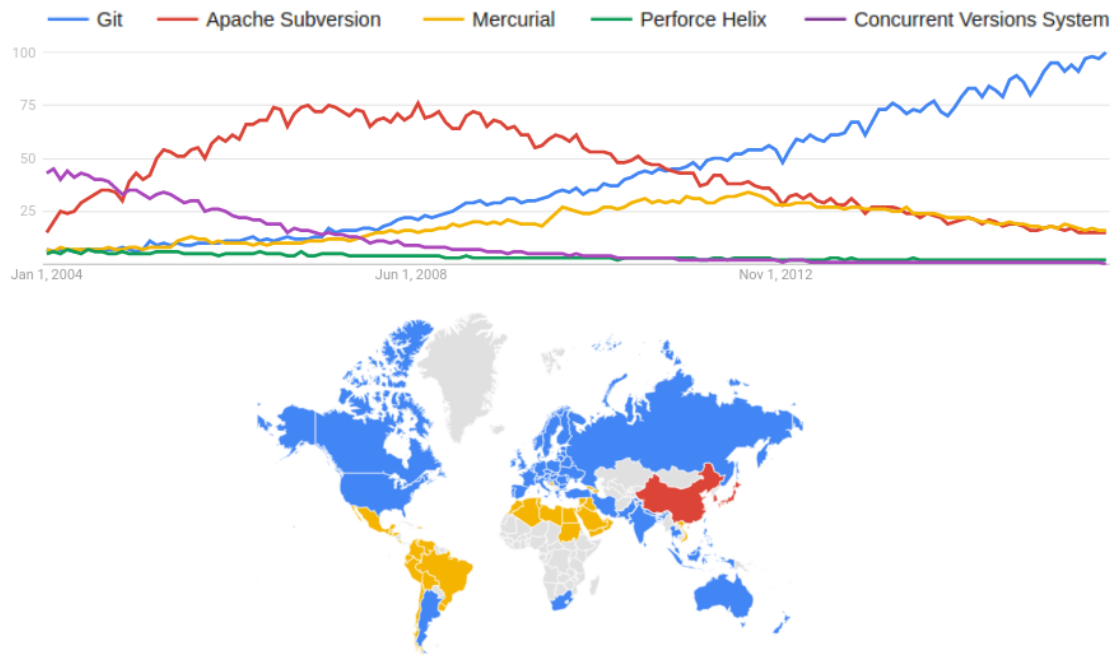
The need for filtering and normalization of the acquired datasets is explained on Section 3.4.

Version Control System

In order to estimate VCS popularity we will use Google Trends data collected from January 2004 to November 2016 on the most used systems according to the 2015

Stack Overflow Developers Survey (OVERFLOW, 2016 (accessed 2016-11-9)). Google Trends allows comparison between the number of Google searches related to a particular system relative to the total search volume. A world map also points out which is the most popular system in each country.

Figure 3.2: The variable in the graph represents the global search interest relative to the highest point on the chart through time. A value of 100 is the peak popularity for the term. A value of 50 means that the term is half as popular as it once was.



Based on this data, we can confirm that Git is currently the most popular VCS and that its popularity is steadily increasing.

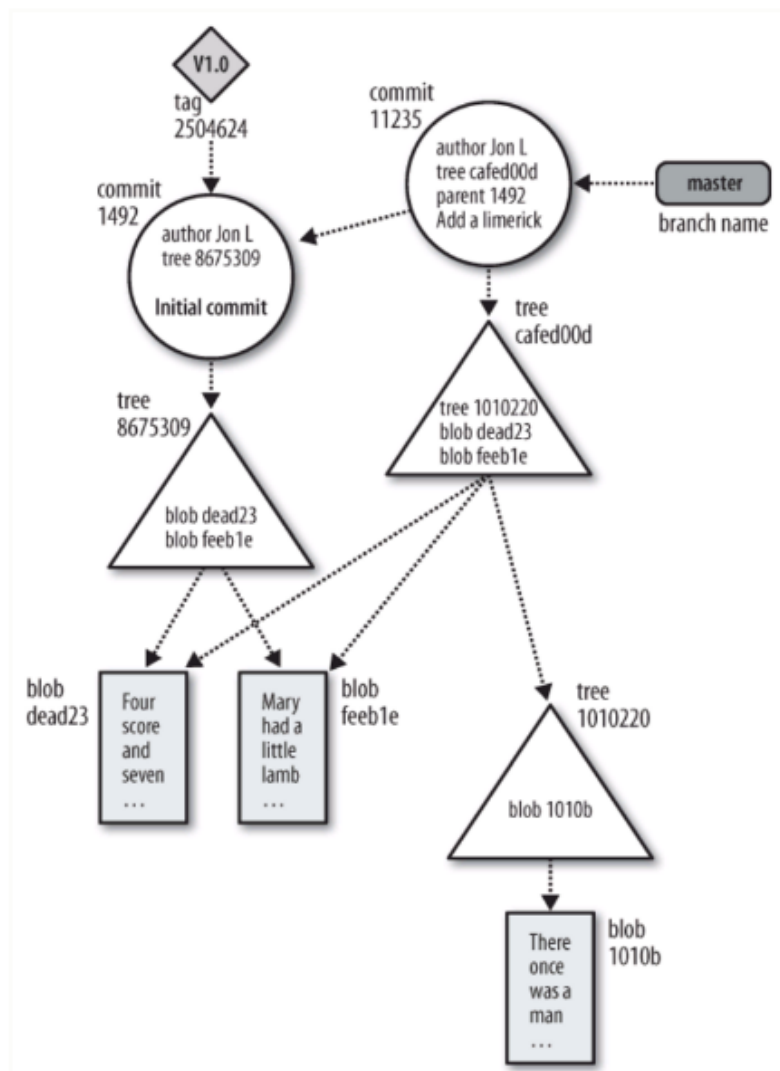
According to GitHub co-founder Scott Chacon on the book *Pro Git* (CHACON, 2009), the major difference between Git and any other VCS is the way Git “thinks” about its data. Conceptually, most other systems (e.g. SVN, Mercurial and CVS) store information as a list of file-based changes, meaning that in 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 revision. Git, instead, stores its data 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.

Despite that, according to the Git Wiki (PEARCE, 2016 (accessed 2016-7-3)), 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 repository 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 the `.svn/` folder 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.

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 3.3.

Figure 3.3: Git tree organization (image from <https://www.hackerearth.com/>)



Subversion has the advantage of having a simpler way to navigate between re-

visions as it uses sequential revision identifiers (1,2,3,...), instead of Git's unpredictable SHA-1 hashes.

Considering the popularity factor, full local repository backups with integral files, better performance and use of resources, we deemed Git as being a more suitable system for our context. Additionally, the well documented and maintained library *libgit2* for C/C++ was fundamental for the choice of VCS.

Software Quality Metrics Extraction Tools

Our requirements for the Software Quality Metrics Extractor Tool regarded the quality and relevance of the metrics, strictness of input acceptance (e.g. 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 in isolation (e.g. number of lines of code, average method complexity, average class cohesion) and inter-file metrics, which are metrics that look at the relations between entities 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 latter 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 analyze 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 (SCITOOL, 2016 (accessed 2016-7-3)).

Table 3.1 shows the performance comparison between the two tools for nine Open Source projects.

Table 3.1: Understand and Analytix analysis time for a single revision

<i>Project (KLOC)</i>	<i>Analytix Time(s)</i>	<i>Understand Time(s)</i>
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 3.2 shows a score from 0 to 5 related to requirements measured on each tool. In order to fit all data in one table, we labeled the requirements in the following fashion:

<i>Requirement</i>	<i>Label</i>
Well maintained	A
Ease to set up	B
Easy to integrate	C
Tolerance of input acceptance	D
Metric Quality	E
Performance	F

Table 3.2: Tool Comparison: Some cells have dashes instead of values because the tool was considerate unsuitable before the attribute was measured.

<i>Tool</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>	<i>E</i>	<i>F</i>
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	4
Understand	5	4	5	5	5	5

Metric Extraction

Once we had 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/google/ExoPlayer.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 revisions. 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.

We thought of two approaches to perform 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 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 its past files compressed in the .git folder, this process is very fast and requires no online access to the repository — this is crucial, considering 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 3.4 where each file is CSV formatted collection of 43 attributes.

Figure 3.4: Metric output file set

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

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.

Dataset Filtering and Normalization

Before the datasets are ready for the visualization two last steps are necessary. The first concerns the unsuitability of current 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 analyzing 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, allowing unbiased projection calculations.

4 VISUALIZATION

Discussions that relate image perception and cognitive understanding can be traced as far back as the early Greek philosophers. Socrates considered that sensory experiences create images in the human mind, regarded as mental representations of the real world. Around 350BC, Aristoteles stated that “thought is impossible without an image”.

Gershon (1994) defines visualization as follows: “Visualization is more than a method of computing. Visualization is the process of transforming information into a visual form, enabling users to observe the information. The resulting visual display enables the scientist or engineer to perceive visually features which are hidden in the data but nevertheless are needed for data exploration and analysis.”

The resulting visual representation can be used to both to confirm the known, (i.e. validating the fit a given model with a dataset) and discovering the unknown (i.e. creating insight that supports a new model in the data). But first, we need first to recognize when visualization is actually useful. As questioned by Telea, if a question can be answered by a compact, precise query (e.g. what is most complex class in a given package), why visualize? Or if a decision can be automated, why put a human in the process? (e.g. decide which classes are considered *dead code* and should be removed from a project.)

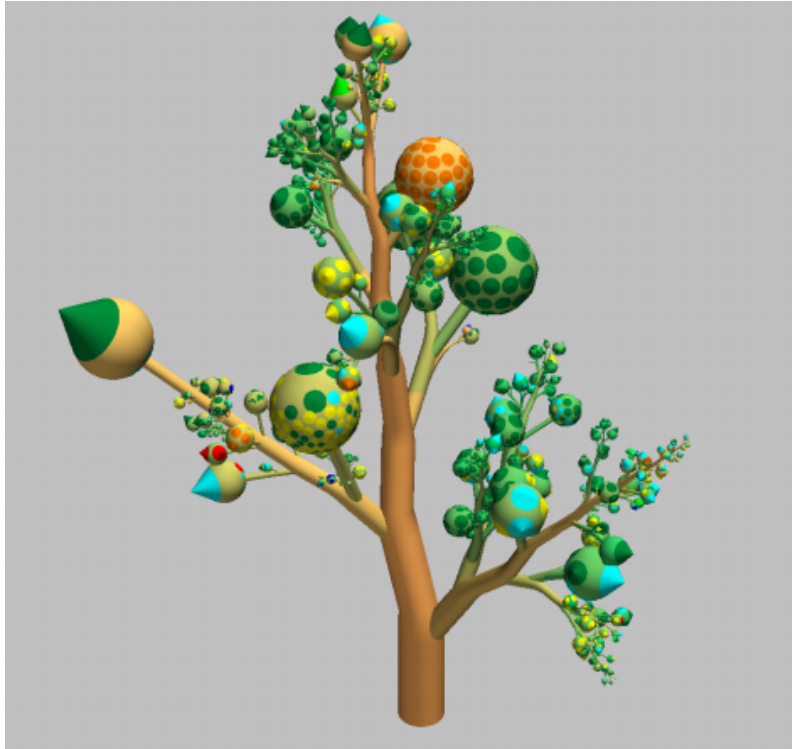
Visualization be can useful in many applications. For example, when there’s too much data and we don’t have time to analyze or make sense of it all, allowing for a overview of the data’s features. It is useful to get insight about distribution, correlations, behavior, and relationships. It can also be useful to answer qualitative and complex questions. Lastly, it is fundamental for communication.

The challenge of creating a good visualization lies in tailoring visual representations that objectively communicate features of the data. When our representations are not suitable, as recognized by van Vijk on figure 4.1, visualization might not be very helpful. Bad representations might also lead to a visual result that is in misalignment with the original data, and as stated by Edward Tufte, “clutter and confusion are failures of design, not attributes of information”.

The dataset we are trying to understand (illustrated as A on Figure 3.1) is a set of R revisions 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 $r_t \in [r_s, r_e]$, a dimensionality reduction technique is used to generate a 2D projection $P_t = \{q_i\} \subset \mathbb{R}^2$ where i is the number of observations and

Figure 4.1: van Wijk questions: “Botanic visualization contents of a hard disk. Useful or just a nice picture?”



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 3.1. More details on the technique are present on Section 4.3.

Once we have collected the two time-dependent datasets, we must develop a tool to visualize it. In this chapter we will discuss our attempts in transforming this complex data into visual forms that allow for insightful answers. To develop these techniques no libraries were used apart from OpenGL for the rendering and GLUT/GLUI for the window/GUI management. The source code is available at <https://github.com/EduardoVernier/metric-view> and from now on, we will address the visualization tool as MetricView.

Hierarchical Data

When thinking about a software project, it is very natural to think of it 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.

As we mentioned on Section 1.2, one of our objectives is to represent the hierarchical relationship in the data. There are many traditional techniques that portray this

sort of relationship, each of those with its pros and cons. A few honorable mentions are Squarified and Voronoi Treemaps, Icicle Trees, Sunburst Diagrams, and Circle Packing.

In this application, we have implemented both Squarified Treemap and Sunburst Diagram, each with a different task in mind. Squarified Treemaps are better at using “real state”, while Sunburst Diagrams give a intuitive and unambiguous representation of deep level hierarchies.

Squarified Treemap

A Treemap is a method to display hierarchical data traditionally 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 maximum number of lines of code each class has had during the project’s history. The area of a package frame is defined by the sum of all classes’ areas belonging to it or to it’s children packages.

There are several tiling algorithms in the literature with different characteristics (e.g. Ordered Treemaps (SHNEIDERMAN; WATTENBERG, 2001), Slice and Dice Treemaps(SHNEIDERMAN; WATTENBERG, 2001), Quantum Treemaps(BEDERSON; SHNEIDERMAN; WATTENBERG, 2002)). In addition, several algorithms have been proposed that use non-rectangular regions (AUBER et al., 2013; BALZER; DEUSSEN, 2005; ONAK; SIDIROPOULOS, 2008).

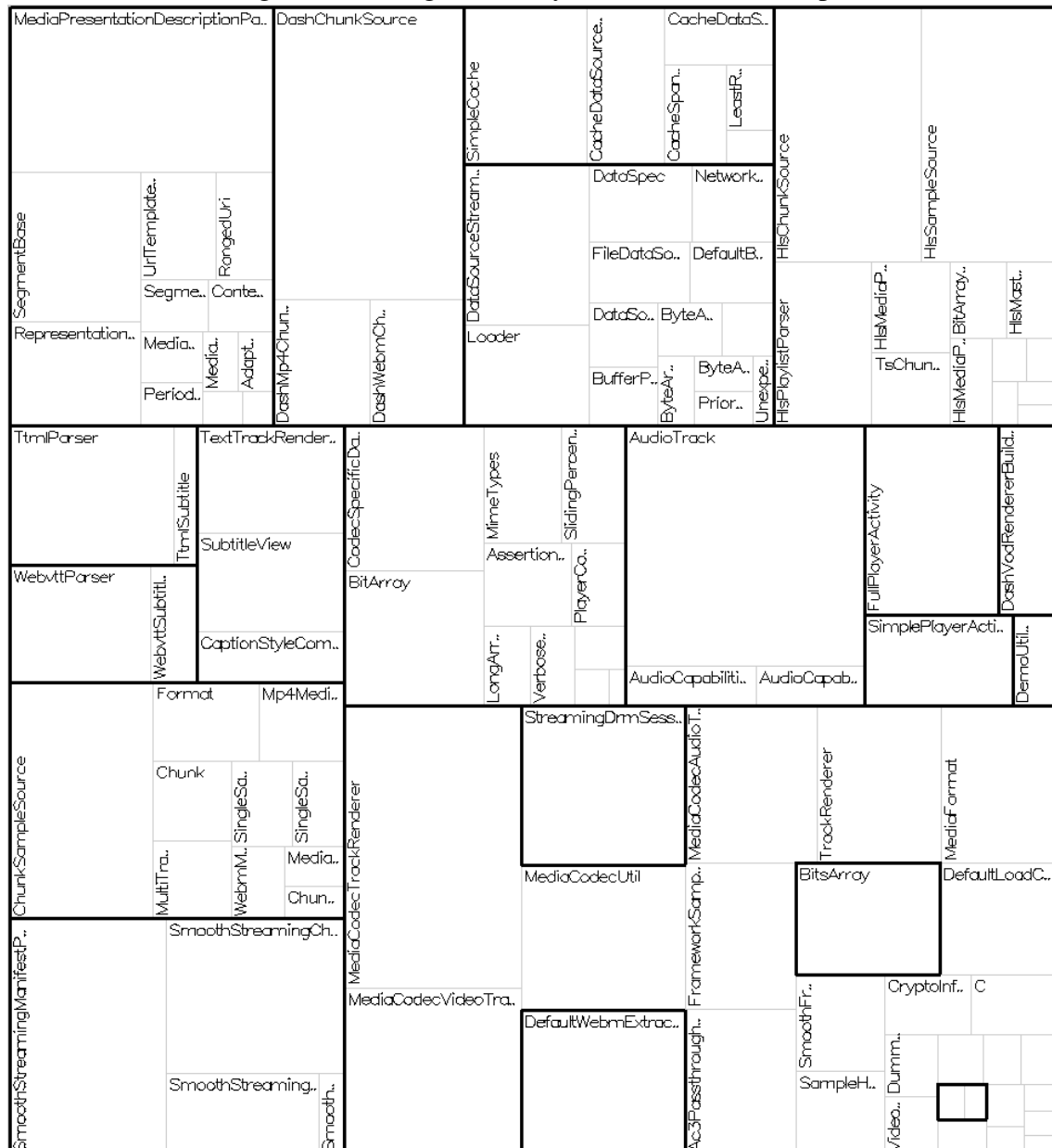
But all these techniques offer a trade-off between maintaining the input order and keeping a low aspect-ratio. In our project, the algorithm used to plot these maps is called Squarified Treemaps(BRULS; HUIZING; WIJK, 2000). It prioritizes low aspect-ratios over ordering, reducing the amount of thin, elongated rectangles (see images 4.2a and 4.2b). The results generated by our tool on two Open Source projects can be found on Figures 4.3 and 4.4.

(a) Stock portfolio with traditional slice-and-dice layout.

(b) Stock portfolio with squarified layout.

(a) Stock portfolio with traditional slice-and-dice layout.

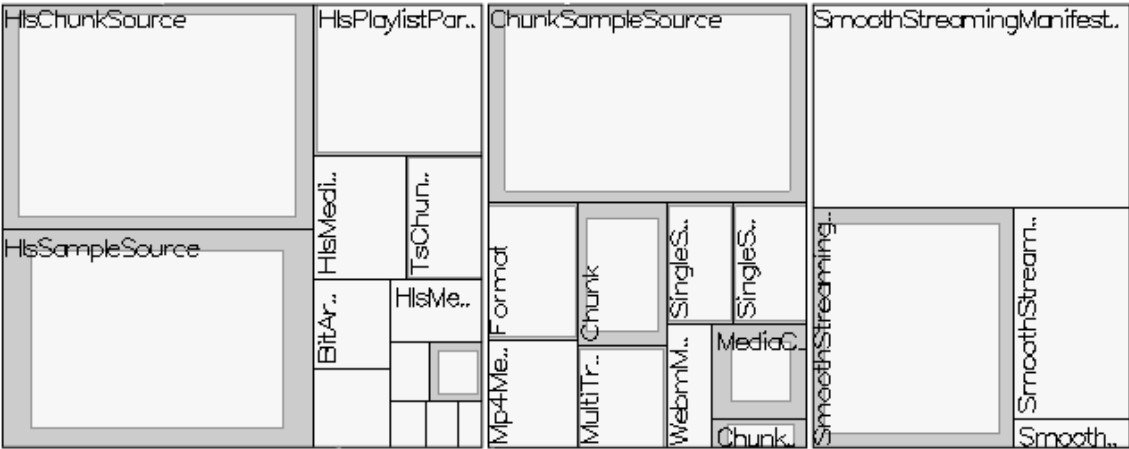
(b) Stock portfolio with squarified layout.



[illegible]

Squarified treemaps are inherently static, and in order to add information about the dynamic LOC metric value without rearranging the layout, we have encoded in the filled rectangle area the current metric value. The increasing/decreasing of value is shown with a smooth transition between states.

Figure 4.5: Base rectangle area represents maximum LOC value and filled rectangle area encode current's



Sunburst Diagram

This visualization technique uses a series of sliced rings in order to show hierarchy. Each ring corresponds to a level in the hierarchy, with the central circle representing the root node and the hierarchy moving outwards from it.

Rings are sliced up and divided based on their hierarchical relationship to the parent slice. In our project, the angle of each slice is given based on the number of leaf children each node has.

Figure 4.6: Representation of hierarchy using Sunburst Diagram

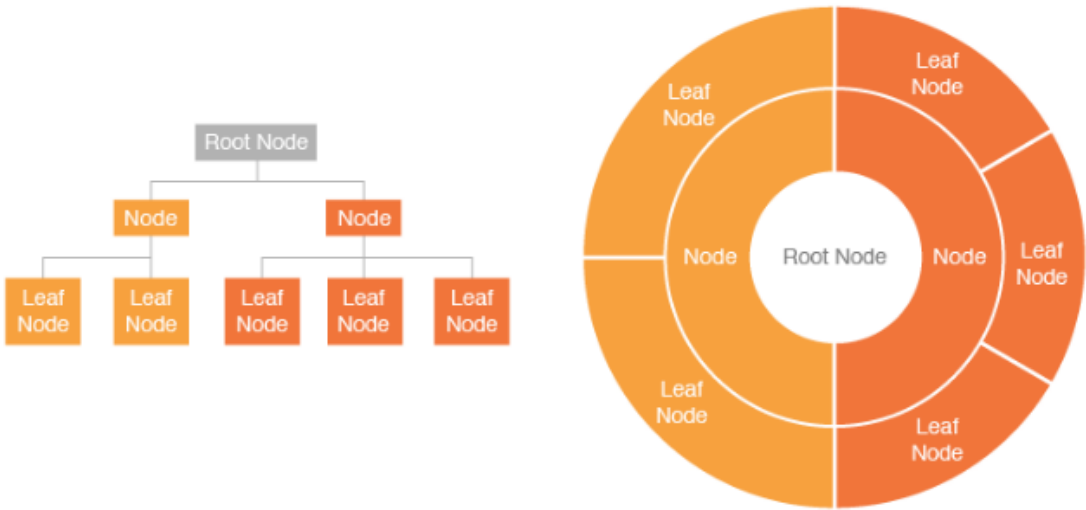
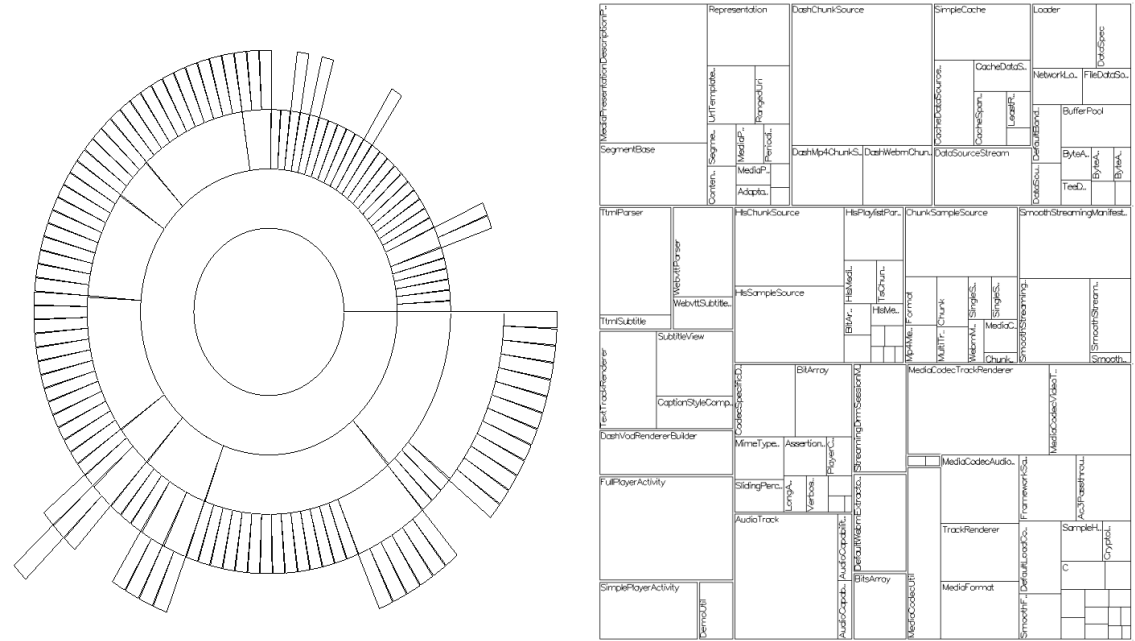


Figure 4.7: Representation of the ExoPlayer’s project hierarchy using both Sunburst Diagram and Squarified Treemap



Colormap

Colormaps are mapping functions that for every point of the domain of interest, assign to it a color based on the scalar value at that point. In this tool, three different color mapping schemes based on the ColorBrewer’s (BREWER, 2016) samples 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.

Figure 4.8: Sequential Colormap

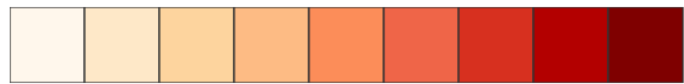


Figure 4.9: Diverging Colormap

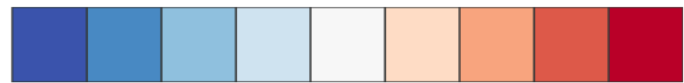
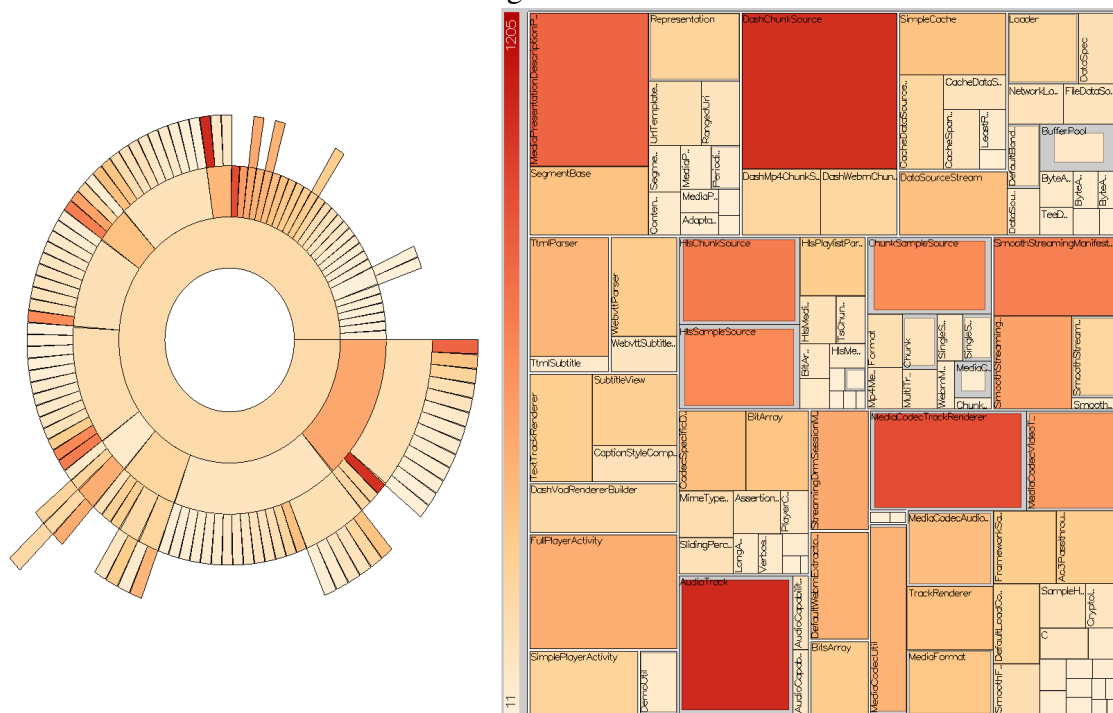


Figure 4.10: Qualitative Colormap



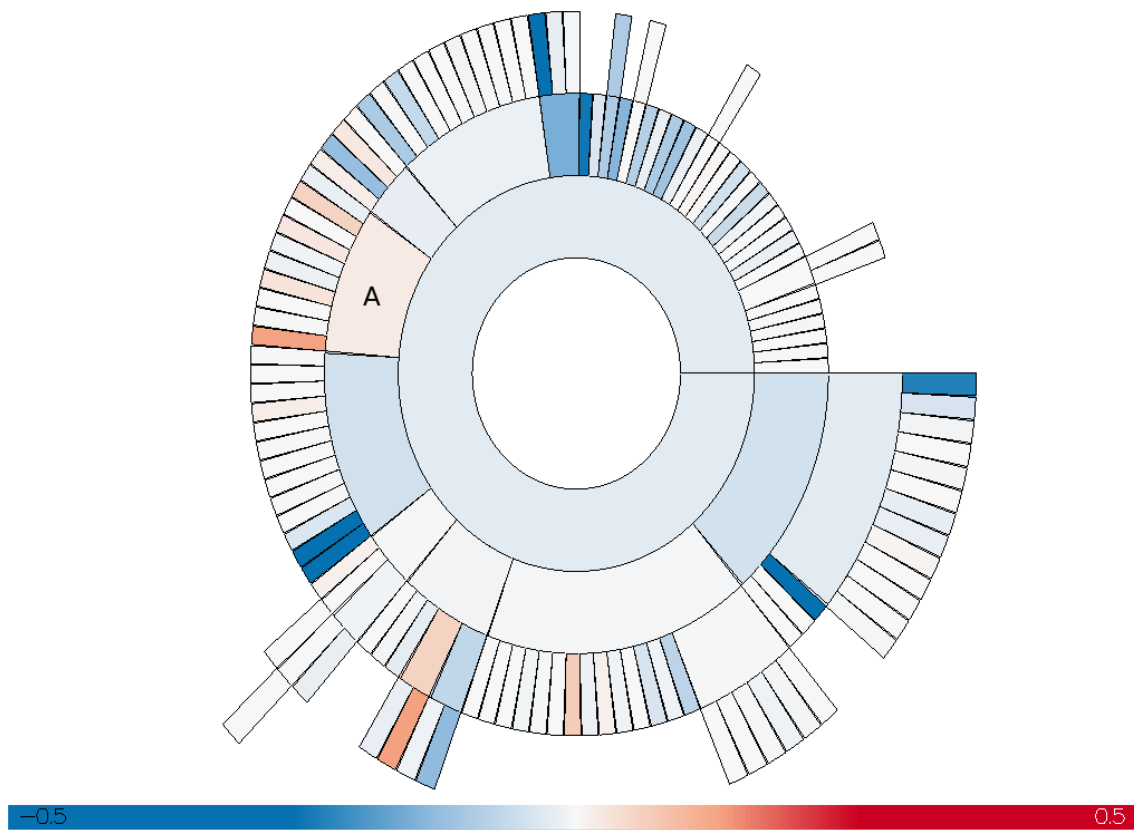
Combining colormaps with the hierarchical display techniques, we can start to understand more complex phenomena. On figure 4.11 we have used the Sequential Colormap to encode the same metric that is represented by the area of treemap rectangles (i.e. number of lines of code) in the last collected revision of the ExoPlayer project. Between the two representations there is a color legend that displays the full colormap and presents the minimum and maximum values of the selected metric for the whole project's history. The ring sections that represent packages in the Sunburst Diagram are colored with the recursive average value of its children, allowing for objective comparison between packages, and not only classes.

Figure 4.11



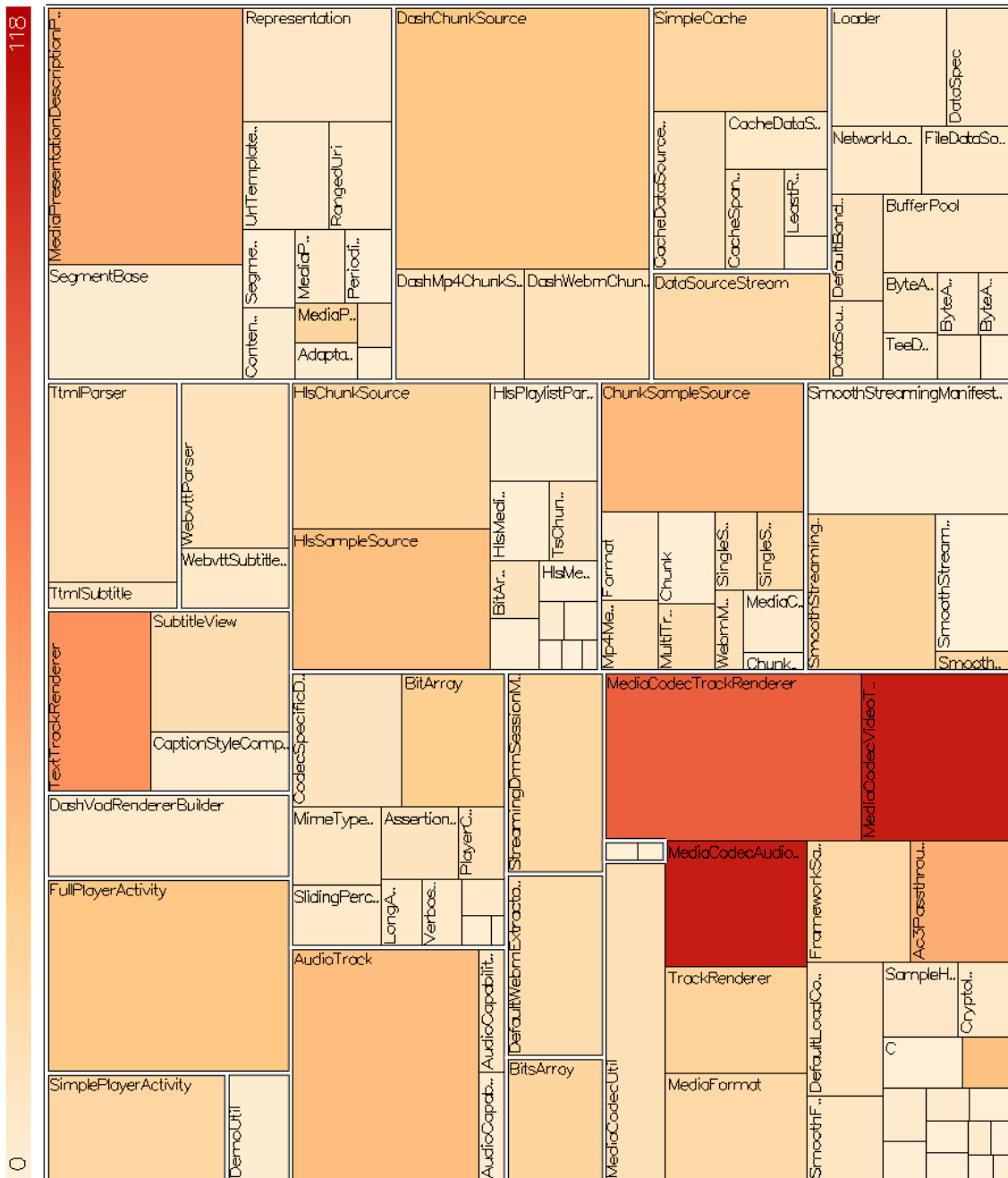
The divergent colormap can be used to analyze the activity between revisions. On figure 4.12, we can see that a lot of effort has been put in the current revision in order to decrease the size of most classes, with exception of the ones in package A, which have increased significantly. This might suggest that refactoring with code movement took place in this revision.

Figure 4.12



We can use the color mapping technique with the treemap in order to try to find interesting pattern occurrences between the LOC metric (i.e. rectangle area) and other arbitrary metrics. On Figure 4.13, we have used the Sequential Colormap to illustrate the Number of Methods per Class metric on the last revision of the Exoplayer project. Naturally, large classes tend to have more methods than small ones, but it is also possible to notice three classes with over a hundred methods inside of each, which might indicate code that is very hard to understand and maintain.

Figure 4.13: ExoPlayer Treemap with number of methods per class metric



Projection

Multidimensional projections (MP) are a set of techniques that allow efficient and effective visual analysis of high-dimensional data. They do so by mapping data points from a high-dimensionality space \mathbb{R}^n to a low-dimensionality space \mathbb{R}^m , where $n > m$ and $m \in \{2, 3\}$. This mapping aims to preserve the neighborhoods present in \mathbb{R}^n , generating visual representations that naturally convey the similarity relationship present in the

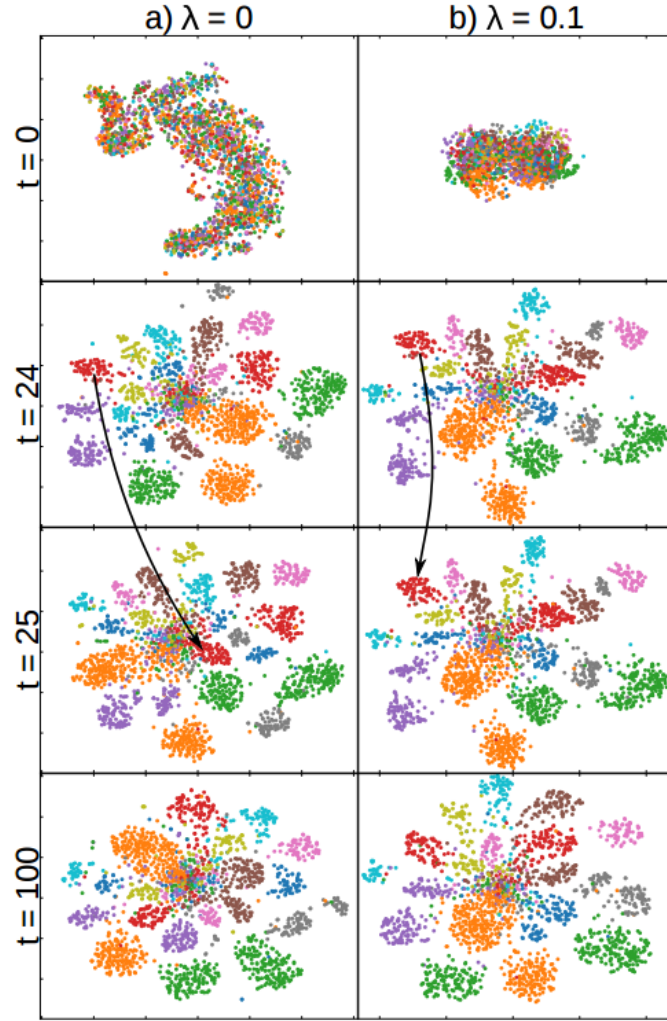
data points.

When compared to other high-dimensional visualization techniques such as table lenses (RENIERS et al., 2014), evolution lines (VOINEA; TELEA; WIJK, 2005), evolution matrices (LANZA, 2001), parallel coordinates (INSELBERG; DIMSDALE, 1990), and scatterplot matrices (HARTIGAN, 1975), 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.

Modern MP techniques, such as ISOMAP (TENENBAUM; SILVA; LANGFORD, 2000), LAMP (JOIA et al., 2011), LSP (PAULOVICH et al., 2008), and t-SNE (MAATEN; HINTON, 2008) score highly in distance preservation for static datasets, however, none of them is able to handle time-dependent datasets. The recent dynamic t-SNE (dt-SNE) (RAUBER; FALCAO; TELEA, 2016) 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 4.14 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 4.14: Dynamic t-SNE results on SVHN CNN



Source: (RAUBER; FALCAO; TELEA, 2016)

Projections serve two main goals. First, by simply removing a number of dimensions that are highly correlated or present low variance, we are able to create a simpler and easier to handle representation of the original dataset that might suit our needs. Secondly, projections can be used to assist in the exploration on high-dimensionality datasets. In this case, instead of eliminating a number of dimensions, all attributes are taken into consideration on the \mathbb{R}^n to \mathbb{R}^m mapping, generating a visual representation that can be conceptually understood in the same fashion as a 2D scatterplot or a 3D point cloud.

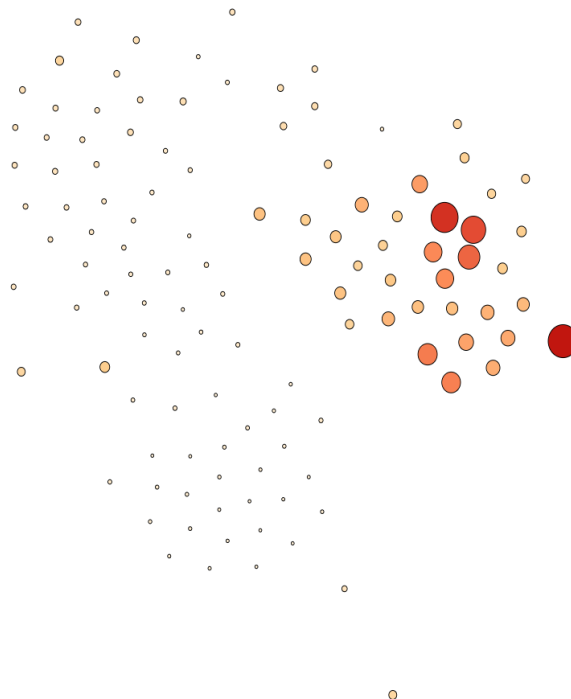
In our approach, for each revision $r_t \in [r_s, r_e]$, dt-SNE is used to generate a 2D projection $P_t = \{q_i\} \subset \mathbb{R}^2$ such that the pairwise distance between points $\|q_i - q_j\|$ are as close as possible to the high-dimensional space distance $\|e_i - e_j\|$ and the distances between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

Glyphs

The resulting dt-SNE projection is a set of points in 2D space, but we can still add a lot of information to the representation by means of color mapping, through glyphs size and shape, creating connection between points, and even by using the background's empty space to encode some other feature.

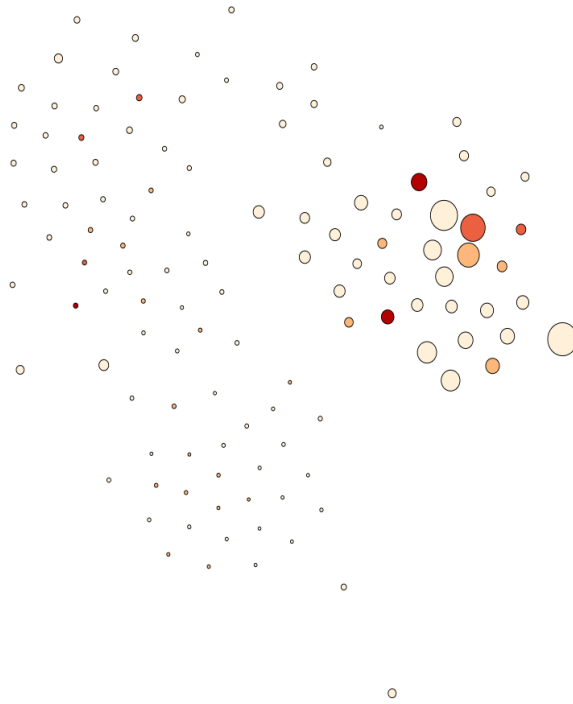
Figure 4.15 illustrates the initial tool set-up. Both the radius of the circular glyph and color are set to encode the number of lines of code metric. Right away we can start to identify cluster of points, indicating similarity on the \mathbb{R}^n space, and outliers.

Figure 4.15: LOC metric displayed in both radius and color attributes



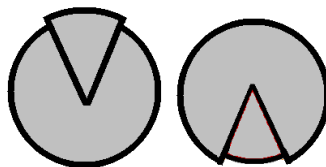
One example of use case is to change the color mapping to represent another metric. On figure 4.16, the maximum inheritance tree depth is coded in the glyph's color. This allows us to reason about the similarity relationship between entities in relation to class size (given by the radius) and the number of intermediate classes between each data point and the root class of its inheritance tree.

Figure 4.16: LOC metric displayed in both radius and color attributes



In order to add information about change of metric value to the projection, we have designed the glyph illustrated on figure 4.17. 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 4.17: 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 4.18, where the decrease is portrayed by the removal of a pie section.

Figure 4.18: Revised glyph

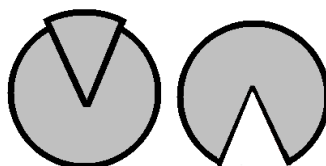
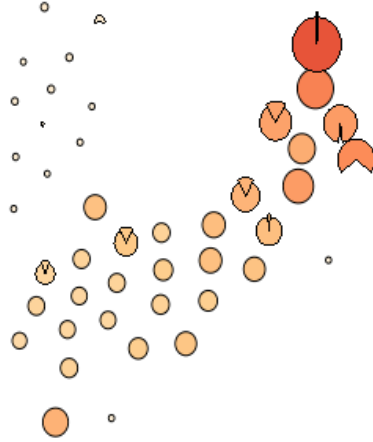


Figure 4.19: Set of entities with glyph’s shape, radius and color representing the current value and change for the Lines of Code metric.



In the application, the pie section is only drawn if the change in metric value is higher than 1%, as shown in Figure 4.19.

Visualizing Evolution

So far we are able to reason about the similarity of entities, their hierarchical relationships, their current metric values for any given collected revision, and the metric value change from revision t_i to t_{i+1} . What we yet cannot do is to intuitively display the evolution of a set of entities throughout large time intervals. With this in mind, we’ve implemented two visualization techniques.

The first is the Streamgraph, inspired on the design present on the SolidTA tool (RENIERS et al., 2014) illustrated on figure 4.20. It subdivides the screen space vertically into R sections, where R is the number of revisions being analyzed. Then, for an arbitrary number of selected classes, it draws a lines where the height at any given time subdivision represents the value of a chosen metric at that moment. This information can be emphasized by color coding (with one the colormaps introduced on section 4.2) the line section with the same metric attribute, as exemplified on figures 4.21 and 4.22 for 10 entities of the ExoPlayer project and the lines of code metric between the December 2014 and February 2016.

Figure 4.20: Evolution of the file count on the GIMP project color coded by file size using SolidTA

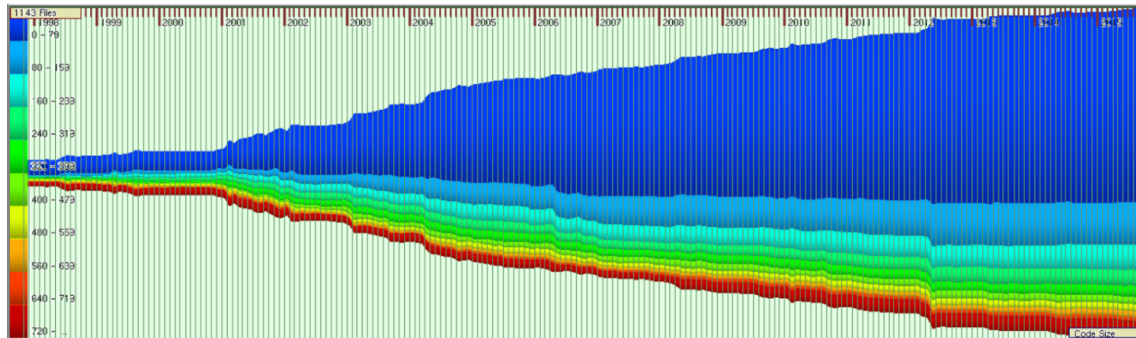


Figure 4.21: Evolution of the LOC metric for 10 classes using the sequential colormap

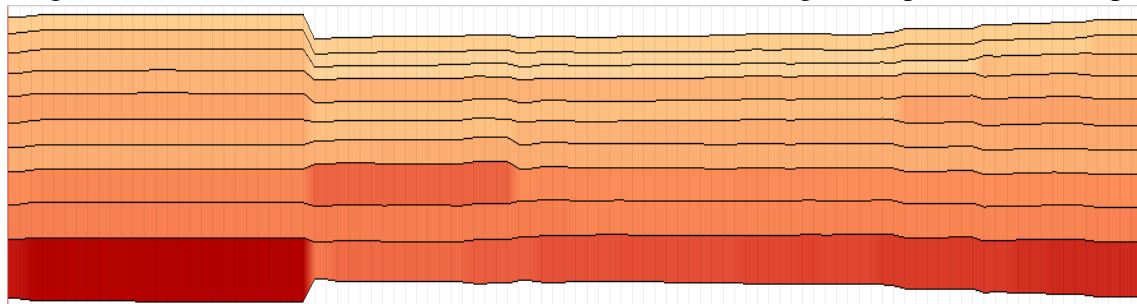
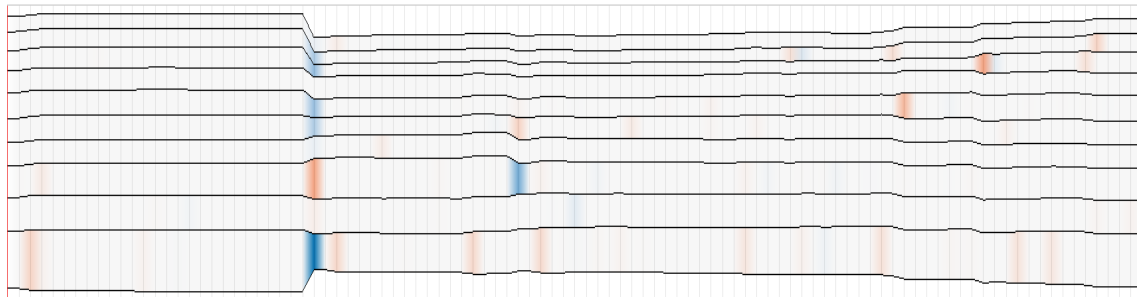


Figure 4.22: Evolution of the LOC metric for 10 classes using the divergent colormap

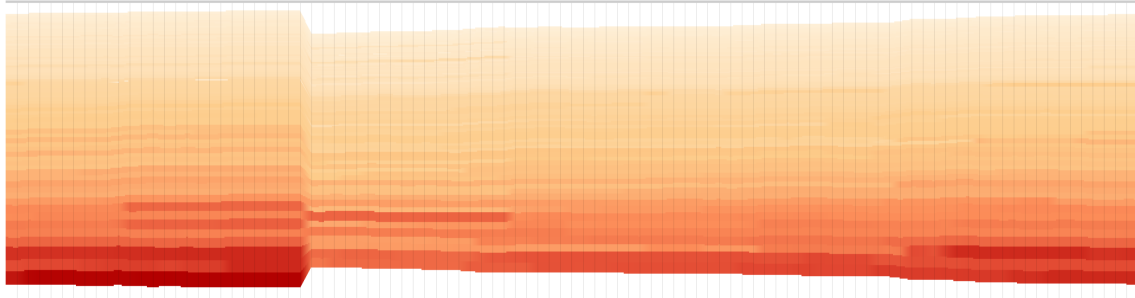


One characteristic of the Streamgraph is that it can naturally represent the global evolution of an attribute through time. Figure 4.23 shows the evolution of the LOC metric on ExoPlayer for the 15 earlier mentioned months. The biggest insight that we can extract from this visualization is not about ExoPlayer, but about our own technique.

Software projects grow and, using *git checkout* commands and the CLOC tool , we discovered that in the analyzed period of time the project has evolved from 18.4KLOC (thousand of lines of Java code) to 42.2KLOC, almost doubling the number of classes from 169 in December 2014 to 332 in February 2016. Yet, due to the filtering process (see section 3.4) that is imposed given the limitations of state of the art projection techniques, we can't see this change, supposedly because we were only able to analyze 117 of the total number of classes. Additionally, one can argue (and later, on figure 4.27, confirm)

that our classes are in a very stable state (fact that might relate to their time of creation) and that the time classes evolve the most is right after their creation.

Figure 4.23: Filtered evolution of the LOC metric on ExoPlayer from December 2014 to February 2016



The impression given by our visualization is that the size of the project hasn't increased significantly during the studied period, but we know that this is not true. In fact, an increase of 129% in the number of lines of code has taken place during this time.

This artifact is not a one-off occurrence, the same can be seen on projects such as RxJava (figure 4.24), Google Closure (figure 4.25), and Eclipse Vert.x (figure 4.25).

Figure 4.24: Filtered evolution of the LOC metric on the RxJava project from December 2013 to February 2016

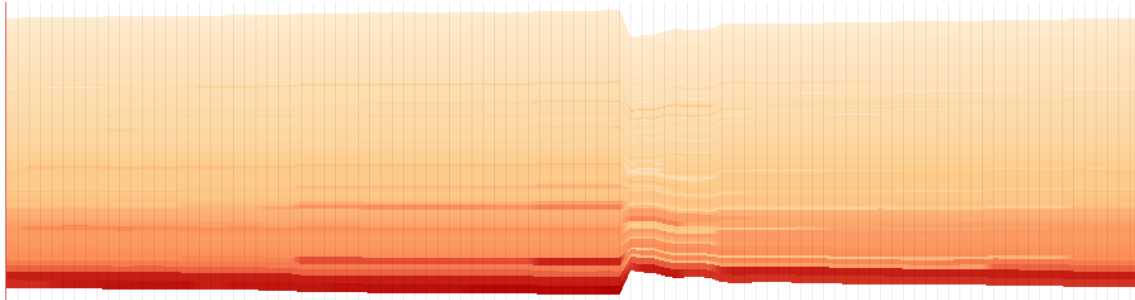


Figure 4.25: Filtered evolution of the LOC metric on the Google Closure project from August 2011 to February 2016

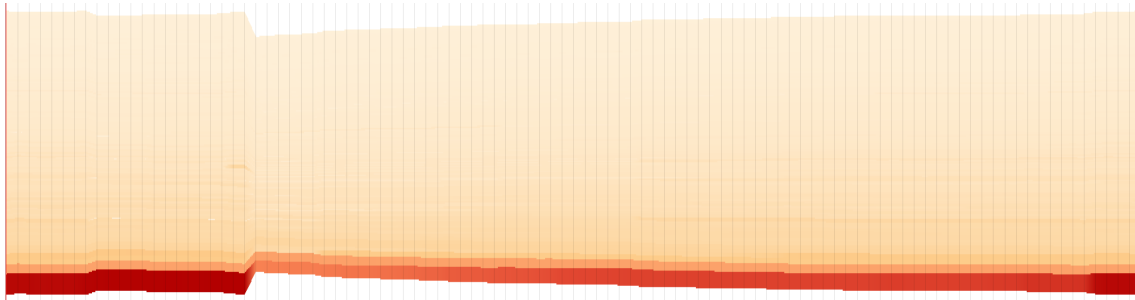
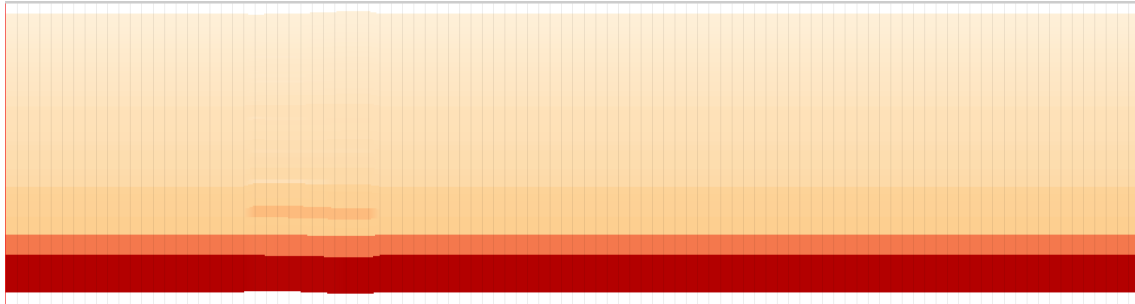


Figure 4.26: Filtered evolution of the LOC metric on the Eclipse Vert.x project from January 2014 to February 2016

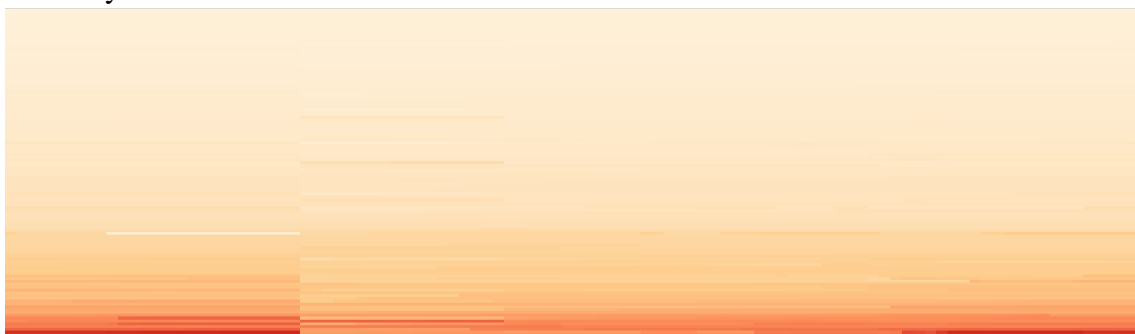


This artifact is a result of the filtering process, but not all datasets need filtering. Imagine a time dataset that consist of atmospheric, fluvial, pluvial, fauna, flora, as well as other terrain related attributes. In this case, there wouldn't be a problem, as there is no creation or deletion of terrains.

The other evolution visualization tool we implemented is the Spectrograph. In contrast to the Streamgraph, it uses a fixed line height and relies solely on color to communicate metric value at each instant. The Spectrograph excels at presenting the distribution of the data and element-wise evolution. Additionally, similar to the Treemap, it uses all “real state” available for display of data.

Figure 4.27 is different portrayal of the same data on figure 4.23. It confirms our suspicions about the stability of the data points.

Figure 4.27: Filtered evolution of the LOC metric on ExoPlayer from December 2014 to February 2016



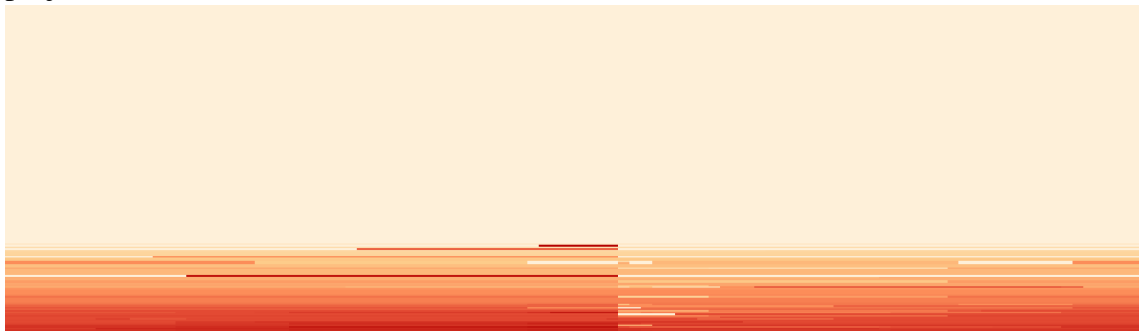
Similar to the Streamgraph, the Spectrograph can show the evolution of any metric for a set of entities. In figures 4.28 and 4.29, we compare the distribution of the Percent Lack Of Cohesion metric distributions between the ExoPlayer and RxJava projects. Cohesion is an important concept in OOP (CHIDAMBER; KEMERER, 1994), as it indicates whether a class represents a single abstraction or multiple abstractions. Cohesiveness of methods within a class is desirable, since it promotes encapsulation. Lack of cohesion suggests that classes should probably be split into two or more sub-classes, decreasing

complexity and reducing the likelihood of errors during the development process. Therefore, through this somewhat inelegant metric name, we get the understanding that lower metric values represent a better situation. What means that the RxJava project (or at least its filtered classes) has a better, least prone to errors, design than ExoPlayer.

Figure 4.28: Filtered evolution of the PercentLackOfCohesion metric on the ExoPlayer project.



Figure 4.29: Filtered evolution of the PercentLackOfCohesion metric on the RxJava project.



Linking the Views

So far we have three views. A projection view on the left, a hierarchical view on the right, that by default displays the Squarified Treemap, and if solicited through the UI, an evolution view on the bottom.

Selecting or brushing elements or groups of elements on one of the views, highlights them on the others. This is a simple mechanism that attempts to support the visualization of correlation between \mathbb{R}^n similarity, project structure and metric evolution. Its downside is that it relies on having to move the user's attention from one view to the next. In this section, we experiment with ways of hinting information about hierarchy on top of the projection (which as far as we are concerned, no one has attempted to on the literature) and vice versa.

For the projection, we've developed a technique based on force-directed graph layout that links each class to its hierarchical parent. Points in the projection (i.e. classes) are constrained to their original \mathbb{R}^2 positions (leaf nodes) and package representative nodes are free to move according to the forces acting upon them (non leafs).

In this graph, each class point connects to its package representative. In turn, every package representative node connects to the tree root node.

Each node is capable of generating attraction and repulsion forces. Two linked nodes attract each other with a force linearly proportional to their distance. Additionally, each node repels all other nodes with a force that degrades quadratically with their pairwise distance and is scaled by the node's radius in order to try to avoid overlap.

If a node is able to move (i.e. package representative or root), the sum of the forces acting upon it as well as its current velocity dictate its position on the next iteration, creating a system that simulates particle momentum. This system iterates until it reaches equilibrium, which is only broken when we move to the next revision and node positions change. In which case, the system smoothly transitions to a new stable configuration.

Figure 4.30 is an example of a stable configuration for a simple hierarchy. Figure 4.31 shows the same scenario for a more complex dataset (117 classes). With this number of classes it is already impossible to get any hierarchical insight. But if we are only interested in understanding the \mathbb{R}^n neighborhood of a couple of packages, figure 4.32 shows that it is possible to do so through filtering.

Figure 4.30: Linking nodes hierarchically on a synthetic projection



Figure 4.31: Linking nodes hierarchically on the ExoPlayer project

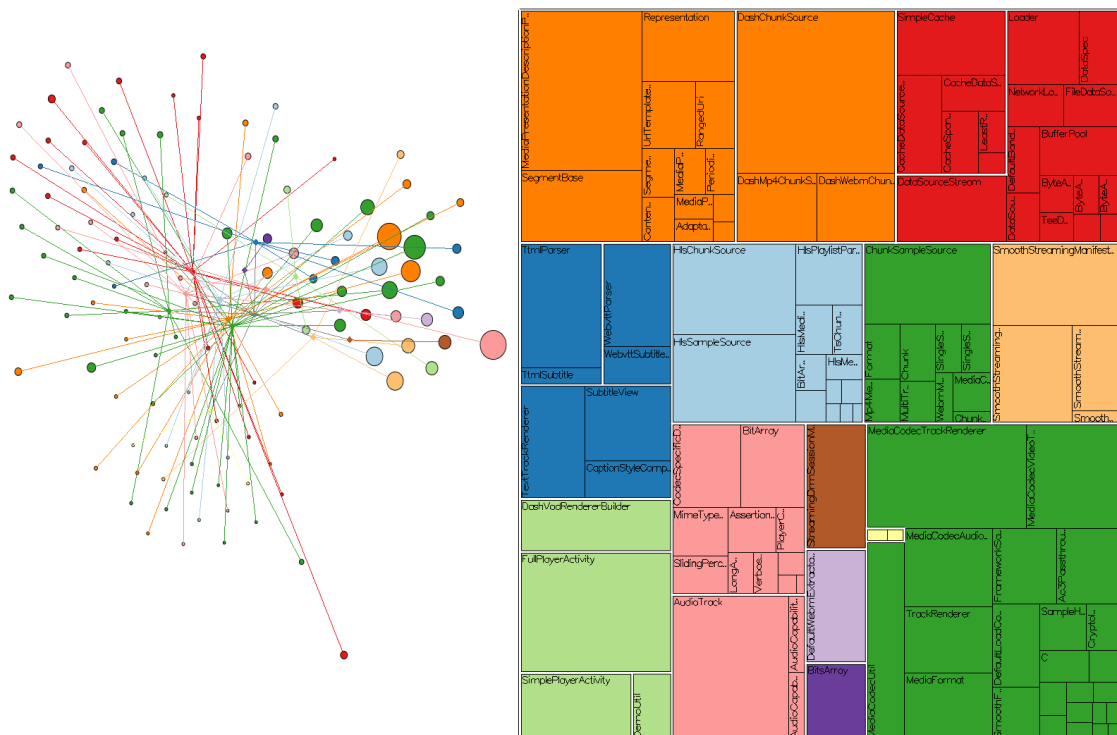
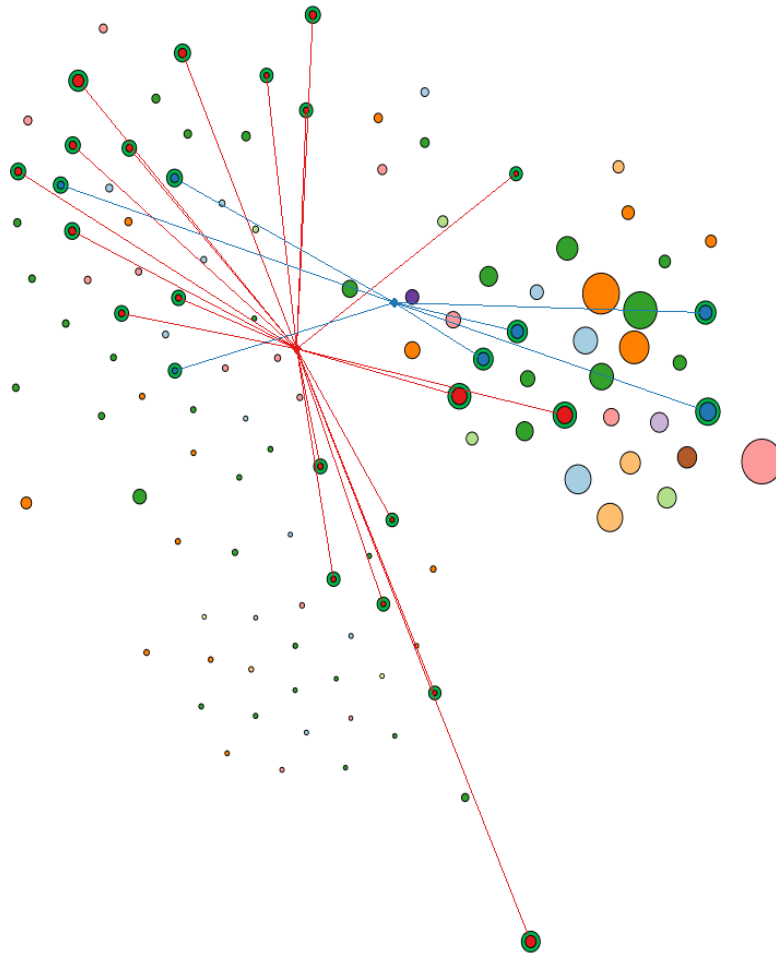


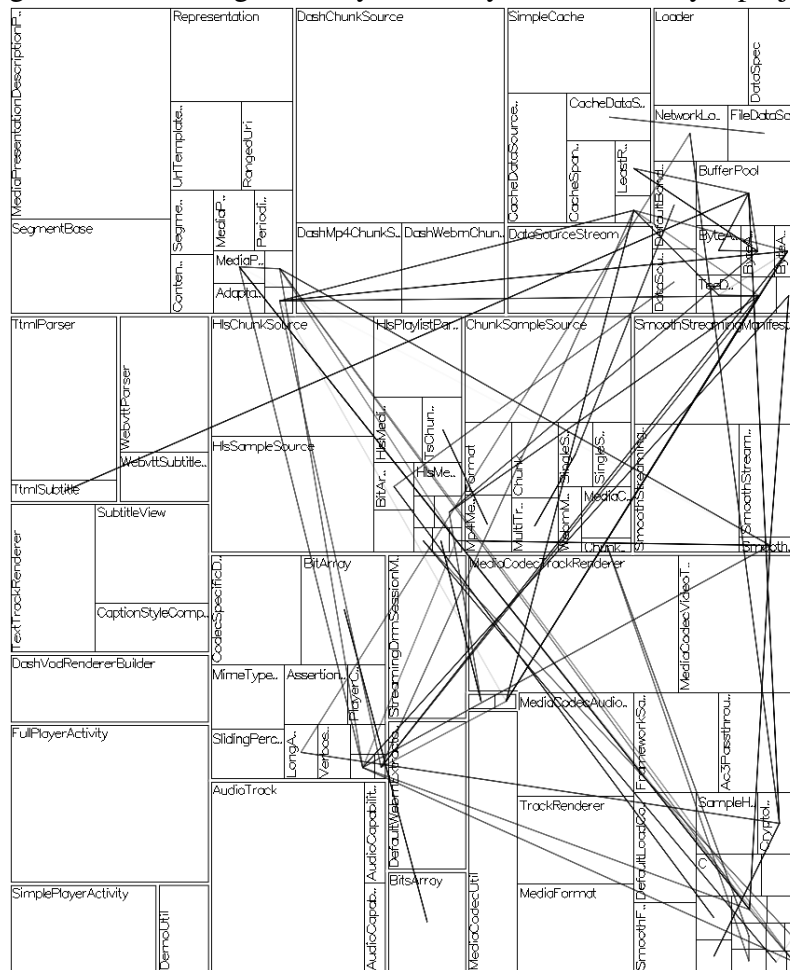
Figure 4.32: Linking filtered packages hierarchically on the ExoPlayer project



This experiment was an attempt to find correlation between similarity of classes and their location in the project structure. The conclusion we arrived at is that, for our datasets, there is none. Which is an interesting result nevertheless.

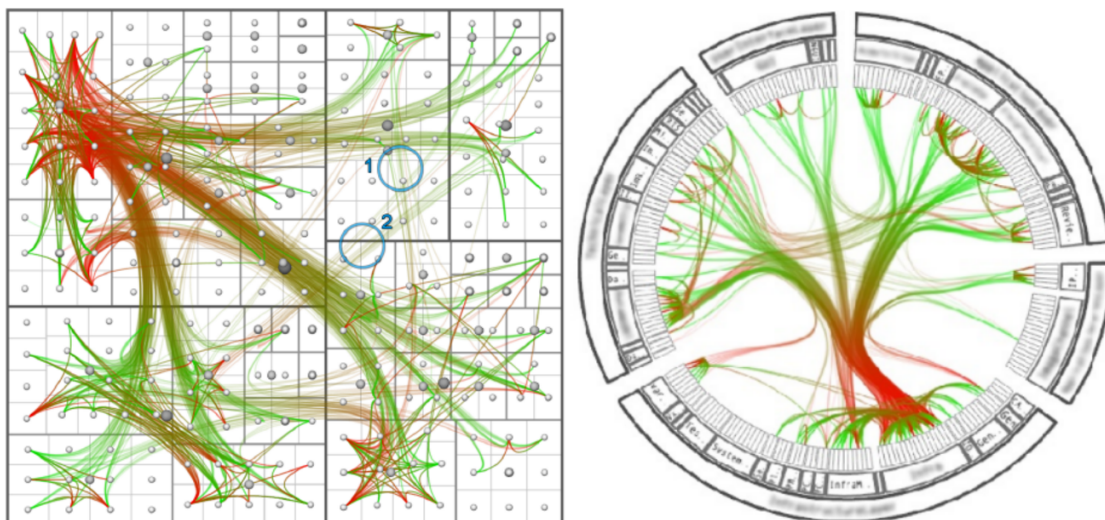
Our other goal was to encode similarity on the Treemap. In order to do so, we computed the \mathbb{R}^n distances between every pair of points, selecting only the 1% closest points. Between each pair in this set, we drew a line which uses the opacity to communicate the strength of the resemblance between the two points. The result for a single revision can be seen on figure 4.33. It is interesting to see as the project evolves, how these relationships change.

Figure 4.33: Linking nodes by similarity on the ExoPlayer project



One future improvement would be to use Holten (2006) approach with edge bundling to encode the similarity between nodes, as presented in figure 4.34.

Figure 4.34: Visualization of a software system's call graph with edge bundling



Source: (HOLTEN, 2006)

REFERENCES

- AIGNER, W. et al. **Visualization of Time-Oriented Data**. [S.l.]: Springer-Verlag London, 2011.
- AUBER, D. et al. Gospermap: Using a gosper curve for laying out hierarchical data. **IEEE Transactions on Visualization and Computer Graphics**, IEEE Computer Society, Los Alamitos, CA, USA, v. 19, n. 11, p. 1820–1832, 2013. ISSN 1077-2626.
- BALZER, M.; DEUSSEN, O. Voronoi treemaps. IEEE Computer Society, Washington, DC, USA, p. 7–, 2005. Available from Internet: <<http://dx.doi.org/10.1109/INFOVIS.2005.40>>.
- BEDERSON, B. B.; SHNEIDERMAN, B.; WATTENBERG, M. Ordered and quantum treemaps: Making effective use of 2d space to display hierarchies. **ACM Trans. Graph.**, ACM, New York, NY, USA, v. 21, n. 4, p. 833–854, oct. 2002. ISSN 0730-0301. Available from Internet: <<http://doi.acm.org/10.1145/571647.571649>>.
- BREWER, C. **Color Brewer**. 2016. <<http://colorbrewer2.org/>>. Accessed: 2016-06-25.
- BRULS, M.; HUIZING, K.; WIJK, J. van. Squarified treemaps. **Data Visualization 2000: Proc. Joint Eurographics and IEEE TCVG Symp. on Visualization**, 2000.
- CHACON, S. **Pro Git**. [S.l.]: Apress, 2009.
- CHIDAMBER, S.; KEMERER, C. A metrics suite for object oriented design. **IEEE Transactions on Software Engineering**, v. 20, n. 6, p. 476–493, 1994.
- DIEHL, S. **Software visualization: visualizing the structure, behaviour, and evolution of software**. [S.l.]: Springer, 2007.
- FOUNDATION, T. E. **Eclipse Neon Release Train Now Available**. 2016 (accessed 2016–11–9). <http://www.eclipse.org/org/press-release/20160622_neon.php>.
- GERSHON, N. D. From perception to visualization. *Scientific Visualization: Advances and Challenges*. Academic Press, 1994, 1994.
- HARTIGAN, J. Printer graphics for clustering. **Journal of Statistical Computation and Simulation**, v. 4, n. 3, p. 187–213, 1975. Available from Internet: <<http://dx.doi.org/10.1080/00949657508810123>>.
- HOLTEN, D. Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. **IEEE Transactions on Visualization and Computer Graphics**, 2006.
- INSELBERG, A.; DIMSDALE, B. Parallel coordinates: A tool for visualizing multi-dimensional geometry. In: **Proceedings of the 1st Conference on Visualization '90**. Los Alamitos, CA, USA: IEEE Computer Society Press, 1990. (VIS '90), p. 361–378. ISBN 0-8186-2083-8. Available from Internet: <<http://dl.acm.org/citation.cfm?id=949531.949588>>.
- JOIA, P. et al. Local affine multidimensional projection. **IEEE TVCG**, v. 17, n. 12, p. 2563–2571, 2011.

LANZA, M. The evolution matrix: Recovering software evolution using software visualization techniques. ACM, New York, NY, USA, p. 37–42, 2001. Available from Internet: <<http://doi.acm.org/10.1145/602461.602467>>.

MAATEN, L. van der; HINTON, G. E. Visualizing high-dimensional data using t-SNE. **J Mach Learn Res**, v. 9, p. 2579–2605, 2008.

ONAK, K.; SIDIROPOULOS, A. Circular partitions with applications to visualization and embeddings. ACM, New York, NY, USA, p. 28–37, 2008. Available from Internet: <<http://doi.acm.org/10.1145/1377676.1377683>>.

OVERFLOW, S. **Stack Overflow 2015 Developer's Survey**. 2016 (accessed 2016–11–9). <<http://stackoverflow.com/research/developer-survey-2015>>.

PAULOVICH, F. V. et al. Least square projection: A fast high-precision multidimensional projection technique and its application to document mapping. **IEEE TVCG**, v. 14, n. 3, p. 564–575, 2008.

PEARCE, S. **GitSvnComparison**. 2016 (accessed 2016–7–3). <<https://git.wiki.kernel.org/index.php/GitSvnComparison>>.

RAUBER, P.; FALCAO, A.; TELEA, A. Visualizing time-dependent data using dynamic t-sne. in **Proceedings EuroVis Short Papers**, 2016.

RENIERS, D. et al. The solid toolset for software visual analytics of program structure and metrics comprehension: From research prototype to product. **Science of computer programming**, Elsevier, v. 79, p. 224–240, 1 2014. ISSN 0167-6423. Relation: <http://www.rug.nl/research/jbi/> Rights: University of Groningen, Johann Bernoulli Institute for Mathematics and Computer Science.

RIBECCA, S. **Data Visualization Catalogue**. 2016 (accessed 2016–11–9). <<http://www.datavizcatalogue.comAndgt>>.

SCITOOL. **Metric Implementation Notes**. 2016 (accessed 2016–7–3). <<https://scitools.com/documents/metricImplementationNotes.pdf>>.

SHNEIDERMAN, B.; WATTENBERG, M. Ordered treemap layouts. IEEE Computer Society, Washington, DC, USA, p. 73–, 2001. Available from Internet: <<http://dl.acm.org/citation.cfm?id=580582.857710>>.

SPENCE, R. **Information Visualization: Design for Interaction (2Nd Edition)**. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2007. ISBN 0132065509.

TEAM, T. G. **About Gimp**. 2016 (accessed 2016–11–9). <<https://www.gimp.org/about/>>.

TENENBAUM, J. B.; SILVA, V. de; LANGFORD, J. C. A global geometric framework for nonlinear dimensionality reduction. **Science**, v. 290, n. 5500, p. 2319, 2000.

VOINEA, L.; TELEA, A.; WIJK, J. J. van. Cvsscan: Visualization of code evolution. ACM, New York, NY, USA, p. 47–56, 2005. Available from Internet: <<http://doi.acm.org/10.1145/1056018.1056025>>.

WANG, L.; ZHANG, Y.; FENG, J. On the euclidean distance of images. **IEEE Transactions on Pattern Analysis and Machine Intelligence** 2005.