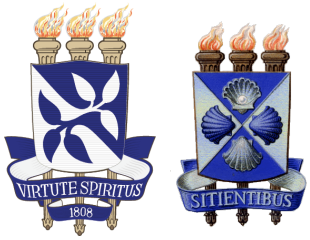


Universidade Federal da Bahia
Universidade Estadual de Feira de Santana

DISSERTAÇÃO DE MESTRADO

EVOWAVE: A multiple domain metaphor for software evolution visualization

Rodrigo Chaves Magnavita

Mestrado em Ciência da Computação – MMCC

Salvador
15 de Janeiro de 2016

RODRIGO CHAVES MAGNAVITA

EVOWAVE: A MULTIPLE DOMAIN METAPHOR FOR SOFTWARE EVOLUTION VISUALIZATION

Dissertação apresentada ao Mestrado em Ciência da Computação da Universidade Federal da Bahia e Universidade Estadual de Feira de Santana, como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Orientador: Manoel Gomes de Mendonça Neto
Co-orientador: Renato Lima Novais

Salvador
15 de Janeiro de 2016

M196 Magnavita, Rodrigo Chaves.
Evowave: a multiple domain metaphor for software evolution visualization/ Rodrigo Chaves Magnavita. – Salvador, 2016.
89 f. : il. color.

Orientador: Prof. Manoel Gomes de Mendonça Neto
Co-orientador: Prof. Renato Lima Novais.

Dissertação (Mestrado) – Universidade Federal da Bahia. Instituto de Matemática, 2016.

1. Engenharia de Software. 2. Evowave. 3. Evolução de software. I. Mendonça Neto, Manoel Gomes de. II. Novais, Renato Lima. III. Universidade Federal da Bahia. IV. Título.
CDD: 005.3

TERMO DE APROVAÇÃO

RODRIGO CHAVES MAGNAVITA

EVOWAVE: A MULTIPLE DOMAIN METAPHOR FOR SOFTWARE EVOLUTION VISUALIZATION

Esta dissertação foi julgada adequada à obtenção do título de Mestre em Ciência da Computação e aprovada em sua forma final pelo Mestrado em Ciência da Computação da UFBA-UEFS.

Salvador, 15 de Janeiro de 2016

Prof. Dr. Manoel Gomes de Mendonça Neto
Universidade Federal da Bahia

Prof. Dr. Rodrigo Rocha Gomes e Souza
Universidade Federal da Bahia

Prof. Dr. Rodrigo Oliveira Spínola
Universidade Salvador

RESUMO

A evolução do software produz uma grande quantidade de dados durante os ciclos de desenvolvimento. Engenheiros de software precisam interpretar esses dados para extrair informações que os auxiliarão na execução de suas atividades diárias. O uso de Visualização de Evolução de Software (VES) tem sido uma abordagem promissora para auxiliar nessa interpretação. Essa abordagem faz uso de recursos visuais que facilitam a interpretação desses dados. Ainda assim, não é trivial representar visualmente todos os dados gerados durante a evolução do software, pois além do software possuir diferentes entidades e atributos, ainda é necessário lidar com a dimensão temporal da evolução.

As VES geralmente são construídas com objetivo de auxiliar na execução de atividades relacionadas a um domínio específico da engenharia de software. Muitas dessas visualizações focam apenas em apresentar uma visão geral da evolução do software, sem focar nos detalhes. Entretanto, a maioria das atividades de desenvolvimento de software requer tanto combinar diferentes domínios quanto ter uma visão detalhada das informações. As metáforas visuais (i.e., conceitos, associações e analogias a entidades concretas) utilizadas nessas visualizações, são muito específicas, objetivando auxiliar apenas um determinado domínio. O uso de múltiplas visões do software para construir o modelo mental do sistema vem sendo apontado como uma abordagem efetiva para o completo entendimento do mesmo. Na maioria dos casos, essas visualizações possuem conjuntos de metáforas visuais. Devido a isso, surge uma necessidade do engenheiro de software compreender e se familiarizar com as metáforas visuais de cada uma das visualizações durante a utilização das mesmas. Uma das formas de mitigar esse problema é usar visualizações que possuem uma única metáfora visual para visualizar diversos aspectos e perspectivas do software.

Esta dissertação apresenta uma nova metáfora visual, chamada EVOWAVE, capaz de ser utilizada em múltiplos domínios e que permite visualizar os dados de forma global e detalhada. A EVOWAVE é inspirada em ondas concêntricas observadas de cima. Essa metáfora consegue representar grandes quantidades de dados e seus conceitos são transversais a domínios na área de engenharia de software. O desenvolvimento desta metáfora passou por fases iterativas que refinaram os conceitos associados a ela. Primeiramente foi desenvolvido um protótipo que validou a capacidade da metáfora de representar grandes quantidades de dados. Em seguida, foram realizados estudos para validar a capacidade de representar dados de diferentes domínios. Os resultados indicam que a metáfora proposta pode ser utilizada de forma efetiva em diferentes domínios da área de engenharia de software para auxiliar na execução de atividades de manutenção e evolução.

Palavras-chave: Visualização de Software, Evolução de Software, Compreensão de Software, Engenharia de Software

ABSTRACT

The software evolution produces a lot of data during software development. Software engineers need to interpret these data to extract information that will help them in carrying out their daily activities. The use of Software Evolution Visualization (SEV) has been a promising approach to support this interpretation. This approach makes use of visual attributes that facilitate the interpretation of such data. Still, it is not trivial to visually represent all the data generated during the software development because software have different entities and attributes. Nevertheless, it is still necessary to deal with the temporal dimension of evolution.

The SEV are usually built with the goal of helping activities related to a specific domain of software engineering. Many of these visualizations focus only on presenting an overview of the software development, without focusing on the details. However, most software development activities requires both: combine different domains and detailed information. Visual metaphors (i.e., concepts, associations and analogies to specific entities) used in these visualizations are very specific, aiming to assist only a certain domain. The use of multiple visualizations of the software to build the mental model of the system has been touted as an effective approach for the complete understanding of it. In most cases, these visualizations have a sets of visual metaphors. Because of this, the software engineer need to understand and become familiar with the visual metaphors of each of the visualizations while using them. One way to mitigate this problem is to use visualization that have a unique visual metaphor to view various aspects and perspectives of software.

This work presents a new visual metaphor, called EVOWAVE, able to be used in multiple domains and to visualize the data in a comprehensive and detailed way. EVOWAVE is inspired by concentric waves as seen from above. This metaphor can represent large amounts of data and concepts cut across domains in the software engineering field. The development of this metaphor went through iterative phases that have refined the concepts associated with it. First we developed a prototype that has validated the ability of metaphor to represent large amounts of data. Then, studies were performed to validate the ability to represent information in different domains. The results indicate that the proposed metaphor can be used effectively in different domains of software engineering to assist in the execution of maintenance and development activities.

Keywords: Software Visualization, Software Evolution, Software Comprehension, Software Engineering

CONTENTS

LIST OF FIGURES

Chapter 1

This chapter presents a overview about the concepts used in this work. Its main goal is to describe the context, motivation and problem related to this dissertation as well as the goals and contributions describing the approach to achieve them.

INTRODUCTION

1.1 CONTEXT

Software evolution has been highlighted as one of the most important topics in software engineering (Novais et al., 2013). It has very complex activities because the software development process generates a huge amount of data and has many stakeholders. Dealing with these scenarios is challenging because developers may use over 60% of the maintenance effort understanding the software (Corbi, 1989). Since the maintenance effort uses up to 90% of the software budget (Erlikh, 2000), the time spent to understand the software in order to perform a maintenance activity uses up 54% of the budget.

This leads the software engineering research community to create methods, processes and techniques to improve software comprehension. The goal is to increase the overall effectiveness of software development by making the software more comprehensive. This represents a challenge because as the software evolves it becomes more complex due to the insertion of more features, the involvement of new developers and other factors. The more complex the software the more difficult it is to extract information about it, therefore, the harder it becomes to comprehend it.

Software visualization has been used as one way to deal with software comprehension activities. Software visualization usage is currently increasing. It helps people to understand software through visual elements, reducing complexity to analyze the amount of data generated during the software evolution (Diehl, 2007). Some examples of what this data can be are: software metrics, stakeholders, bugs, features.

This data increases faster when we are dealing with software evolution. Every day a series of events occur during the software development process (e.g. changes in the source code, mails exchanged within the project team, changes in the team, the emergence of new technologies). These events generate a huge amount of data rendering the handling all these data a difficult task to understand the software. That is why software visualization

is a good approach to organize these data and help extract information about the software. Nevertheless, building visual metaphors that effectively represent the time dimension with all the data related to software evolution is a challenge in the field of software evolution.

1.2 MOTIVATION

There has been much effort by the academy and industry to visually organize the software data. Eclipse IDE (Integrated Development Environment), one of the most used in the academic and industrial setting, has several visualizations called views. They are used for different tasks (e.g. debug, aspect configuration, bug correction). Each view organizes its data in a specific way.

Figure 1.1[1] illustrates two of those views highly used during software development tasks: Enterprise Explorer (A and B) and Type Hierarchy (C). Enterprise Explorer is used to visualize the elements of a Java Enterprise Web Project. In our example, on (A), we can see the main structures regarding the Web Project named "Project Example". On (B), it is possible to visualize the details of this Web Project, for example, which Servlets or Filters it has registered. Type Hierarchy is used to visualize the hierarchy between Java elements. In this case, it integrates the overview information with the details in the same view. On the left it is possible to identify which classes implements the interface List (overview information) and, on the right, which methods are specific of the ArrayList class, and which methods came from some element up in the hierarchy (detailed information). As in Eclipse IDE, software visualizations follow the strategy of only showing an overview picture of the system (overview strategy), only the details (detail strategy), or both.

The overview strategy gives more transverse information about the task to be performed. The amount of data visualized is reduced because only high level data is considered. The detail strategy gives a lot of low level details about some aspect of the software. The amount of data visualized is increased when compared to the overview strategy because of all the low level data considered. According to Shneiderman (Shneiderman, 1996), in his visualization mantra: overview first, then details on demand, the mix of those two strategies are the ideal. The overview picture is considered to identify, in a high level abstraction, which part of the software needs to be thoroughly analyzed. In the context of software evolution visualization, authors have taken different approaches. Some present the big picture of the software, providing an overview of the entire software history (Kuhn et al., 2010)(Voinea; Telea, 2006)(Lungu, 2008), while others show snapshots of the software evolution in detail (Abramson; Sosic, 1995)(Novais et al., 2011)(Novais et al., 2012)(Bergel et al., 2011)(D'ambros; Lanza; Lungu, 2009). They are both important because each approach fits better to specific software evolution tasks. An important issue in the area is to understand how to combine both approaches in a practical and useful way so that users can really take advantage of the visualizations proposed (Novais et al., 2013).

A mapping study performed in the area (Novais et al., 2013) highlighted other issues for the software evolution visualization community. Many works address software evolution

[1]Since this is a software visualization work, and to improve understanding, it is important to see this picture, and the others, in a colored version

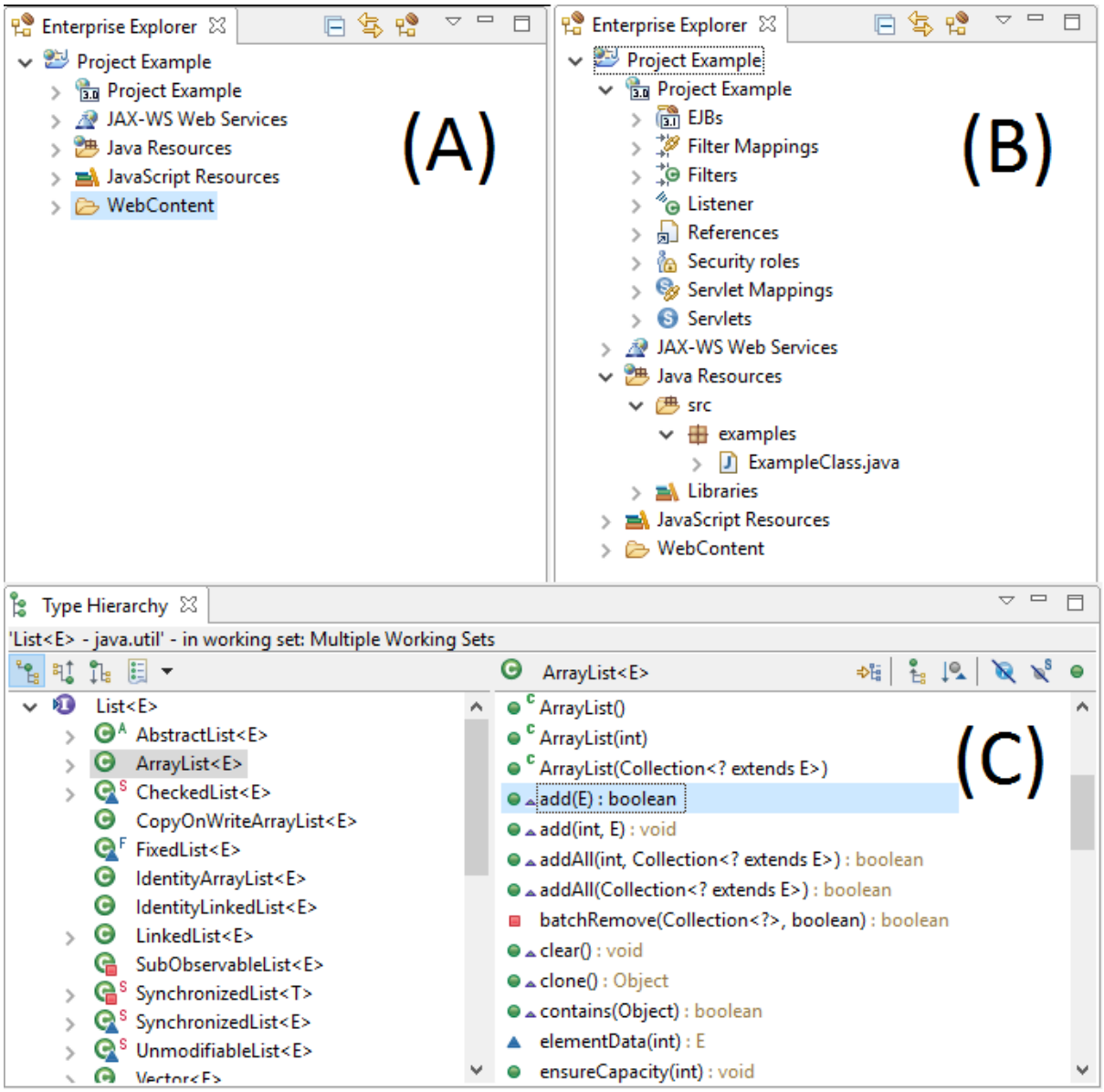


Figure 1.1 Two software data visualizations embedded in Eclipse IDE.

by viewing only one type of data (e.g. source code change, defects, features). They do not usually display or cross-reference different information that can be recovered from different sources. Again, they are able to help users to perform few or specific tasks.

A visual metaphor is a representation of a person, place, thing, or idea by way of a visual image that suggests a particular association or point of similarity. The use of multiple metaphors for different tasks can increase the comprehension time. For each one of those, software engineers need to learn how it works before actually being able to extract valuable information for the task. It is difficult to create a metaphor that helps in all software engineering tasks, because it would display too much data at the same time. The use of filters to reduce the amount of data displayed is a common technique

used by the views.

These issues motivated this work to develop a metaphor that can represent both the overview and the details of software evolution for different domains using different data sources.

1.3 PROBLEM STATEMENT

One of the main problems in software evolution visualization area is the amount of data that needs to be analyzed. Besides the normal complexity to visualize several software elements of one version, the visualization still needs to handle the time component, increasing the amount of data. Time introduces more information when we analyze each version of the software at the same time. Visualization must portray both the data about the software in each version, and the data between the versions.

These data are storage in different software repositories (e.g., source code repositories, bug repositories, and mail lists) with different semantics (e.g., metrics, classes, and bugs) to help different tasks (e.g., comprehension, refactoring, and library dependency upgrades). Given the diversity of this data, the majority of works in the area focus in the creation of metaphors to solve specific problems. They solve this problem by the analysis of some data semantics extracted from few sources. Unfortunately, this approach created a huge number of metaphors in the field (Novais et al., 2013).

Software engineers usually need to perform different tasks in their daily work. For each of those tasks, and following the aforementioned approach, they will need to use a different metaphor with different concepts, methodologies, and tools to address the task at hand. The problem is that this may require a considerable time simply learning or adapting his/her mind to the specific metaphor.

To develop a metaphor that encompass same concepts, methodologies and tools aiming to help software engineers performing different tasks from different domains (e.g., software collaboration, software architecture, and library dependency) is not an easy achievement. The metaphor needs to be generic enough to be able to represent different data and also be able to show detailed information.

1.4 GOAL

In this work, we aim to propose a new software evolution visualization metaphor called EVOWAVE. It is able to visualize different types of data generated in the development process using both overview and detail approaches. EVOWAVE can represent a huge number of events which occurred during the software development in a glance. It can be applied to different software engineering tasks and domains.

We can point out the following specific goals:

- To idealize and specify a metaphor that can represent different types of data from different domains from the overview to the details.
- Develop a tool with the metaphor implemented.
- Implement algorithms to extract data from different repositories.

- Perform experimental studies to validate the use of the metaphor in multiple domains.

1.5 APPROACH

Figure 1.2 presents an overview of the work performed in this research, activity by activity in chronological order. The white rectangle with orange borders represents mandatory classes of the masters program. The green rectangles represent theoretical activities. The blue rectangles represent development activities. The blue circles with the letter P inside indicates a publication accomplished. The white circles with the letter P inside indicates a publication that will be submitted. The blue arrows link the publication to the items related to it.

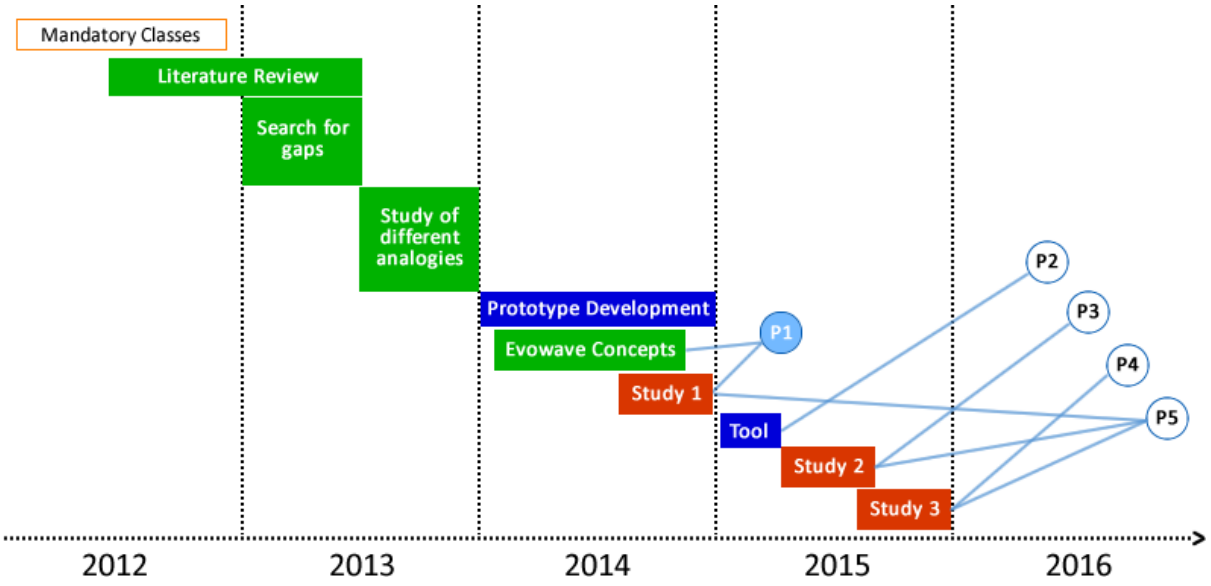


Figure 1.2 The work performed in this research.

We start this work through informal study on material given in the Software Evolution class. In this moment, a set of material regarding to Software Evolution, Software Visualization, Software Visualization Evolution was read. Later, the systematic mapping review by Novais (Novais et al., 2013) was used to find gaps in the area. During this activity we observed the huge number of metaphors proposed in the area, each one addressing a specific task. At this point we start to explore this gap analyzing why the proposed metaphors can be only used to the proposed context. During this analysis we identify that the main problem was not in the layout or in how the information was displayed, but the fact that concepts introduced were too specific for a domain.

After that, we started trying to identify natural phenomenons and common known analogies (i.e. possible metaphors) that could be applied to our problem. We based in analogies already used and others that were not used yet. The first analogy that we thought was the cities (Wettel; Lanza, 2008). The problem is that, as in the cities, they grow too fast and create a complex visualization to handle. The second analogy was

with the concentric waves. We thought in this phenomenon because in has the temporal component associated to it (the wave propagation) and uses a radial layout used in some metaphors already existed (e.g. a reference in the field: evolution radar (D'ambros; Lanza; Lungu, 2009)). The problem with the evolution radar was the concepts associated to the metaphor that makes it too specific to the logical coupling domain. We started to explore the concentric waves phenomenon and tested the idea in several domains.

With good results we presented our qualification project based on them and started to work in the prototype to perform a exploratory study in a real open source project. The main goal of the prototype was to better study this phenomenon in our context. Then, we were able to create generic concepts to be able to represent many domains. During the development of the prototype the metaphor concepts was being improved. In the last months of this process, we started to perform a experimental study (Study 1) on the context of software contribution.

This study answered some of the common questions asked by developers during the development process (Fritz; Murphy, 2010). We had interesting results, since we were able to answer all the question that we had data available. This study resulted in a full paper published at ICEIS with the EVOWAVE concepts (Magnavita; Novais; Mendonça, 2015).

During the publication, the prototype was improved to a tool generic enough to display any data that satisfies the metadata specification. A full paper, addressing this novelty, is programed to be submitted in the VISSOFT 2016.

To validate the use of the metaphor in others domains, we performed the studies 2 and 3. The Study 2 is in the library dependency context with tasks related to the comprehension of the library usage and help in the decisions such as the upgrade of some library. The Study 3 did a retrospective analysis using the logical coupling between modules.

1.6 STRUCTURE OF THIS DOCUMENT

The structure of this document was defined in order to provide a easy understanding of this work. Therefore, this dissertation is divided in the following chapters:

- CHAPTER 1 - describes the motivation, goal and scope of this research, as also the structure of this document.
- CHAPTER 2 - presents the literature review used in this dissertation.
- CHAPTER 3 - describes the metaphor concepts in detail and how it was developed.
- CHAPTER 4 - validates the metaphor through exploratory studies for different domains.
- CHAPTER 5 - discusses the contributions, limitations, and future works.

Chapter 2

This chapter discusses the main topics and concepts related to this work. Initially, the concepts of software evolution are presented emphasizing that systems need to evolve to stay alive. Afterwards, we discuss how software evolution impacts on the system complexity and introduce some approaches to help to understand those system. Then, we discuss how visualizations can be used in different scenarios for many purposes, highlighting their use to analyze the software and its evolution.

LITERATURE REVIEW

2.1 SOFTWARE EVOLUTION

Software needs to evolve in order to stay alive (Lehman, 1980). However, before performing any effective maintenance, it is crucial to understand the system (Mayrhauser; Vans, 1995). Software comprehension field provides mechanisms to achieve knowledge about all software aspects (e.g., features, structure, and behaviour). The importance of properly understanding how the software works and evolves is transversal to all software development phases. Software engineers spend more time understanding the software than actually performing software engineering tasks (Corbi, 1989)(Pigoski, 1996). This fact enhances the importance to provide more efficient comprehension techniques.

The growth in software complexity occurs during all its evolution (Lehman; Ramil, 2001)(Jay et al., 2009). Accordingly, software comprehension decreases (Caserta; Zendra, 2010). To prevent this problem, some approaches were developed in the field. For example, design patterns (Gamma et al., 1995) can be used to create well known solutions to certain common problems. This approach will lead to a better understanding of the code because the developers will be using common solutions. However, this approach is still an issue. Programmers do not know or simply do not use those solutions if they are under the pressure of time. Others approaches were proposed in software re-engineering (Briand, 2006). Unfortunately, the effectiveness of those approaches still needs to be improved.

Until the start of the last decade, software maintenance tasks had their participation uniform during the software development process. In 2000, a new model of a software development process was proposed because it had changed drastically since early days (Rajlich; Bennett, 2000). This model, called Staged Model, defines five stages which represent the software life cycle:

- Initial Development: During this phase, an early version of the system is built by the team experts and the system architects. The documentation represents the system and should be built with numerous well known tools and methods.
- Evolution: Engineers extend the capabilities and functionality of the system to meet user's needs, possibly in major ways.
- Servicing: Engineers repair minor defect and make simple functional changes.
- Phaseout: The company decides not to undertake any more servicing, seeking to generate revenue from the system for as long as possible.
- Closedown: The company withdraws the system from the market and directs users to a replacement system, if one exists.

Each one of those states is composed by activities. The elicitation of requirements, encoding and tests are examples of such activities. They differ depending on companies and the processes they use. Each activity generates different software artifacts (e.g., source code, requirements and architectural documents, test-cases). Most of these artifacts are textual data which are hard to be analyzed.

Other issue is related to high level artifacts. The software are not always updated during its evolution (Lethbridge; Singer; Forward, 2003). In other words, while software source code evolves, its documentation may not. As a consequence, current team developers are the ones who keep the knowledge of the software. Unfortunately, team developers may change frequently. This leads to a major problem: new developers have a hard task on the software understanding. This is much harder when considered the large amount of data to be analyzed as the software evolves.

2.2 SOFTWARE COMPREHENSION

Software knowledge can be divided into two types: independent and specific software application knowledge (Mayrhauser; Vans, 1995). The first one is acquired by participating in different software projects. Developers holds generic knowledge about software development such as design patterns and algorithms. The second one is acquired during the understanding process when working on a specific software project. It holds specific knowledge about the software business rules under development.

Various theories were proposed to explain how developers achieve knowledge about a software (Shneiderman; Mayer, 1979)(Letovsky, 1986)(Brooks, 1983)(Soloway; Adelson; Ehrlich, 1988)(Pennington, 1987). They can be classified within two categories: bottom-up, top-down. The bottom-up approach suggests that software knowledge is constructed from low levels of abstraction (e.g. source code) and as the programmer understands the code he creates a more abstract, high level mental model of the software. The top-down approach describes that the metal model is firstly created by understanding the domain of the software and its high-level artifacts (e.g., requirements, architectural documentation). Then, the programmer gets deeper into software details such as languages, technologies, and source code data.

The most reliable source of knowledge is the source code, since the high-level abstraction of a software may not be available in most projects due to process flaws and obsolete documentation (Lethbridge; Singer; Forward, 2003)(Deursen et al., 2004). Unfortunately, the code is the most primitive source of knowledge for most software engineering tasks. It is easier to have techniques and tools for tasks that are intrinsic to syntax or low level semantics (e.g. searching for code anomalies to do refactoring) because they are generally available belonging to the projects. More complex tasks such as the identification of concerns (Brito; Moreira, 2003) are more difficult to be automated due to the low level of semantics available.

Complex tasks require more information than only the source code. Since the information generally does not exist in concrete forms (e.g. requirements and architectural documentations), there is a need for techniques that extract software high-level information. The reengineering domain has conducted a major effort to extract high-level information about the software from the source code (Fuhr et al., 2013)(Detten; Platenius; Becker, 2013) (Fontana; Zanoni, 2011).

Automatic processes to extract high-level mental models is encouraged by the industry since it does not have resources to maintain their documentation and most companies place a high value in problems they are currently facing, than in the consequences of their acts in the future. Tools and techniques have been created to extract information that is automatically generated as software evolves. Such data can be found in repositories such as Source Code Managements (SCM) and Bug Track Systems (BTS). They have reliable information, since they are part of the development process and not from an additional phase that could be easily missed.

To help software engineers understand the high-level abstraction of the software two major areas can be mentioned: Software Metrics and Software Visualization. Software Metrics can be used to characterize the software systems. An example is to subdivide the classes of an object-oriented language into modules according to their coupling. This could help in understanding the location of the main modules that your software can not run without. Software Visualization is being explored as another promising area since it uses visual metaphors to represent different aspects of the software. Various endeavors were implemented by this field in the attempt to represent the software by real world metaphors in order to extract high-level information about the software (Kuhn et al., 2010)(Steinbrückner; Lewerentz, 2010)(Wettel; Lanza, 2008).

2.3 REPRESENTING DATA

Visualization is used in many areas such as mechanical engineering, physics and medicine. Its wide range of applications is due to its facility in representing concrete objects and enabling extraction of valuable information just by looking at it. We need to deal with an enormous amount of information regarding problems we face every day. Currently it is practically impossible to handle such a huge amount of information without techniques and mechanisms to help us to get only helpful information.

Visualization is the most appropriate way that humans have to extract information from a set of data. As humans perceive visual attributes easier (Ware, 2004), we can

represent different data by mapping its real attributes to visual attributes (Mazza, 2009). For example, by replacing a set of textual numeric values with a set of bars with different widths, we can quickly extract the minimum and maximum number, as well as repeated numbers or oscillation patterns within the numbers. Notice that the initial function of the numbers still remains since we can realize which number is bigger or smaller. Every visual attribute mapped needs a legend to explain what it is mapping and the information it represents. The primary goal of visualization is to help extract more complex information about raw data. This is possible because the human brain is always processing a huge amount of data which can lead us to extract the maximum and minimum values, the existence of relationships, grouping, trends, gaps, or interesting values (Mazza, 2009).

Visual representation (i.e., visualization) can be used in different scenarios for many purposes. The main cases where visual representations can be used are: Presentation, Explorative Analysis, Confirmative Analysis, Scientific Visualization, Information Visualization, Software Visualization.

We can use pictures that will help the receptor to go through the explanation without getting lost. When visualization is needed to explain ideas that are too complex to put in words. That is the scenario of a presentation. In this case we are using visual elements as a communication channel to express concepts, ideas. This is really difficult because different receptors can make different interpretations about the message. That is why visual representation needs to be idealized with clarity, precision, and efficiency (Tufte, 1986).

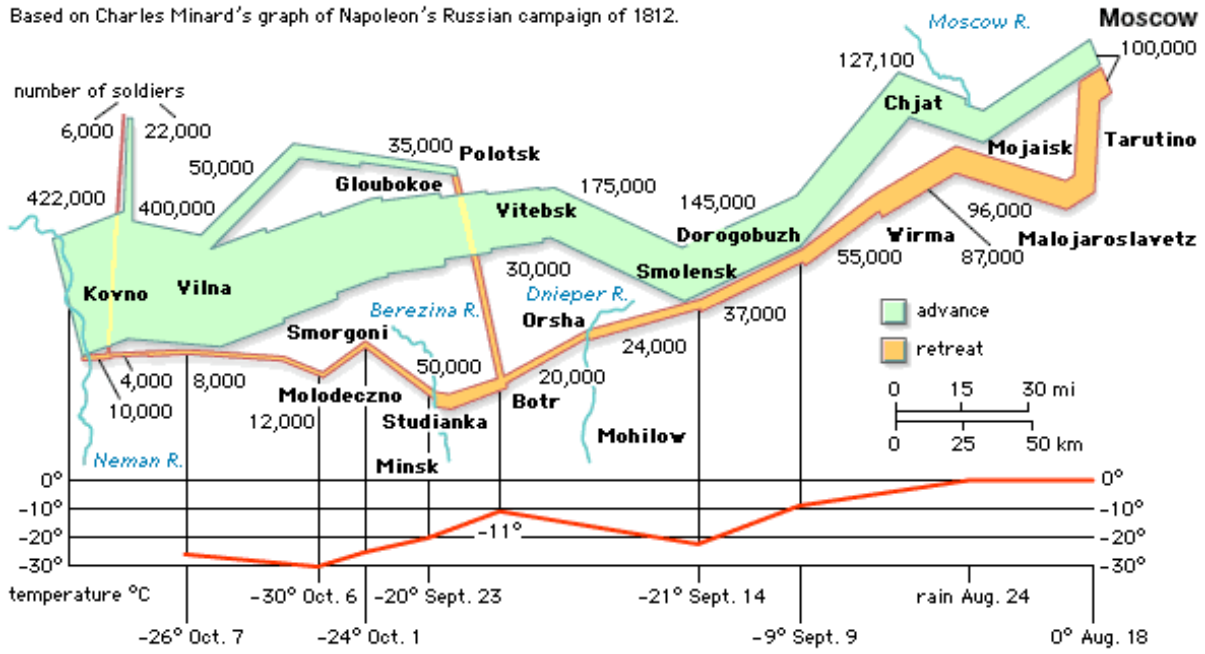


Figure 2.1 The march of Napoleon's army into Moscow at the Russian Campaign of 1912 (Encyclopaedia-britannica-online, 2013).

Figure 2.1 is an example of a presentation. It is possible to recovery a huge amount

of information in this picture. The size of Napoleon's army is expressed by the width of the green and orange lines, respectively, marching to Moscow and on their way back. At the bottom of the image there is a statistical graphic representing the variation in temperature during their retreat. It is possible to see that the number of soldiers during the retreat has drastically reduced from 100,000 to 4,000 (approximately). One of the factors that contributed to their death was the drop in temperature from 18 degrees to -26 degrees reaching -30 degrees at some points.

When we have a lot of information about some subject it is important to try understanding what else is there that we can not see just by looking at the data. That is the case of explorative analysis, visual representations used together with our ability of analysis to identify properties, relationships, patterns, and regularities.

Figure 2.2 is a visualization of friendships around the world according to Facebook data and can help us to do an explorative analysis. Each line connects two cities and its colour depends on the Euclidean distance and the number of friends between them. The initial purpose of this visual representation was to see how geographic and political borders affected where people lived relative to their friends. It is noticeable that the most connected places in the world are in Europe and North America and that each blue line might represent a friendship made while travelling, a family member abroad, or an old colleague or friend pulled away by the various forces of life (Facebook, 2013). It is possible to explore some aspects of this image such as why the Russian country is wiped off the map or why the concentration of connections in Brazil is in the south.

Figure 2.2 Facebook data about friendship connectivity around the world (Facebook, 2013).

Unlike explorative analysis, there are situations where hypotheses have already been raised and require confirmation. The confirmation analysis uses visual representations to quickly confirm these questions without the need to look at a lot of data and sometimes complex formulas.

There are different types of data to be displayed as visual elements. They can be divided into two groups: abstract and concrete data. The first is data without correspondence with physical space. The effect of the temperature on the Napoleon's army during the retreat is an example of abstract data. There is no physical entity related to this information. The second is data with well structured shapes such as mathematical formulas and three-dimensional phenomena with real physical shape (e.g., the rain).

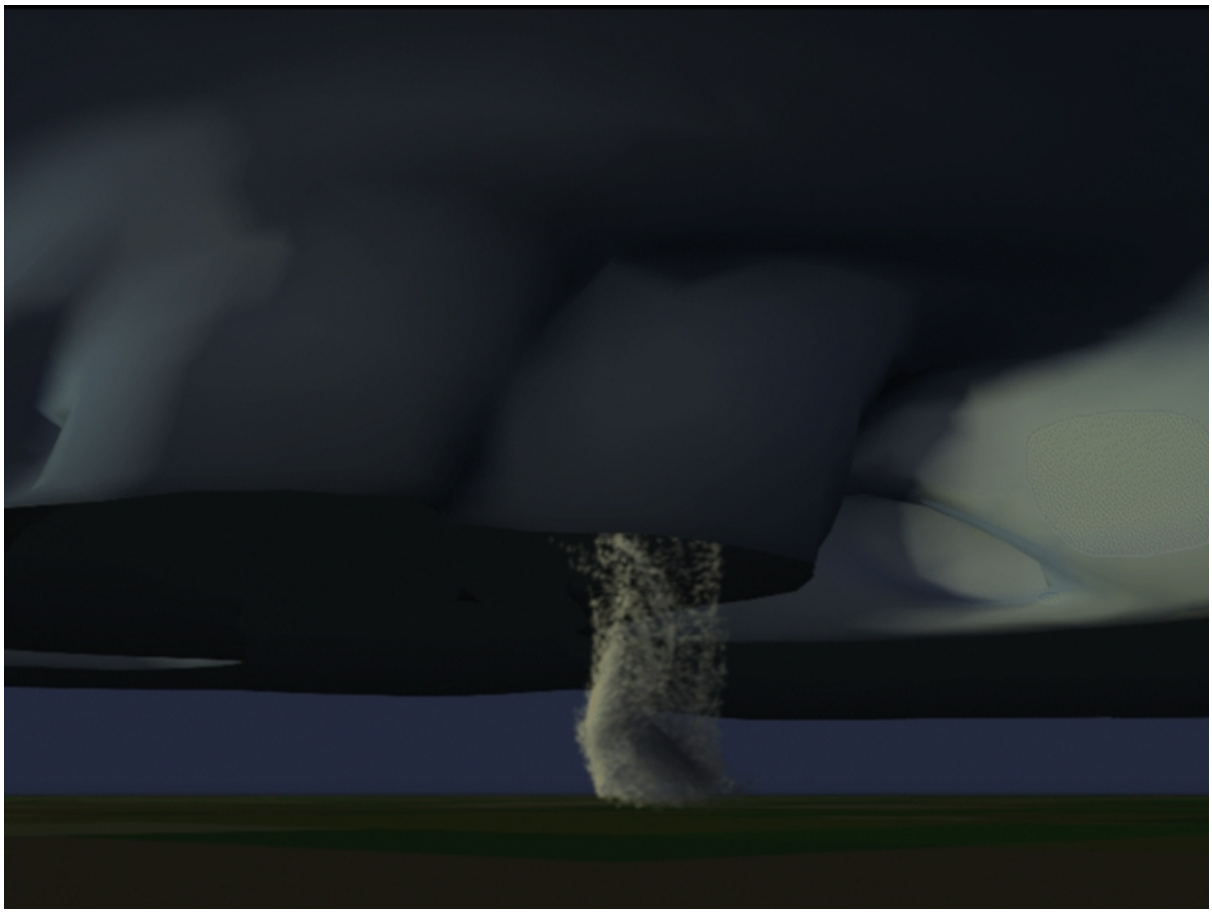


Figure 2.3 Visualization of a developing tornado displayed using thousands of circulating particles (Ncsa, 2013).

Scientific visualization allows scientists to visualize concrete data offering a realistic representation of some elements that are being studied. The visualization of any kind of flow has been an important and active research subject for many years (Johnson; Hansen, 2004). A lot of data is generated from simulators that calculate the flow dynamics, and analyzed using scientific visualization to provide an explanation for the flow. Figure 2.3 is an example of scientific visualization. In the film Stormchasers (Nova/wgbh, 1995), OMNIMAX theaters simulated a tornado for approximately ten minutes. It had relative wind speeds over 60m/s (134 mph) near the ground, and a 40 millibar pressure drop (Wilhelmson, 1996). Approximately 40GB of data was produced in this simulation and we can see it as an actually physical phenomenon through visualization in Figure 2.3.

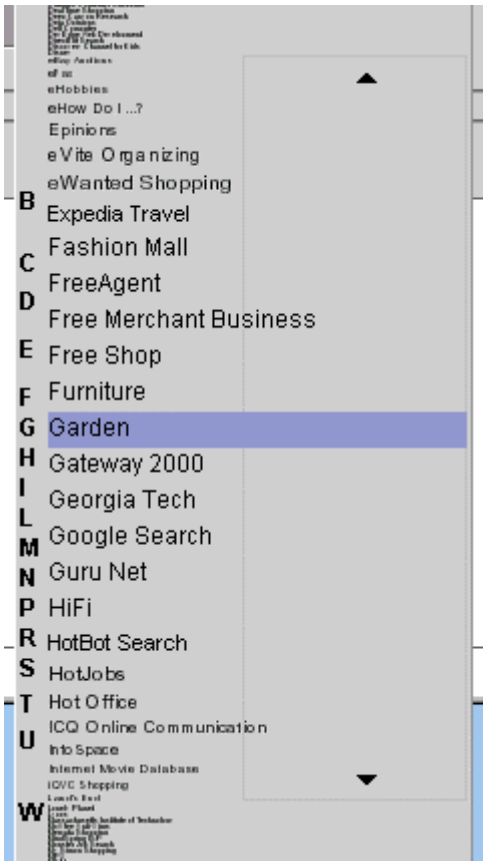


Figure 2.4 FishEye menu showing 100 web sites at the same time. Adapted from (Bederson, 2000).

Information visualization can visualize abstract data that can be generated, calculated, or found in many ways, such as data from common searches, data that affects the result of a soccer match and data displaying global climate changes. This kind of visualization, for abstract data, is very difficult to work with. Since abstract data does not have a physical shape or a human format convention, visualizations are built as a metaphor for some well know representations. Fish Eye (Furnas, 1986) is an example of information visualization and is used to explore detailed data without loosing the global context. How a fish would see an ultra-wide hemispherical view from beneath the water (PHILOSOPHICAL..., 1906) was the real world inspiration to build this visualization. It uses a real phenomenon to represent abstract data, that is how information visualization works. Figure 2.4 shows an example of the use of FishEye visualization in menus that have a lot of sub-items.

2.4 SOFTWARE VISUALIZATION

Software visualization is a subarea in information visualization which visualizes abstract data generated in the software development process. In this area, scientists are concerned with visualizing the structure, behaviour, and evolution of the software (Diehl, 2007). The

structure are all artifacts that were generated statically during the software development process. Source code, requirements, and test cases are examples of structures. Behaviour refers to software behaviour during its execution. An example could be the allocation of memory and resources or higher level information such as function calls. Finally, software evolution refers to static and dynamic information generated during the software evolution process. In this work, we use software visualization aiming to provide a new visual metaphor in order to investigate questions such as: How the quality of the software changes during the software development process? What bugs appears more often and in which parts of the system? The use of visualization to analyze the software evolution will be thoroughly described in the next section.

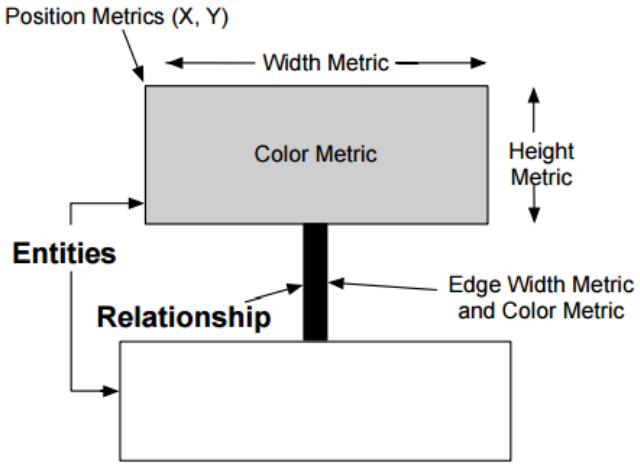


Figure 2.5 The principles of a polymetric view (Lanza, 2004).

In 1999, Lanza introduced the concept of polymetric views (Lanza, 1999) to visualize the software structure. With his tool, the CodeCrawler (Lanza, 2004), he was able to support the reverse engineering of software systems by visualizing the system structure with their relationships extracting this information from the source code. The polymetric views are simple interactive graphs, enriched by various software metrics (Figure 2.5). It is composed by different rectangles each one representing a different software entity (e.g., classes or packages). Its possible to associate different entity metrics to the position of the rectangle, its width, height and color. The rectangles are connected by lines that represent relationships between entities. Its possible to associate different metrics related to the relationship through the width and color of the line. If the polymetric entities are classes and their relationships are the inheritance between classes, its possible to see one perspective of the software structure. Lanza validated the effectiveness of the CodeCrawler by attempting to reverse engineer an industrial system in few days. The successful result is another proof of the capabilities software visualization has to support software comprehension.

Still in the 90s, Staples proposed a tridimensional exploration to visualize the software structure (Staples; Bieman, 1999). The approach, named Change Impact Viewer (CIV), is a tridimensional matrix where the base (x and z axis) represents the system classes and

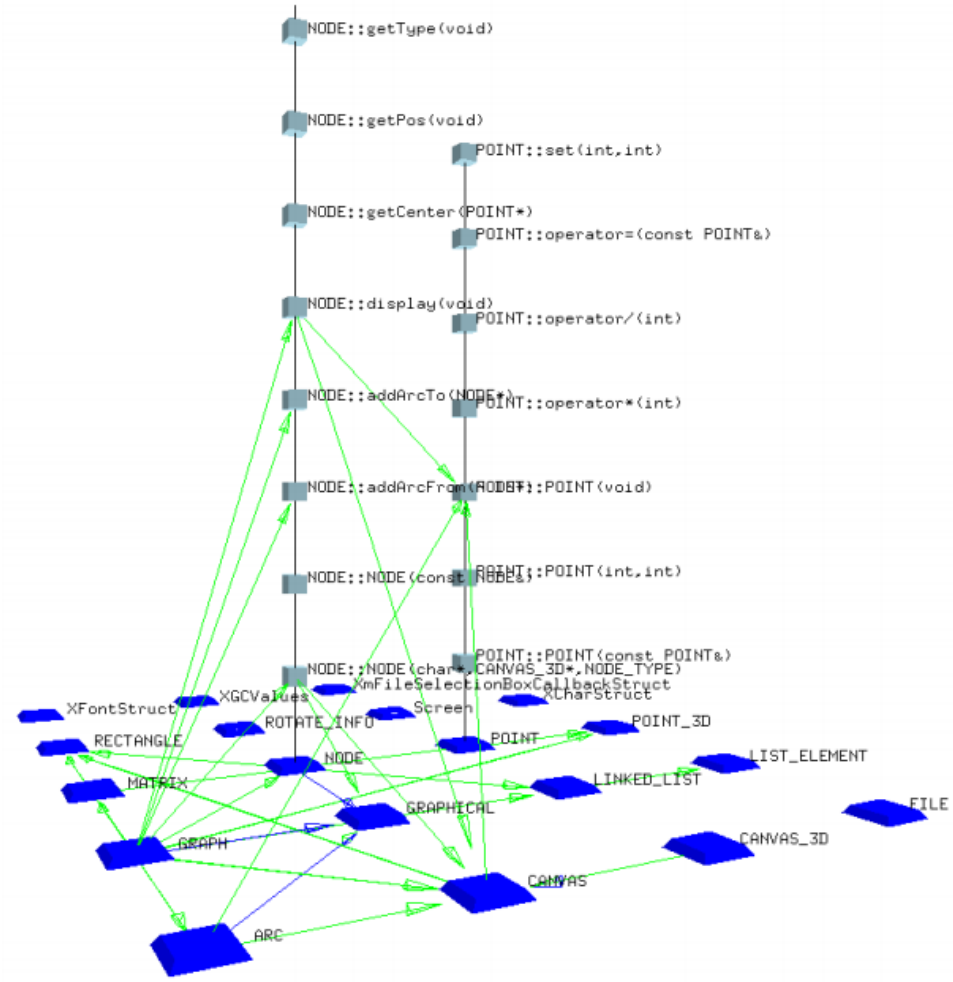


Figure 2.6 Inter-class view displaying a call-graph with two classes expanded (Staples; Bieman, 1999).

the height (y axis) are the functions implemented by the class (Figure 2.6). The arrow between the classes indicates function calls.

Most of the previous works did not take full advantage of the third dimension, but later on another works started to emerge with new approaches (Teyseyre; Campo, 2009). The CodeCity can be highlighted as one of the new 3D visualizations (Wettel; Lanza, 2008). It represents the system as a 3D interactive urban environment. The city provides an overview of the system's structural organization by drawing the classes as buildings and the packages as districts (Figure 2.7). The width of the buildings represents the number of attributes the classes have, and the height represents the number of methods. With the visualization it is possible to identify some patterns such as massive buildings (potential god classes (Riel, 1996)) and some antenna-shaped constructions (potential bean classes). The visualization can offer consistent location and solid orientation points for the user.

One important challenge in visualizing the software behaviour is the scalability. The

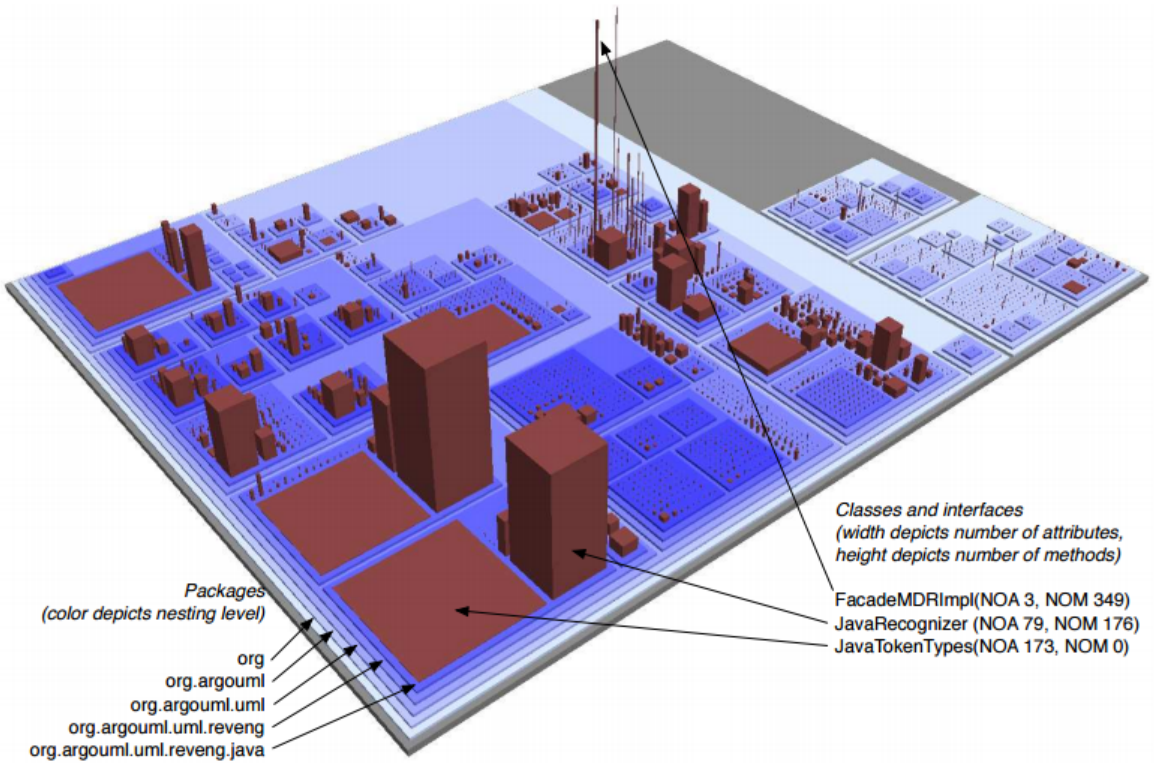


Figure 2.7 An overview of the city of ArgoUML v.0.24 (Wettel; Lanza, 2008).

execution of the system escalates really fast, the amount of data that needs to be manipulated is huge. However, due to the increasing computational power available today, its possible to see many works addressing this area.

In 2007, a tool to visualize execution traces in order to support program comprehension during software maintenance tasks was proposed (Cornelissen et al., 2007). The approach, named Extravis, presents two synchronized views: a circular view and a massive sequence view (Figure 2.8). The first one shows the system's structural decomposition and the nature of its interactions during the trace. The sectors represent the system entities (e.g. modules and classes). It is possible to have a hierarchic for them down to the system's functions. Everytime a function calls another function a line is drawn between the sectors. Colors can be used to represent the direction of the call or if the call was least recent or most recent. The second view provides a concise and navigable overview of the consecutive calls between the system's elements (e.g. classes and methods) in a chronological order. It works like a trace history that can be used for a more detailed analysis.

The synergies and dualities of the structural and behaviour approaches have been recognized (Ernst, 2003). Therefore, works combining these two approaches to extract information about the software are relevant. In 2006, a 3D visualization metaphor to support the animation of the behavior of features was proposed. The author integrates

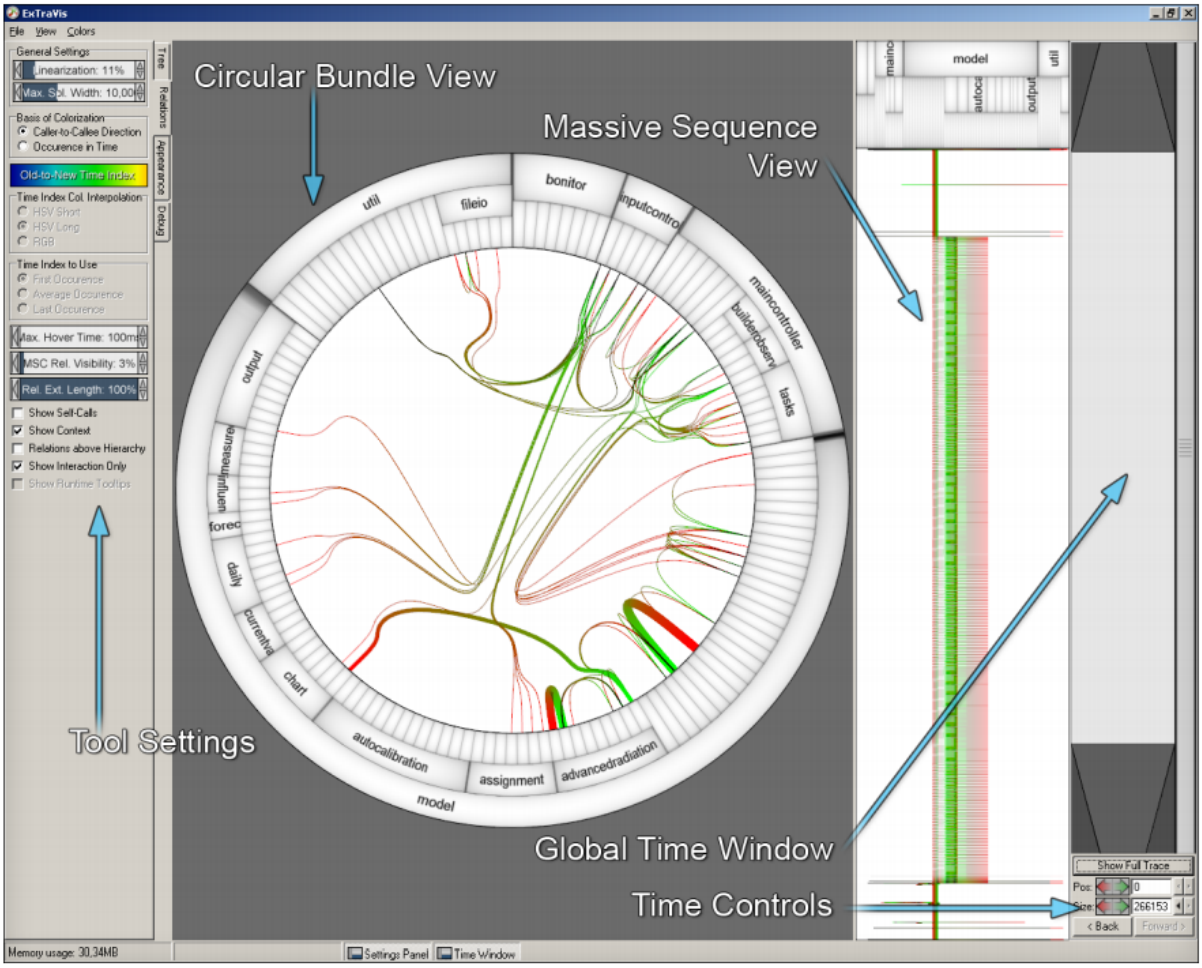


Figure 2.8 Full view of the entire Cromod system trace (Cornelissen et al., 2007).

zooming, panning, rotating and on-demand techniques to support the usability of the visualization (Greevy; Lanza; Wysseier, 2006). They visually represent the dynamic behavior of features in the context of a static structural view of the system (Figure 2.9). They first apply static analysis to the source code of a system to extract a static model of the source code entities. Then, they create a 3D visualization with the same principals proposed in the polymetric views (Lanza, 1999) with the static model. The classes are represented by the gray boxes and the inheritance by the black lines between them. The width, length and color of the boxes can be associated to metrics. The next step is to apply dynamic analysis to obtain the stack traces of the executions. Then, they use these analysis to create more boxes on top of the classes. The number of boxes created on top of the class is related to the number of instances the class has created in the moment of analysis. At the end, the instance boxes are connected by red edges. Each one meaning that a message (or function call) was made between instances.

In 2008, an approach was proposed to use multiple views of the software that can be configured and combined according to the particular needs of the user to support specific

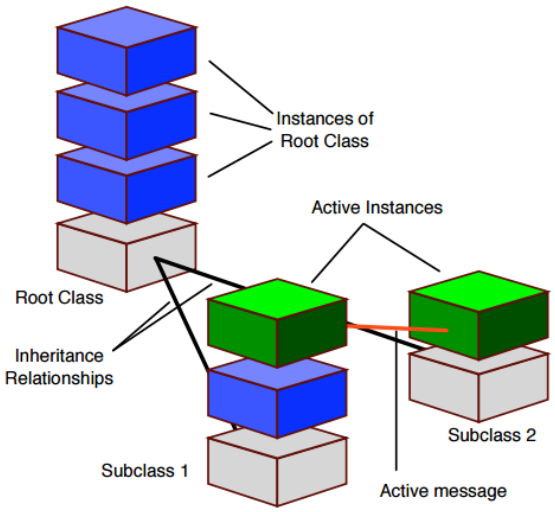


Figure 2.9 A schematic view of the 3D visualization (Greevy; Lanza; Wysseier, 2006).

software comprehension activities (Carneiro; Magnavita; Mendonça, 2008). Later, in 2010 this approach was used to enrich four categories of code views with concern properties: a) concerns package-class-method structure; b) concerns inheritance-wise structure; c) concern dependency, and d) concern dependency weight (Carneiro et al., 2010). Figure 2.10 illustrates all four views. The first one (a) is related to the structural representation of the packages, classes and methods. It was developed based on the Treemap view (Shneiderman, 1992). The area of the rectangles can be configured to express the lines of code or complexity of the methods they represent. The rectangle color was used to represent methods that are affected by a specific concern. With this view it is possible to identify the parts of the system structure that are related to a specific concern. The second one (b) is based on the polymetric view (Lanza, 1999), therefore, it is used to visualize the software structure on the first one. However, this view has the capacity to represent module hierarchy realized by the use of both class and interface inheritances. The width of the rectangle is associated to the number of methods in the class and the height the lines of code. The color is used to mark which classes and interfaces satisfy a concern. The third one (c) represents the dependencies among software packages and classes using a graph-view. This view uses graph-nodes (small circles) to represent packages and classes, and arrows to represent their dependency relationship. The color also is used to mark which packages and classes are affected by a specific concern. The last view (d) shows the weight of each dependency between the packages and classes. This view uses two metaphors: a graph and a chess board. The graph view is similar to (c), however, it highlights a package or class in the middle and shows all dependencies between it and the rest of the nodes. The chess board view plots classes from the entire system as rectangles in multiple rows arranged in decreasing order of the total dependency weight. Both use the color of the nodes and rectangles to identify which concern they satisfy. For all the four views it is possible to filter the information on it. The entities that do not satisfy the search query are colored in white. They used these four views to identify code

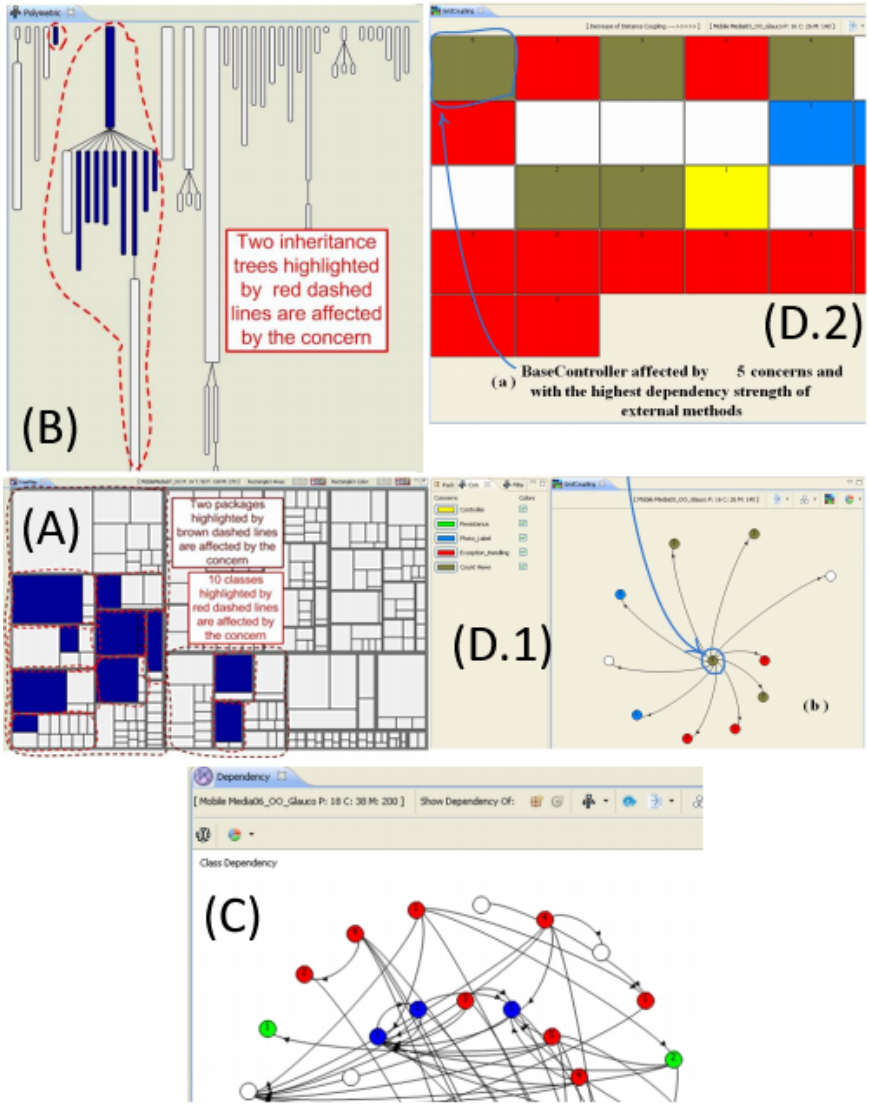


Figure 2.10 All four views used to identify code smells. Adapted from (Carneiro et al., 2010).

smells in software systems. During their studies they discovered seven observations that can be used to derive hypotheses about the support of the multiple views approach to characterize programs. Later on, those hypotheses were proved by a thesis defending the usefulness of multiple views during software comprehension activities (Carneiro, 2011).

2.5 SOFTWARE EVOLUTION VISUALIZATION

The use of visualizations to represent software evolution is well justified because of the amount of data generated during the software development process. All these data generated from many sources such as version control systems, emails and technical meetings is too much to be analyzed using only textual techniques. Visualizations can reduce the complexity of analysis and help to understand some aspects of software evolution such as:

structural decay (Beyer; Hassan, 2006), architectural changes (Godfrey; Tu, 2001), software dependency evolution (Kula et al., 2014).

The use of visualizations to help analyzing software evolution is not something new. There are works dated 20 years ago, as reported in the systematic mapping study (Novais et al., 2013). However, there have been an increasing number of works during the last years.

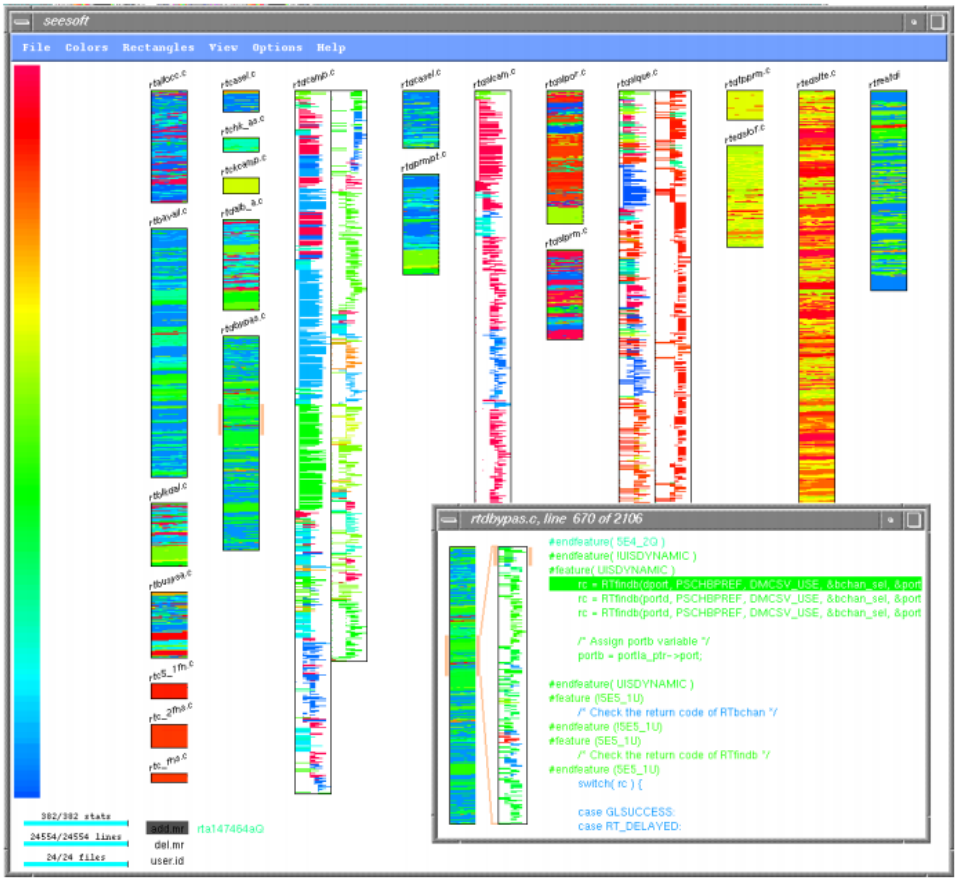


Figure 2.11 SeeSoft showing the code age (Ball; Eick, 1996).

One of the first works in this area was successfully applied in several contexts, each one with a different perspective on software, e.g., static properties, performance profiles, and version histories (i.e., evolution) (Ball; Eick, 1996). That work created the SeeSoft tool which became one of the most well known software visualization tool. Several features of the SeeSoft metaphor assured its success and usefulness. One of the most important features is the natural and direct mapping from the visual metaphor to the source code and the other way around. This leads to natural navigation between representations. It uses pixel-oriented colorful paradigm to represent relationships between software elements rather than graph-based representations. Figure 2.11 has an example of the use of text, line, and pixel representations to analyze the code age. The newest lines are shown in red, the oldest in blue, with a gradient color in between. The browser (lower-right in the

Figure 2.11) incorporates all three representations at once.

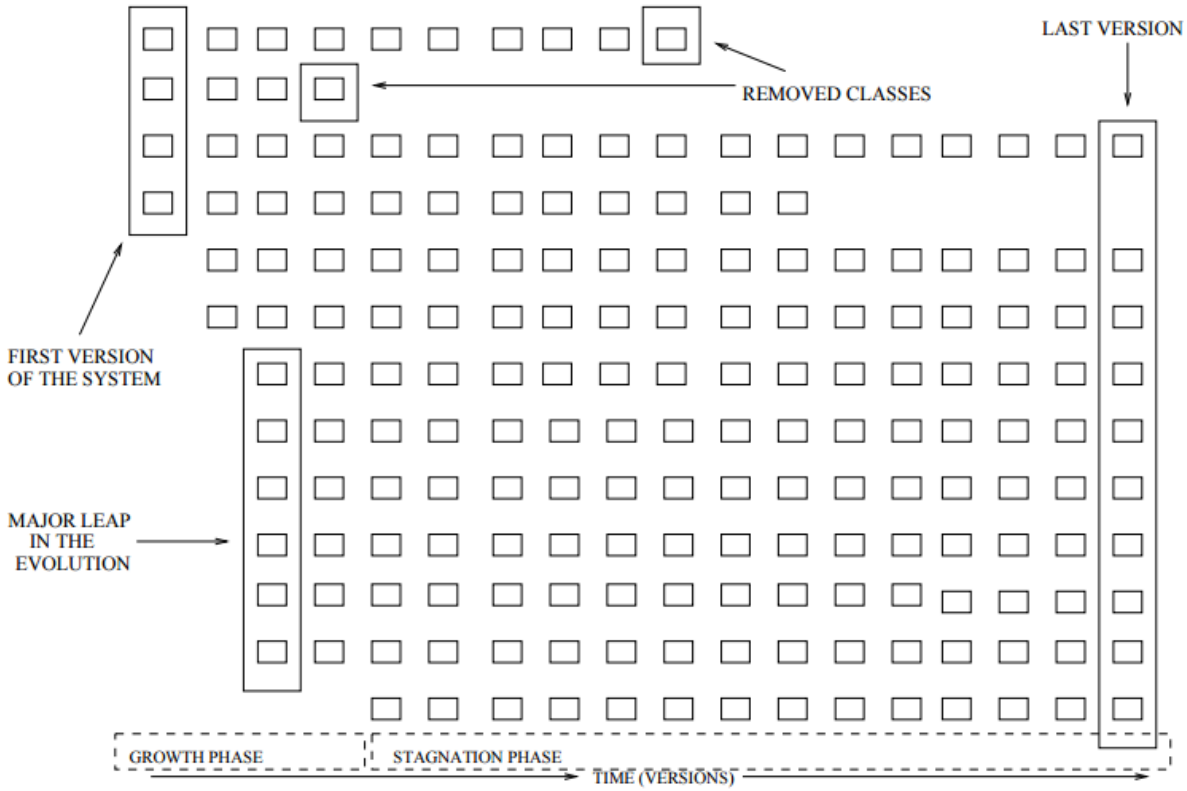


Figure 2.12 The Evolution Matrix and some characteristics about it (Lanza, 2001).

In 2001, Lanza proposed the use of software visualization techniques to recover the software evolution (Lanza, 2001). This approach was named Evolution Matrix because it uses a matrix layout to represent versions (columns) and classes (rows) (Figure 2.12). In each position there is a bi-dimensional box which represents the class (row) in a given version (column). The width and height of the box are associated to class-related metrics such as: lines of code (LOC), number of methods (NOM). They used this metaphor to understand how the class metrics changes during the versions. They found out some patterns in software evolution and used an astronomy analogy for most of them. An example is the pulsar pattern which classifies all classes that grow and shrink repeatedly during the software development. Another example could be the supernova pattern which is a class that suddenly explodes in size. In 10 years, this work became the most referenced paper inside the community of software evolution visualization (Novais et al., 2013).

Several works are starting to use the third dimension in their visualizations to express more information about the software evolution. One of the reasons is the evolution of user interaction techniques besides the common mouse-keyboard pair. SkyscapAR uses a tridimensional visualization with augmented reality to visualize the software evolution (Souza rodrigo; Manoel, 2012). It reuses the CodeCity algorithm (Wettel; Lanza, 2008) that represents a version of the software, and amplifies its functionality to visualize the software

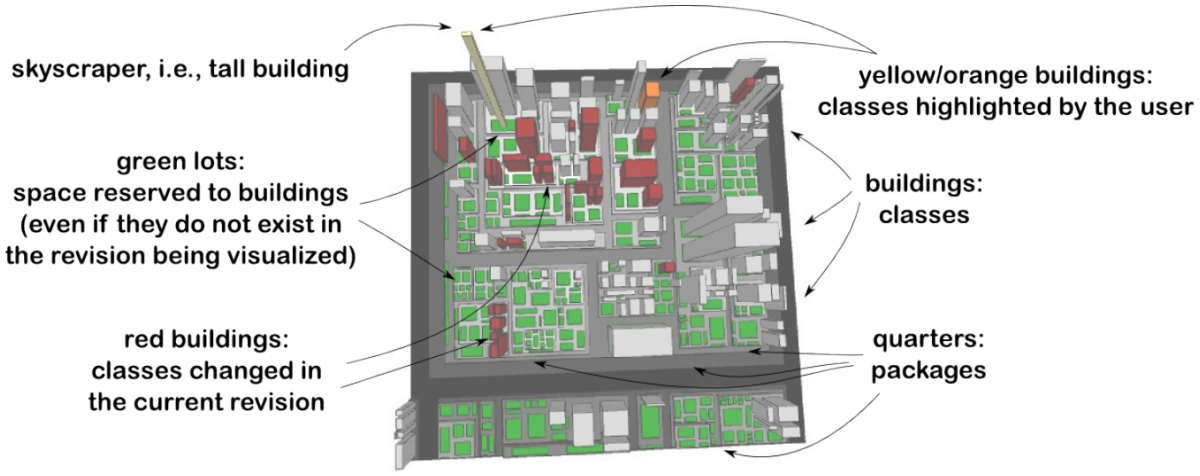


Figure 2.13 The JUnit framework visualized as a city using SkyscrapAR (Souza rodrigo; Manoel, 2012).

evolution (Figure 2.13). Similarly to CodeCity, SkyscrapAR represents software packages as rectangular city lots and on top of them the sub-packages are added. Classes are represented by buildings (boxes with different areas and heights) located on top of their respective packages. The area covered by them is proportional to the number of lines of code of the class. The tool is able to visualize only one version at a time. Therefore, they set the terrain big enough to support the largest size the class had, and paint it green. Thus, the user can compare the size of the class in the current version of analysis against its largest size. The tool paints red the buildings that were changed, also comparing the last version to the current version of analysis. They use a piece of paper or any other object with a predefined black and white square pattern printed on it as a marker. This associated to a camera are able to put the visualization on top of this marker with augmented reality. The application of this tool could be its use in code review meetings and module inspections with a team effort to identify design flaws.

A more recent work was presented by (Rufiange; Melancon, 2014). In this work, the authors proposed a software evolution matrix-based visualization named AniMatrix. They used a taxonomy of dynamic graph visualization to organize the evolving graph network having multiple types of nodes and edges, focusing on both node-link and matrix views (Figure 2.14). They associated the nodes with different types of classes (e.g., normal class, abstract class, and interface) and the edges with different relationships (e.g., declarations, extensions, and constructor calls). If there are many relationships between two classes they subdivide the position into many squares according to the number of relationships (Figure 2.15). The visualization only shows one version of the system at a time but use colors inside the squares to represent state transitions.

If there is a case of a new relationship, the square will be filled with a gradient color from green to gray. Otherwise, if a relationship ended, the square will be filled with a gradient color from gray to red. Otherwise, if a relationship was modified, the square will be filled with the color blue. Otherwise, if the relationship already exists in previous

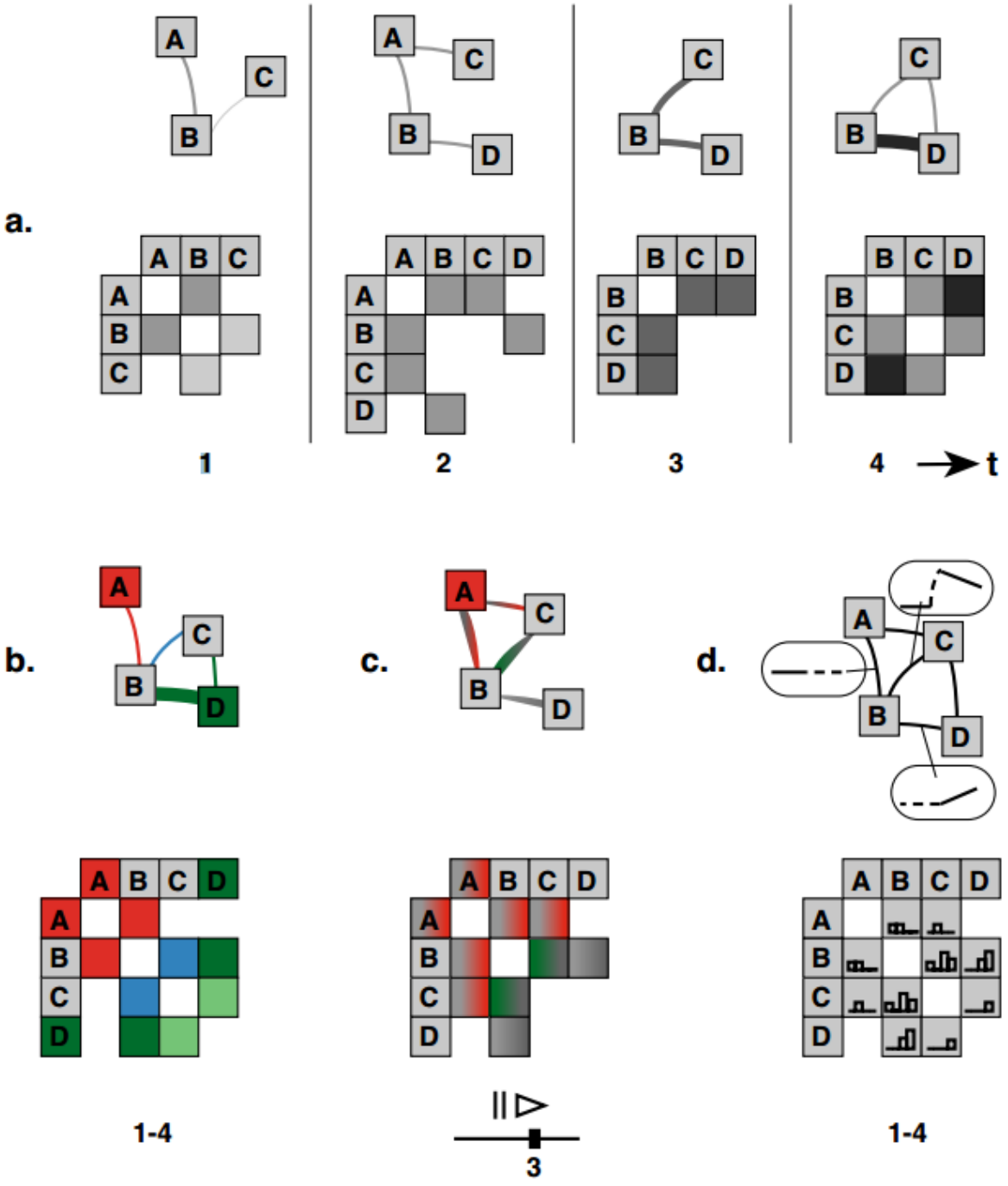


Figure 2.14 A taxonomy of visualization techniques for dynamic graphs. (Rufiange; Melancon, 2014).

versions and nothing happened, the square will be filled with the color gray. The intensity of the gray color is used according to the weight of the relationship. They track the weight change of the relationship by filling the square with a gradient color from the previous

tone to the current one.

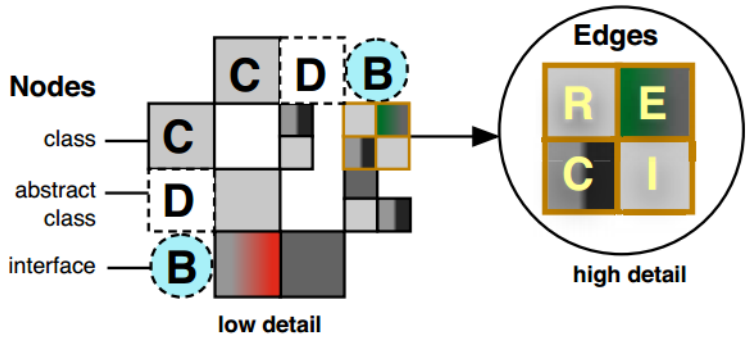


Figure 2.15 Staged animations used in the matrix-based visualization can show changes concerning multiple types of nodes and edges. (Rufiange; Melancon, 2014).

Along the years, many works in software evolution visualization were proposed. According to the systematic mapping study (Novais et al., 2013), 146 studies were identified until 2011, and other six studies were identified during this work. Among these works two stand out for being similar to this work. They use a similar layout but their proposal focus on different tasks for a unique domain.

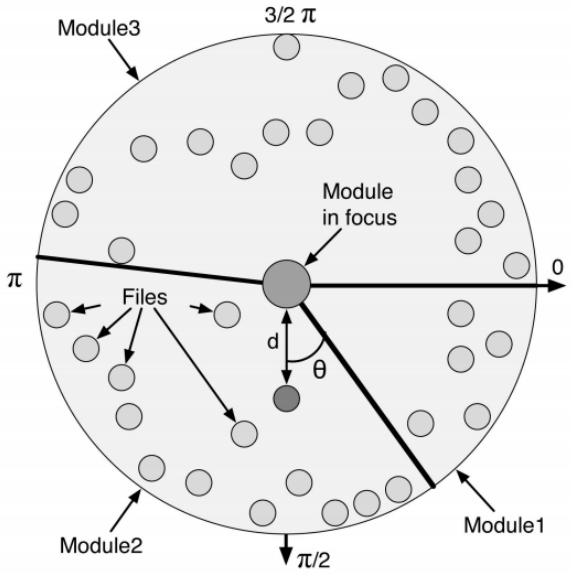


Figure 2.16 Principles of the Evolution Radar (D'ambros; Lanza; Lungu, 2009).

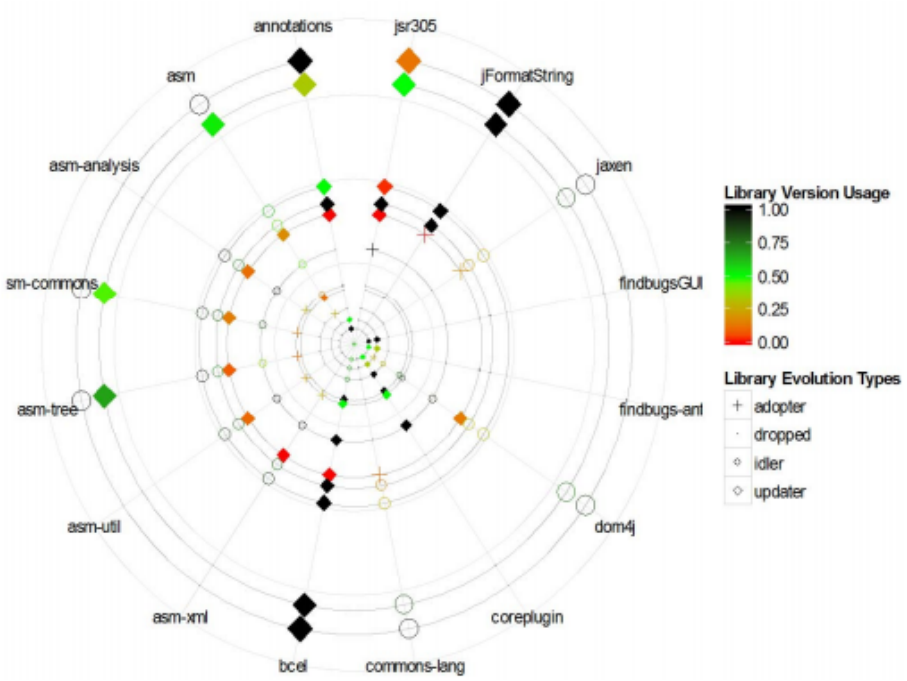
The oldest of these two works was published in 2009. They proposed a visualization-based approach that integrates logical coupling information at different levels of abstraction (D'ambros; Lanza; Lungu, 2009). The work, named Evolution Radar, shows the dependencies among a module in focus and all the other modules of a system. The module in

focus is represented as a circle and placed in the center of a circular surface (Figure 2.16). All of the other modules are visualized as sectors, whose size is proportional to the number of files contained in the corresponding module. The sectors are sorted according to their metric size, and placed in clockwise order. Within each module sector, files belonging to that module are represented as colored circles and positioned using polar coordinates, where the angle and the distance to the center are computed according to their name and their logical coupling, respectively. It is possible to see only one version of the software at a time but users can move through time using a slider. With this visualization, it is possible to track dependency changes detecting files with a strong logical coupling with respect to the last period of time, and then, analyze the coupling in the past allowing us to distinguish between persistent and recent logical couplings.

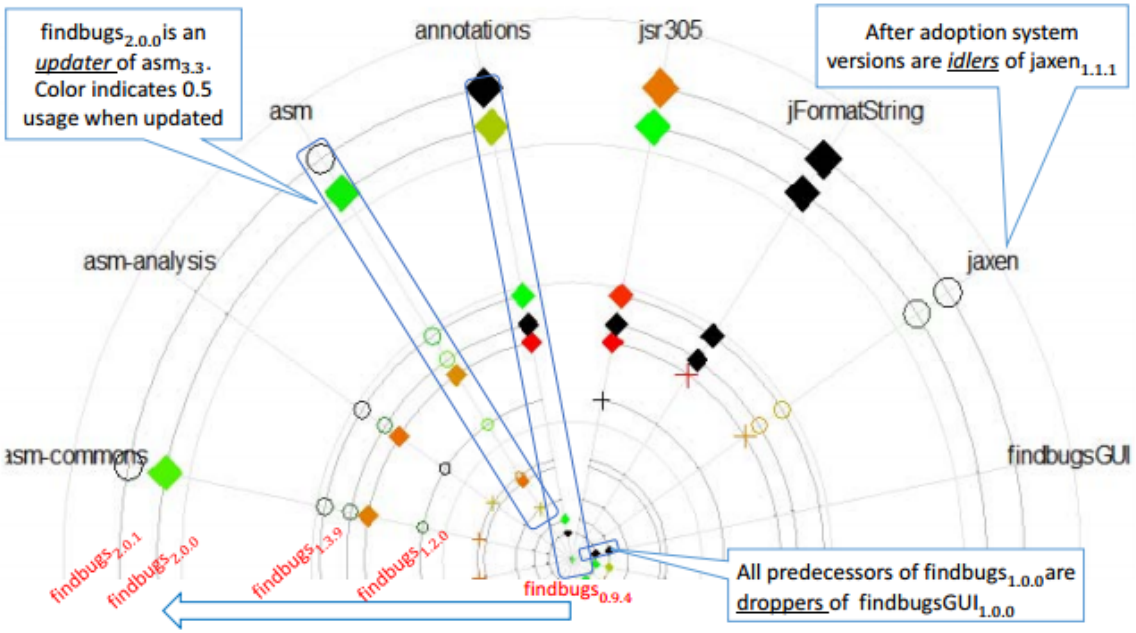
The other work was published in 2014. They proposed to visualize how the dependency relationship in a system and its dependencies evolves from two perspectives (Kula et al., 2014). The first one uses the same radial layout but with different concepts, and includes the use of heat-map to provide visual clues about the change in the library dependencies along with the system's release history. They called a system-centric dependency plots (SDP). It is represented in Figure 2.17. The second one uses statistic graphics to create a time-series visualization that shows the diffusion of users across the different versions of a library, they called a library-centric dependants diffusion plot (LDP). It is represented in Figure 2.18. In the SDP visualization, each axis represents a library that the system depends upon. Starting from the center, each circumference represents a system version released. The time between releases is represented by the distance between circumferences. On each circumference, each of the dependencies used by that version is represented. The shape and color of the dependency represents the type of dependency relationship and the version. In the LDP visualization, the x-axis represents the time-series and the y-axis represents the accumulative sum of system versions per library version. Each point in the graph is plotted using a specific shape to indicate the dependency relationship between that particular system and the library (e.g. adopter, idle, or updater). The color is used to classify the different library versions that are represented by the group of lines that connect the plots.

A recent study characterized the software evolution visualizations in temporal strategies (Novais, 2013). The main difference among them is the approach used by the visualization to visualize the evolution. They could show only one picture representing the changes in the software modules, metrics, and properties as it evolves. Otherwise, they could show only one version (or period of time) of the software, also representing the changes in it, but with more detail. Those strategies are specialized in three: Overview, Snapshot, and Overview Accumulative. The Overview strategy shows information regarding many version at the same time. The Snapshot strategy shows information about a specific version of the system. The Overview Accumulative strategy takes into account the absolute value of the changes in the software properties between versions to analyze the software evolution. The Temporal Overview and Temporal Snapshot are the strategies more frequently used to visualize the software evolution (Novais et al., 2013). However, less then half of the works use a combination of these two strategies.

This work assumes that the use of combined strategies in the same visualization



(a) Overview. According to the library version usage scale, the red indicates 'low usage', green 'mid usage' and black 'high usage' at that point of dependency. Figure 2(b) zooms in on the top-half of the plot



(b) This is a zoomed in cross-section of Figure 2(a). Note that *dropped* libraries are not plotted.

Figure 2.17 SDP for FINDBUGS system. (Kula et al., 2014).

increases the capabilities of the analysis. The use of multiple different visualizations

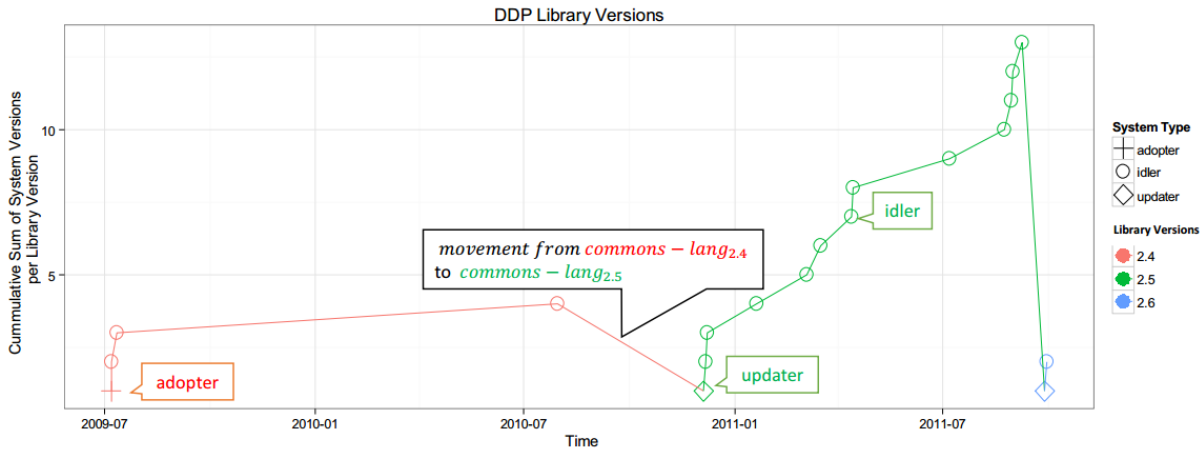


Figure 2.18 Simplified example of a LDP with a single system for the COMMONS-LANG library (Kula et al., 2014).

could decrease the user's learning curve. The user will need to study and have some practice time with each one of the visualizations to be able to better identify aspects of the software evolution. The lack of integration between the visualizations is another problem. The user could easily get lost in years of the data due to the lack of synchronism between approaches.

2.6 CHAPTER CONCLUSION

This chapter presented a literature review about the concepts related to the area covered by this work. It clarifies that the software evolution is an eminent fact which will lead to a more complex system. As a result, the development of techniques and methods to better understand a complex software is not trivial, but very important. It demonstrates that the use of visualization techniques to represent the software has been successfully applied. In the case of the software evolution analysis, visualization has been indicated as a good approach due to the amount of data generated during the software development.

Chapter 3

This chapter discusses the origin of the EVOWAVE metaphor and how it works conceptually and technically to represent evolutionary data for different domains. Initially, we introduce some facts about concentric waves that helped the idealization of the metaphor concepts. Afterwards, we present and describe each one of those concepts highlighting how they can be mapped to software properties. Then, we discuss the algorithms to create a visualization of the metaphor and the metadata that makes it generic enough to represent data from different domains.

THE EVOWAVE METAPHOR

Software evolution generates a huge amount of valuable data from different sources. However, tons of data without analysis tells us little or nothing about the software. This indicates the need for techniques that helps the data analysis through visualizations that efficiently organize the data. For example, it would be interesting to know which software module has been using most of the project resources. Therefore, we need to extract this data in an organized way, visualize it in such a way that users can correctly analyze it and get useful information.

EVOWAVE is a new visualization metaphor that enriches the analysis capabilities of software evolution. It is inspired on concentric waves with the same origin point in a container seen from the top as shown in Figure 3.1. This section presents the facts and concepts related to concentric waves which make EVOWAVE a promising software evolution visualization metaphor. Later, we explain how those concepts can be mapped to software properties and how the tool was implemented.

3.1 CONCENTRIC WAVE FACTS

During this research, we identified some facts about the formation of concentric waves which can be used to represent evolutionary data. Figure 3.1 will help to understand the facts related to the proposed metaphor. Each of them is discussed below.

3.1.1 The concentric wave propagation occurs in all directions

An external force must be applied in a container filled with liquid to generate the propagation of waves. In normal situations, they are distributed equally in all directions from

Figure 3.1 A snapshot of real concentric waves.

the center. This happens when the force direction is 90 degrees to the flat surface of the container. This force pushes the same amount of molecules in all directions which creates redundant information. This is by no means what the metaphor wants to achieve. If delimiters were installed in the container following the propagation path from the center (as a radius line), it creates regions that have no influence on adjacent ones. Thus, an external force could be applied to each region to push different amounts of molecules between each pair of delimiters solving the generation of redundant information.

3.1.2 The biggest waves have more molecules

The magnitude of the applied force responsible for the wave formation, will define how many molecules will be pushed away. During the wave formation the strongest force applied will generate the biggest wave. Thus, it will be the wave with most molecules.

3.1.3 The wave closest to the center is the last formed

Concentric waves are formed from the application of a force at its center that spreads over time. When looking at a snapshot of the wave formation process, the least propagated wave is the one closest to the center. This means that it is the last to be formed. This

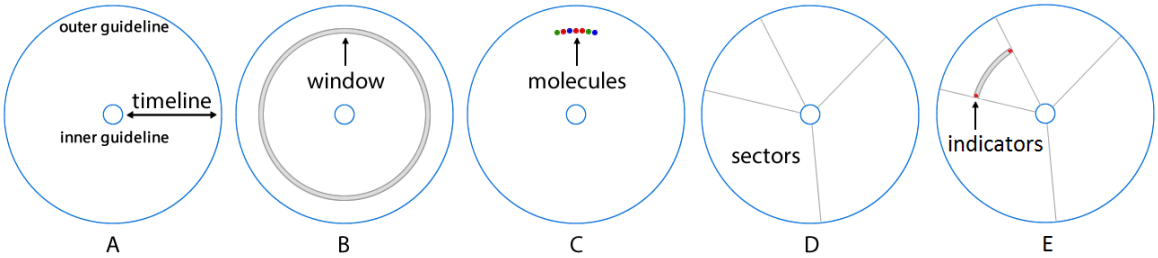


Figure 3.2 The EVOWAVE concepts

leads to a conclusion about the existence of a timeline in the wave propagation path, where the center is when the snapshot was taken and the most propagated wave distance from the center is the beginner. Therefore, each molecule has information about when some force was applied to it according to its location in the propagation path.

3.2 EVOWAVE CONCEPTS

Based on the wave facts, we derived a set of concepts used in the EVOWAVE metaphor. Figure 3.2 represents the concepts, which are explained bellow.

3.2.1 Layout

We observed from the described facts that the wave propagation path has the behavior needed to represent a period of time in software history. EVOWAVE has a circular layout with two circular guidelines (inner and outer), as shown in Figure 3.2-A. They represent a software life cycle period (e.g. [01 January, 2000 10:01:20 AM] to [01 January, 2014 05:20:01 PM]). This period, named timeline (Figure 3.2-A), is comprised by a series of short periods with the same periodicity (e.g., ten days, two hours, one month). The periodicity may differ between visualizations according to the size of the display available. The newest date can be associated with the inner guideline and the oldest date with the outer guideline, or the other way round, to give some orientation to the path between them. The display region between the two circular guidelines contains the timeline used for an overview of the software history for analysis.

3.2.2 Windows

EVOWAVE has a mechanism, named window, which compares a subset of short periods, making it possible to carry out a detailed analysis regarding the overall context. A window (Figure 3.2-B) is a group of consecutive short periods. It is circular in shape and its length depends on the number of grouped periods. The timeline is comprised by these windows, and each one of them has the same number of consecutive short periods.

3.2.3 Molecules

All the data generated from the software development process can be associated to an event that occurred during it (e.g., bug reports, file changes, team changes). Let us consider, for example, a change in class and, therefore, in its package. In this example, the event was the change of a class and its associated data was the package of this class. These events will be organized between the two circular guidelines, in one window, according to when they occurred.

A visual element is provided to actually represent these events because it increases the level of detail provided by the metaphor. Molecules (Figure 3.2-C) are mainly depicted as circular elements inside sectors and windows. Each molecule has an event associated to it. Unfortunately, the display size limits the number of molecules that can be drawn inside the window. When we are unable to draw all the molecules they are gathered and drawn as a quadrilateral polygon that fills the region where the molecules are to be put in.

3.2.4 Sectors

A mechanism to compare different groups of data must be provided to improve the analysis capabilities of software engineer tasks. The concept of a circular sector was used to group events that share some characteristic due to the metaphor's circular layout. A sector (Figure 3.2-D) is a visual element drawn between the two circular guidelines according to its angle. Each sector can have different angles which will result in different areas. The sectors should be one of the first visual element to be realized during analysis. It will clarify how the events are organized. Once we understand how EVOWAVE builds the group of events, we can compare the sectors in two levels: globally or periodically. Globally is used to understand how the events flowed through the timeline more abstractly, while periodically helps to reach a higher level of detail by comparing the same window in different sectors.

The defined characteristic used to group events has, sometimes, an implicit hierarchy. An example would be the java class package (i.e. com.magnavita.evowave) of a changed file event that represents a different level in the hierarchy before each dot. Treemap (Shneiderman, 1992) means the notion of turning a hierarchical structure into a planar space-filling structure. This concept was achieved by drawing a sector inside a sector according to its level in the hierarchy. This renders the capacity to see tens of thousands of nodes from this hierarchy in a fixed space which certainly will be needed due to the amount of levels it could have.

3.2.5 Number of Molecules Indicator

A mechanism to help differentiate the number of molecules in the windows is important when the windows has too many molecules. The number of molecules indicator (Figure 3.2-F) is drawn as a rectangle located in the frontier of the sectors for each window. Its color varies from red to blue, where the reddest indicator has the largest number of molecules and the bluest has lowest number of molecules. The indicator can refer to the

local sector or to all sectors. If set to local, its color will take in consideration the other windows inside the sector. Otherwise, if set to all sectors (global), its color will take in consideration all windows present in the visualization.

3.3 MAPPING SOFTWARE PROPERTIES

EVOWAVE concepts define how the metaphor organizes and displays events which occurred during any general data history. In this sense, the EVOWAVE metaphor is able to represent software evolution, by mapping its visual elements to software history attributes. It is important to take into account that each mapping will give different information and should be chosen according to the software development task at hand. Find bellow the EVOWAVE characteristics (visual attributes) that can be mapped to software history (real) attributes:.

3.3.1 Timeline

The timeline defines the period of analysis through two dates: the beginning and the end of a software development phase. We can map two software versions and analyze what happened between them. If we map the first version to the inner guideline, and the last version to the outer guideline, the history of the software is portrayed from the center to the periphery.

3.3.2 The Pooler of a Sector

A pooler defines how the events will be grouped. The software property chosen to be the pooler has to categorize the events. Events within the same category will be in the same sector. There are many software properties that can be associated to this property. Some examples are: the package of a changed class event; the file type of a changed file event; the author of a bug report event; the bug type of a bug report event. The user might choose the property according to the task goal. For example, if the goal is to analyze developers' efforts, the pooler could be the author and the event could be a file that was changed, created, or removed. Figure 3.3 shows a snapshot the metaphor we set up for this case. In this example, it is possible that the developer mapped to a sector labeled as 'A' was the most active developer at the beginnings of the project. Around half of the project's life cycle, he/she stopped working. Then, from that point on, two other developers (the ones mapped to the sectors labeled as 'B' and 'C') become the main project contributors.

3.3.3 The Splitter of a Sector

The splitter defines how the hierarchy of the pooler's property will be created. The pooler's property usually has some delimiter that can be point out to be the splitter's property. The splitter needs to be part of the pooler in order to split it into different levels. For example, the slash character could be used as a splitter for the file path of a changed file. It is part of the pooler's property and will divide it into many folders

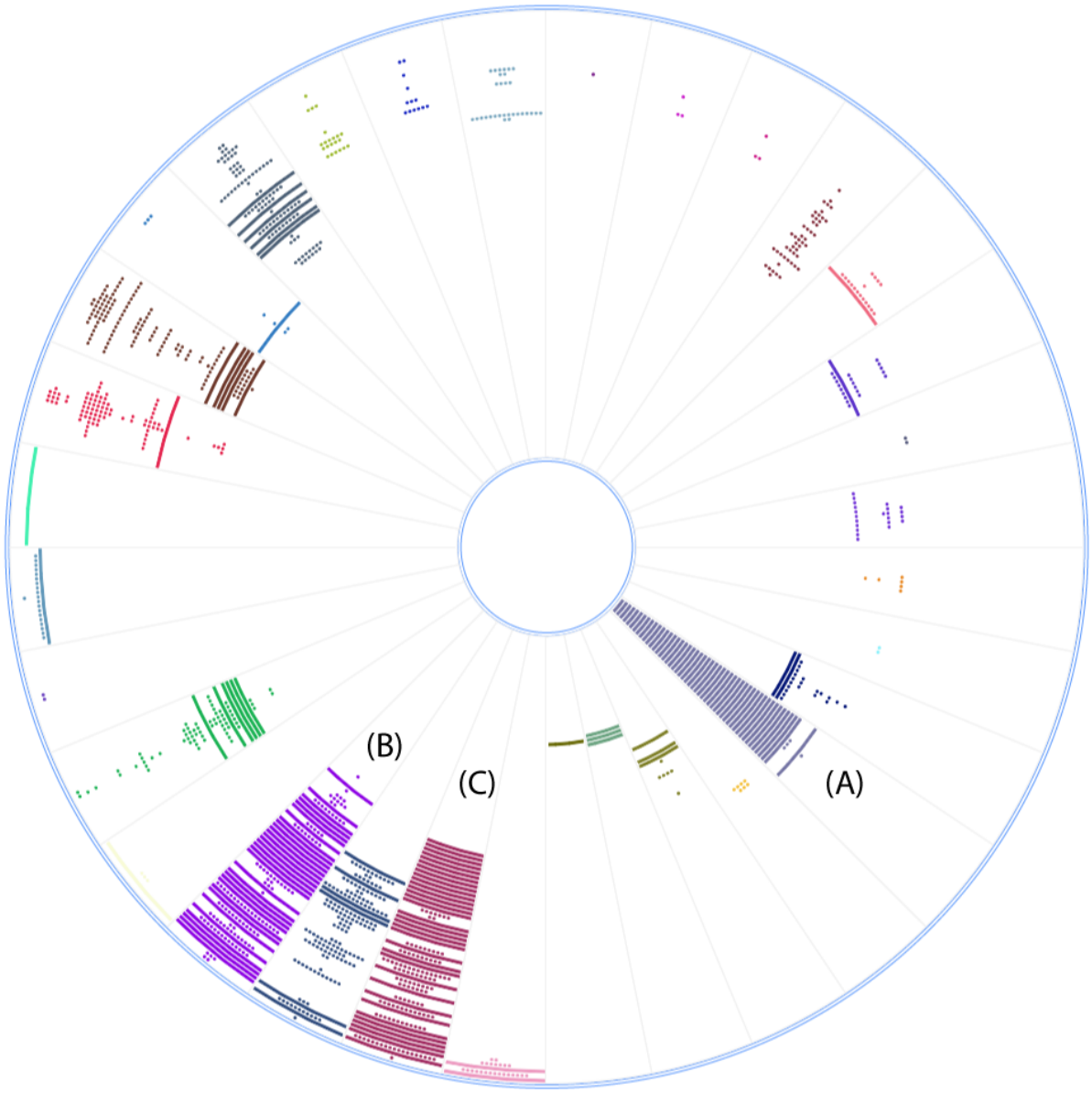


Figure 3.3 The pooler is the authors and the events are any changed files

where each one has a different level in the hierarchy. Figure 3.4 shows an example of the metaphor without (A) and with splitters (B). Again, the decision to use splitters or not depends on the task at hand. For example, if the task is to identify the most active packages, the visualization in Figure 3.4 (A) will achieve the goal faster. Otherwise, if the goal is to identify the big picture of each module, Figure 3.4 (B) gives a clearer view.

3.3.4 The Angle of a Sector

The angle defines how much of some software property the sector has relative to the other sectors. The sum of this software property determines how much bigger it should

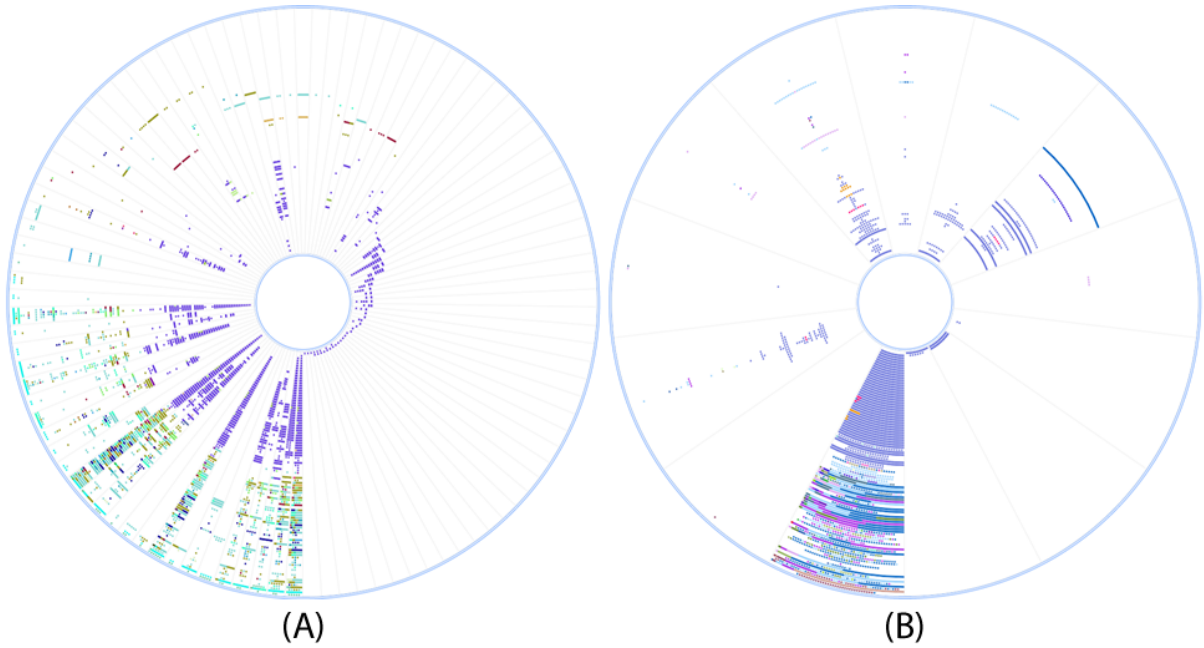


Figure 3.4 The sectors are the java packages. At (A) no splitter was used and all packages are presented at once and at (B) the dot was used as splitter and only one level of the java package hierarchy are displayed at once

be compared to the others. If the pooler is the package of a changed class, the angle could be the increase in the package complexity, for example. In this case, all the events with increased complexities are summed up for each package. The higher complexity is mapped to the larger sector angle.

3.3.5 The Color of a Molecule

The color is another important visual attribute in the EVOWAVE metaphor. It can also be used to map software properties as an event categorization or a numerical property range. An example of the event categorization are the authors of a code change, or of a bug report, where each author could have a different color associated to them. For numerical property range we may consider how much a Java class complexity grew or shrank. In this last case, we can select two specific colors to paint the changed file with the most increased complexity and the most decreased complexity. Any event between these two ones will have its color interpolated.

When there are too many molecules to display, a quadrilateral polygon is drawn and its color can be associated to the number of molecules in it or to the proportion of each color. The second one can be a linear gradient where the amount of each color in it will be related to the number of molecules that have this color.

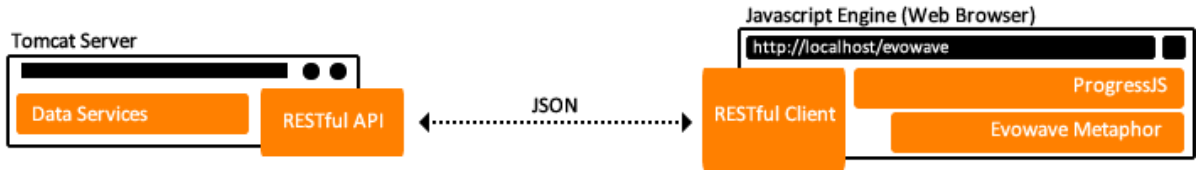


Figure 3.5 The EVOWAVE Tool Architecture

3.4 TOOL IMPLEMENTATION

As we presented before, EVOWAVE is a software evolution visualization metaphor aiming to help software engineers to understand more about the software by looking at its evolution. A tool was developed to validate this metaphor using a MVC (model-view-controller) architecture in a Client-Server environment. Figure 3.5 presents the macro architecture of the tool. The client side uses a well known Javascript framework called ProcessingJS (Processingjs, 2015) which is used to draw the visualization. The metadata used by it is defined in a key-value notation called JSON (Javascript Object Notation). It has all the data needed by the visualization to depict the data following the metaphor's concept. This data is collected by the server according to the period defined. The client will never ask more data from the server as long as the client does not change the period of analysis. The nature of this tool is to consume services that returns all the information needed to perform an action. Thus, in the server's side, a servlet that fulfill the RESTful architecture was used. This architecture was chosen because it uses the Http protocol with the same verbs (GET, POST, DELETE, etc.) as well as its integration with the JSON notation. Every service provided by the server returns the data according to the visualization metadata. The following sections will describe in details how each part of this tool works.

3.4.1 Metadata

The metadata only holds the minimum information necessary to depict the visualization according to the parameters specified. It is generated by a RESTful service in a JSON notation. The JSON structure follows a key-value notation. Each set of key-value pairs is an object and must be between the left bracket "{" and the right bracket "}". The key-value pairs is separated by the colon punctuation (":"), where in the left is the key and in the right is the value (e.g "name" : "John"). The notation specifies arrays of objects as well. The objects must be between the left bracket "[" and the right bracket "]" to represent an array. Following this notation, we illustrate the EVOWAVE metadata structure in Figure 3.6.

At the second line we define the parameters used by the windows in the metaphor. They are two: size and amount. The size specifies the number of circumferences the window will use to draw the molecules. The bigger the size is, the bigger the molecule will be. The amount specifies the number of windows that will be depicted. The next attribute of the metadata defines an array of sectors. Each sector has a parameter specifying the angle of the sector in percentage and has a list of windows associated to

this sector. This list does not need to have all the windows in it, only the windows that have molecules for this sector. The window position defines where the window is located following the orientation where the first window is the closest to the center. At last, the window has a list of molecules associated to it. Each molecule has: i) a hexadecimal color generated by the server according to the parameters specified in the metaphor; ii) a tooltip with a description about the event which the molecule is representing; iii) JSON Object with information about the event that can be used by filters.

```
{
    _windows: {
        size: INTEGER,
        amount: INTEGER
    },
    sectors: [
        {
            angle: FLOAT,
            windows: [
                {
                    position: INTEGER,
                    molecules: [
                        {
                            color: HEX_COLOR,
                            tooltip: STRING,
                            data: {
                                when: DATE,
                                metadata: JSON_OBJECT
                            }
                        }
                    ]
                }
            ]
        }
    ]
}
```

Figure 3.6 The structure of the EVOWAVE metadata

3.4.2 The EVOWAVE Visualization

In this section we explain the main algorithm used on our metaphor development. EVOWAVE uses the immediate mode. This means that every drawing in the canvas is not associated to an object that can be manipulated afterwards. For example, if a red rectangle is drawn on the canvas, there is no object associated to it that could be used to set a different color and draw again with a new color. To achieve that, a new rectangle with the new color needs to be drawn on top of the old rectangle. The retaining mode solves performance

problem, in the sense that it already knows what is drawn (position, size, color, shape, etc.). However, the EVOWAVE would have a lot of objects, one for each molecule at least, and this certainly would consume too much system memory making the system slower. To prevent this, an object manager was implemented to handle and store only the essential data to draw this metaphor. The object manager fulfill the EVOWAVE metaphor with information related to each object of the metaphor (sectors, windows, molecules) to optimize the drawing process. The object manager runs any time some data essential to the drawing functions changes.

Initially the EVOWAVE algorithm needs to let the object manager enter some information about the objects into the metadata. This phase is called initialization and can be separated into two main algorithms: a) to define the start and end angle of each sector (Algorithm 1); b) to identify the smallest circumference that has the minimum size to draw one molecule (Algorithm 2).

Algorithm 1 Define Sector Angles

1: $StartAngle \leftarrow 0$
2: $EndAngle \leftarrow (PI \times 2)$
3: **for each** Sector **do**
4: **if** PreviousSector is not defined yet **then**
5: $Sector.StartAngle \leftarrow StartAngle$
6: **else**
7: $Sector.StartAngle \leftarrow PreviousSector.EndAngle$
8: $DiffAngle \leftarrow (EndAngle - StartAngle)$
9: $Sector.EndAngle \leftarrow (Sector.StartAngle + (DiffAngle \times Sector.Angle))$
10: $PreviousSector \leftarrow Sector$

Algorithm 2 Define the smallest radius possible to draw a molecule

1: $SmallestRadius \leftarrow WindowSize$
2: **repeat**
3: $SmallestRadius \leftarrow SmallestRadius + 1$
4: OffSet $\leftarrow$ ArcTan($\frac{WindowSize}{SmallestRadius}$)
5: **until** $(SmallestStartAngle + OffSet) > (SmallestEndAngle - OffSet)$

Algorithm 1 aims to save, in the metadata, the angle in which each sector will start and finish. Each sector initializes in the final angle of the previously sector (line 7) or zero, if it is the first to be processed (line 5). The end angle of a sector is calculated based on its start angle added to the portion defined in the "Sector.Angle" variable by the server of 360 degrees (line 9). The client side does not validate if the portion of all sectors exceeds the 360 degrees referring this responsibility to the server. The object manager holds this information because it does not change often making the execution of every drawing cycle faster.

Algorithm 2 helps solving the problem of drawing a molecule in a radius that is too small to draw in it. To overcome this problem, the algorithm finds the smallest radius

capable of drawing a molecule. This radius will used as a guideline to start drawing the windows. First, the smallest radius is defined as the defined window size because it can not be smaller than that (line 3). The algorithm continues to increment the smallest radius and to test if a molecule can be drawn in it or not until it finds a suitable radius. The test uses the offset (in radians) of the angle needed to draw the molecule (line 4). Finally, it verifies if this offset added to the start angle of the sector with the smallest angle exceeds its end angle subtracted to the offset (line 5). While this test is true the molecule will be drawn outside the boundaries of the sector, therefore the molecule cannot be depicted and a new smallest radius needs to be found.

After the initialization, the whole canvas is cleared to avoid overlap between the new and the previous visualization. The overlap occurs because a new visualization is drawn when a filter is applied. Now the canvas is ready to display the new visualization. This phase is accomplished by the drawing algorithm represented by Algorithm 3. Since the framework does not use the retained mode, the canvas needs to be redrawn every time something changes in the data. This makes sense when the change impacts the whole canvas, but when the change impacts only a partial area of the canvas, it consumes too much processing to redraw everything. To avoid this situation a DirtyManager component was created to monitor what in terms of sectors, windows and molecules should be redrawn. Every time there is something to be redrawn the DirtyManager is defined and the visual elements (e.g., sectors, windows and molecules) to be redrawn are set as dirty (lines 5 to 14). If the DirtyManager is not defined, the whole visualization should be redraw (lines 1 to 3). The drawing algorithm was divided in four to better understanding: DrawSector, DrawWindow, DrawMolecule and DrawMoleculeGroup. They are explained next.

Algorithm 3 Draw the visualization

```
 1: if DirtyManager is not defined then
 2:     for each Sector do
 3:         DrawSector(Sector)
 4: else
 5:     for each Sector in DirtyManager do
 6:         if Sector.Windows is not empty then
 7:             for each Window in Sector.Windows do
 8:                 if Window.Molecules is not empty then
 9:                     for each Molecule in Window.Molecules do
10:                         DrawMolecule(Molecule)
11:                 else
12:                     DrawWindow(Window)
13:         else
14:             DrawSector(Sector)
```

Algorithm 4 indicates how the sector is drawn in the visualization. First, the sector is painted using the white color to erase any visual information on it (line 2). To start

Algorithm 4 Draw a Sector

```
1: function DrawSector(Sector)
2:     CleanSector(Sector)
3:     x0 ← CenterX + (SmallestRadius × Cos(Sector.StartAngle))
4:     y0 ← CenterX + (SmallestRadius × Sin(Sector.StartAngle))
5:     x1 ← CenterX + (BiggestRadius × Cos(Sector.StartAngle))
6:     y1 ← CenterX + (BiggestRadius × Sin(Sector.StartAngle))
7:     DrawLine(x0, y0, x1, y1)        ▷ Draw the sector line related to the StartAngle
8:     x0 ← CenterX + (SmallestRadius × Cos(Sector.EndAngle))
9:     y0 ← CenterX + (SmallestRadius × Sin(Sector.EndAngle))
10:    x1 ← CenterX + (BiggestRadius × Cos(Sector.EndAngle))
11:    y1 ← CenterX + (BiggestRadius × Sin(Sector.EndAngle))
12:    DrawLine(x0, y0, x1, y1)        ▷ Draw the sector line related to the EndAngle
13:    for each Window in Sector.Windows do
14:        DrawWindow(Window)
```

drawing the sector, it is needed to draw two lines that limit the sector area. They are located at the StartAngle and EndAngle of the sector from the center to the BiggestRadius. The algorithm uses a trigonometric equation (lines 3 to 7 and 8 to 11) to calculate the two points of each line. Then, the framework function DrawLine is used to draw the line (line 7 and 12). Since the boundaries are defined and depicted, the algorithm can draw each windows inside of this sector (line 13 to 14).

Algorithm 5 indicates how the window inside a sector is drawn in the visualization. First the window is painted using the white color to erase any visual information on it (line 2). Then, the algorithm needs to calculate the angle increment needed to draw a molecule in the window (line 3). This information is used to check if the number of molecules that need to be drawn inside the window will exceed the sector boundaries (line 6). If there is space to draw all molecules, the algorithm calculates the distance (in radians) between molecules by getting the angle available in the sector and dividing it with the number of molecules to be drawn (line 7). Now it is possible to calculate the position of each molecule inside the window. For each molecule the relative angle inside the window is calculated by multiplying the position in the array by the fixed distance between molecules (line 10). Thus, the absolute angle can be determined by adding the relative angle to the sector's start angle (line 11). The absolute angle is where the center of the molecule will be depicted. Once calculated, it is possible to determine the two-dimensional position (x and y) using a trigonometric function (line 12 to 13). It uses the radius located in the middle of the window (line 4) in order to draw the molecule with its center in there. At last, the function DrawMolecule is called to actually draw each molecule (line 14). If there is no space to draw all molecules in the window, the algorithm groups all molecules with the same color property and depicts one bar for each group indicating the proportion of each color inside the window (line 17 to 25). At first, all molecules are added in their groups and saved in the metadata (line 17 and 18). Once all molecules have been grouped, for the angle each group will occupy is calculated (line

Algorithm 5 Draw a Window in its Sector

1: **function** DrawWindow($Window$)
2: CleanWindow($Window$)
3: Window.Offset $\leftarrow$ ArcTan($\frac{WindowSize}{(Window.Position \times WindowSize) + SmallestRadius}$)
4: RadiusInMiddle $\leftarrow$ $((Window.Position \times WindowSize) + SmallestRadius) - \frac{WindowSize}{2}$
5: **if** Window has molecules **then**
6: **if** (Sector.EndAngle - Sector.StartAngle)> $(Window.NumberOfMolecules \times Window.Offset)$ **then**
7: DistanceBetweenMolecules $\leftarrow$ $\frac{(Sector.EndAngle - Sector.StartAngle)}{Window.NumberOfMolecules}$
8: Index $\leftarrow$ 0
9: **for each** Molecule in Window.Molecules **do**
10: RelativeMoleculeAngle $\leftarrow$ $Index \times DistanceBetweenMolecules$
11: Molecule.Angle $\leftarrow$ $Sector.StartAngle + RelativeMoleculeAngle$
12: Molecule.X $\leftarrow$ $CenterX + (RadiusInMiddle \times \text{Cos}(Molecule.Angle))$
13: Molecule.Y $\leftarrow$ $CenterX + (RadiusInMiddle \times \text{Sin}(Molecule.Angle))$
14: DrawMolecule($Molecule$)
15: Index $\leftarrow$ $Index + 1$
16: **else**
17: **for each** Molecule in Window.Molecules **do**
18: Window.Groups[Molecule.Color].add(Molecule)
19: PreviouslyGroupEndAngle $\leftarrow$ $Sector.StartAngle$
20: **for each** Group in Window.Groups **do**
21: GroupAngle $\leftarrow$ $(Sector.EndAngle - Sector.StartAngle) \times (\frac{Group.NumberOfMolecules}{Window.NumberOfMolecules})$
22: Group.StartAngle $\leftarrow$ $PreviouslyGroupEndAngle$
23: Group.EndAngle $\leftarrow$ $Group.StartAngle + GroupAngle$
24: DrawMoleculeGroup($Group$)
25: PreviouslyGroupEndAngle $\leftarrow$ $Group.EndAngle$

21). The calculation is done using the proportion between how many molecules there are in the group and how many molecules there are in the window. When multiplying this proportion with the angle of the sector, the resulting angle is the angle filled by the group in the window. Once we have this information, it is possible to determine where is the start and end angle of each group (line 22 and 23). Finally, the algorithm calls the function DrawMoleculeGroup to actually fill the windows according to the calculated angles with the color of the group (line 24).

3.5 CHAPTER CONCLUSION

This chapter introduced the facts about concentric waves that helped during the conception of the EVOWAVE concepts. We explained the motivation and how each concept works. Those concepts were created to enable the mapping of different data generated in many domains in software engineering. Thus, we explained how it is possible to map the visual attributes of the metaphor to different software properties. Later, we explain the main algorithms behind the creation of the EVOWAVE metaphor and the metadata that helps it be generic enough to represent different domains.

Chapter 4

The use of a single metaphor to represent data from different domains is a novel approach. This chapter presents three exploratory studies were the EVOWAVE metaphor was applied to three different domains performing tasks related to each one.

VALIDATION

EVOWAVE is designed to be a multiple domain metaphor for software evolution visualization. In order to validate its goal, we conducted three studies in different evaluation scenarios. Each study exercises the EVOWAVE metaphor in a different software evolution domain. The explored domains were: Software Collaboration (Section 4.1), Library Dependency (Section 4.2), and Logical Coupling (Section 4.3). Tasks related to each one of those domains were performed using the EVOWAVE metaphor with different data from different repositories.

4.1 AN EXPLORATORY STUDY ON SOFTWARE COLLABORATION

In 2014, an exploratory study was conducted to validate the use of EVOWAVE to analyze the collaboration of developers during the software development. This study was performed in the tool prototype, since it was first deployed in 2015. The prototype uses the same concepts and visual elements, but may look a little different. The lines used to split the sectors are much more dark and background color of the windows without molecules is not white. We can see those visual differences in the next two studies (Section 4.2 and Section 4.3), they were performed in the final tool. Using the GQM paradigm (Basili; Rombach, 1988), the goal of this study was:

- **To analyze** the EVOWAVE metaphor
- **With the purpose of** validate
- **Regarding** the evolution of the developers contribution during the software development
- **From the point of view of** EVOWAVE researchers
- **In the context of** software comprehension tasks designed by Fritz (Fritz; Murphy, 2010) in a real world open source system.

4.1.1 Study Settings

In this exploratory study, we use the EVOWAVE metaphor to perform typical questions asked by developers and project managers in the context of software collaboration (Fritz; Murphy, 2010). The Fritz's study uses the source code repository to extract the developers contributions from the jEdit system (jedit, 2015). This system is: an open source programming text editor; written in Java; with more than fourteen years of development; at least, 300,711 lines of code; and more then one million of changes. The developer Slava Pestov started to develop it in 1998. Later, the project received more contributions from the open source community.

4.1.1.1 Setup We developed a parser to read Git log files and extract the commits submitted to it. For each commit, we extract the following data: who sent the commit, what java file changed, and when the commit was sent. With this data, we setup the metaphor as follows:

- **The period of analysis is** from September 30th, 1998 to August 08th, 2012.
- **Sectors** can be java files, java packages or developers.
- **Windows** represent six months
- **Molecules** represent a java file that was changed.
- **Molecule Colors** represent the developer who made the change.

4.1.1.2 Tasks To guide our evaluation, we considered a study that reports typical questions asked by developers and project managers in the context of software collaboration (Fritz; Murphy, 2010). We were able to answer five of these questions about the jEdit using the metaphor with the configuration described. The other questions require other sources of information that were missing in the jEdit repositories (e.g. stack traces and, test cases). The tasks and the motivation to answer those tasks are described below:

1. **Who is working on what?** Developers and managers always want to know who is making changes in what part of the software. This helps to understand how much time is being put into a determined module of the system.
2. **How much work have people done?** To determine the time spent by developers to evolve the system is important because it helps to estimate the level of knowledge about the system that he or she has.
3. **What classes have been changed?** It is important to monitor the classes that are changing. It helps to identify what modules need more testing and the number of developers that are putting work into some functionality.

4. **Who has the knowledge to do the code review?** Code review has been used as a successfully technique to identify anomalies in the source code. The identification of developers that have made changes to a certain module more frequently is important because they will perform a good code review due to the knowledge obtained during the maintenance activities.

5. **Who is working on the same classes as I am and for which work item?** It is important for developers to identify other team members that are changing the same classes because source code conflicts may occur in the future.

4.1.2 Study Execution

Who is working on what?

This is a typical question asked by software managers and technical leaders. In large teams, the manager knows which feature or bug developers are coding using a tool such as Bugzilla (Bugzilla, 2015) or Jira (Atlassian, 2015). However, it is difficult to track what artifacts developers are working on (e.g., what packages they are changing). This happens because source code management tools displays raw data about the changes in the software (e.g., command line tools to see the difference between file versions).

To reach this goal, we narrowed some information to give a more accurate answer. We adapted the question to: Who has been working in the last 5 months on such package?. Thus, we configured the timeline to represent 5 months (March 08, 2012 - August 08, 2012): the oldest (inner guideline) to the newest date (outer guideline), respectively. The molecules are associated to the change of a java file and its color with the author of the change. The sectors are using java package as their pooler. We do not consider the splitter or angle property for this analysis.

Figure 4.1 presents the EVOWAVE visualization within the described set-up for jEdit. The authors who have been working on the last 5 months are listed in the bottom-right area of the figure with their respective colors. By analyzing the timeline paths, we can see when the contributions were made. EVOWAVE still provides a tooltip box, which gives more precise time information when the user interacts with the visualization. In addition, EVOWAVE uses the sectors to show the package developers are working on during the selected period.

This visualization allows us to understand what happened during this period of time. Then, we can start to explore it in order to do some interpretations. For example: a) The transparent black area highlights the author Hisateru Tanaka‘s (brown color) contributions to almost all packages during the last 5 months in the same window. Maybe it is important to understand what changes he made in the past that had many impacts; b) the org/gjt/sp/jedit/ sector (A) had many changes (in terms of commits) performed by different authors. As a possible side effect, bugs can appear later if these authors do not synchronize their tasks. Therefore, a code review should be performed in this package to disclaim any possibility of architecture violations.

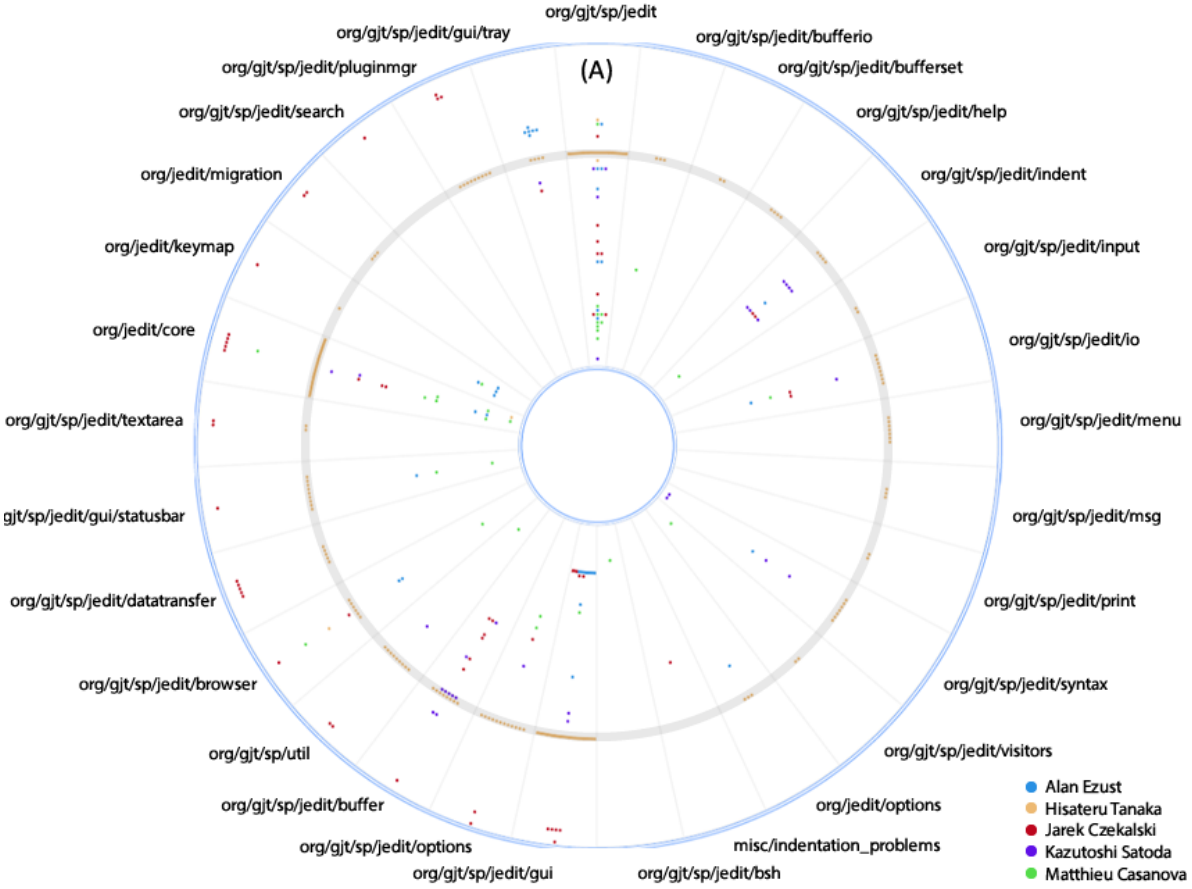


Figure 4.1 EVOWAVE showing who is working on such package

How much work have people done?

In 14 years of development it is hard to know how much work each collaborator has done. Work done may have several interpretations. For example, one may consider the number of commits as a metric to evaluate the quantity of work; others may use the number of lines of code. The timeline could be mapped to any period within the 14 years. For this question, we decided to visualize the whole period. The orientation was set-up from the oldest (inner guideline) to the newest date (outer guideline). The molecules are associated to the change of a Java file and their colors with the author of the change. The sectors are also the authors. Again, we did not need to use the splitter or angle visual properties for this analysis.

Figure 4.2 shows EVOWAVE in action for the period and configuration described. All the developers who worked in the project are listed in the sectors. The number of molecules in each sector represents the amount of work. More precise information can be achieved by using some of the EVOWAVE interaction mechanisms (e.g. hold the mouse over the sector while pressing the keyboard modifier shift key).

Using this visualization, it is possible to clearly see that Slava Pestov (black color) was

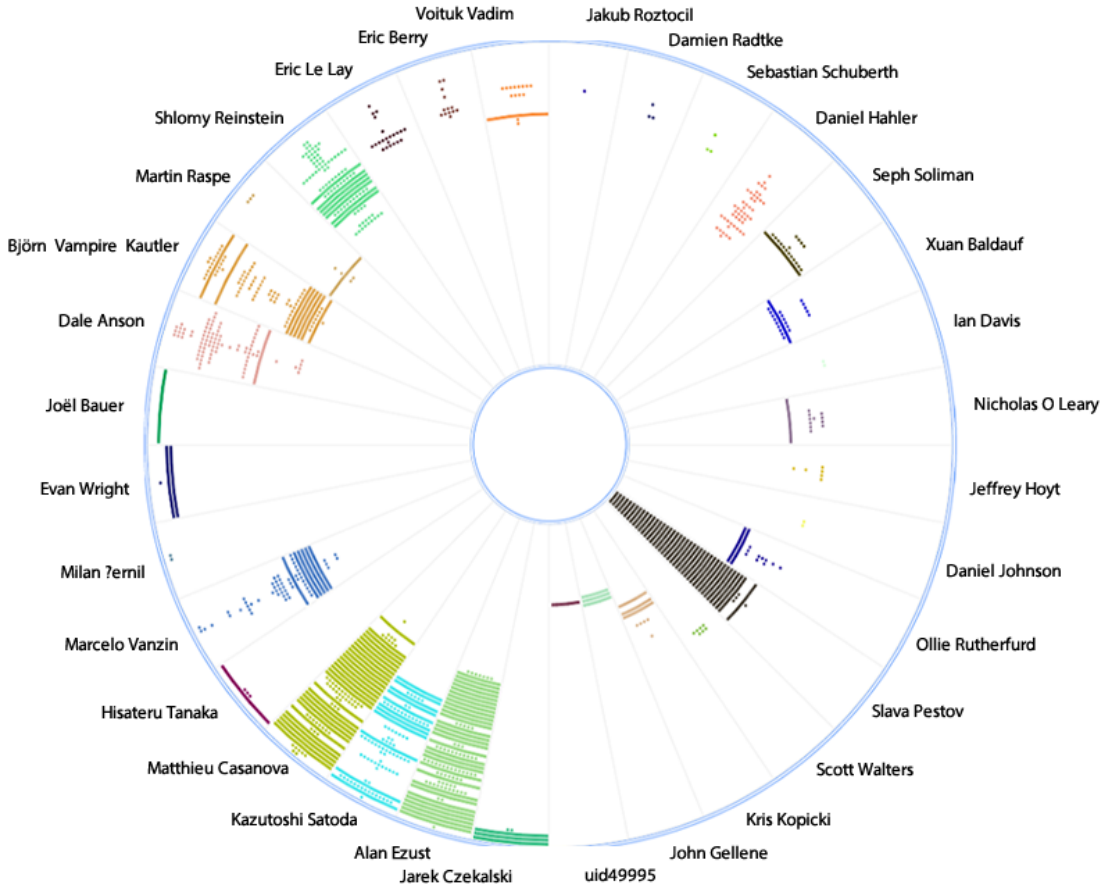


Figure 4.2 EVOWAVE showing how much work has people done

the founder of the project. In the middle of the project he left it. Matthieu Casanova and Alan Ezust were the two main code contributors after the founder, while Jazub Roztocill, Damien Radtke, and Sebastian Schuberth made small contributions before leaving the project. Many questions can be raised, such as: Did those three developers have enough knowledge about the system to make those changes? What were the decisions made by them? It is important to recover these answers as soon as possible, since they may not return to the project or be available for questioning.

What classes have been changed?

During the software development, it is important to keep track of the classes that are being modified. Most of the version control system clients have this feature. However, when there are many commits, they do not provide a big picture of the changes. They only allow us to see what classes were changed, commit by commit. To reach the goal of this task, we established a timeline of one month (August, 2012). The orientation was set-up from the oldest (inner guideline) to the newest date (outer guideline). The molecules are associated to the change of a java file and their colors with the author of

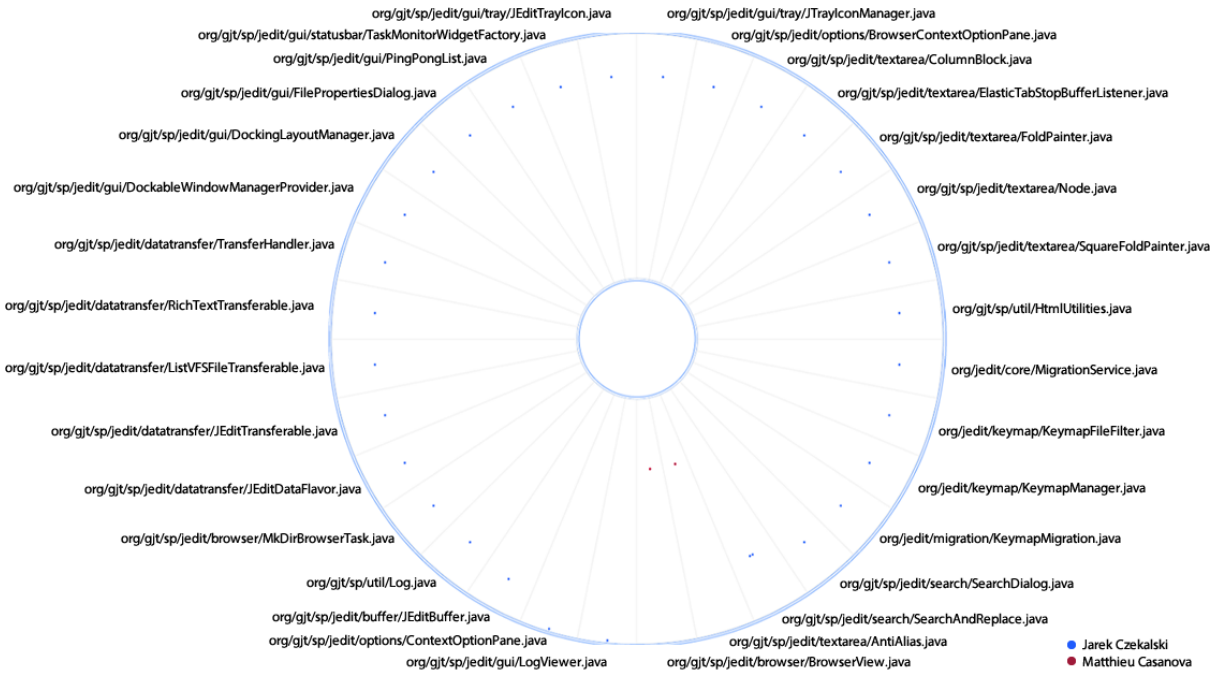


Figure 4.3 EVOWAVE showing what classes have been changed

the change. The sectors are the classes changed. No splitter or angle property was used.

Figure 4.3 shows an EVOWAVE visualization for the period and configuration described. All the classes changed in August, 2012 are displayed in the sectors. The authors who made those changes are listed in the bottom-right corner. Jarek Czekalski was the author who most changed the system in that month. During one window, he changed a lot of classes in different packages. The changes Matthieu Casanova made do not seem to have a direct impact on the changes by Jarek Czekalski. They changed different artifacts in different days.

Who has the knowledge to do the code review?

This is also a hard question to answer since there are many factors involved (e.g., business knowledge, architectural knowledge). However, we can assume that the author who made a lot of changes in some artifact during a long period of time knows it. We set a timeline from 2006 to 2012 because this was the period that the project received many contributions. The orientation is from 2006 (inner guideline) to 2012 (outer guideline). The molecules are associated to the change in a java file while their colors mean the author of the change. The sectors are using the java package as their pooler.

Figure 4.4 shows an EVOWAVE visualization for the period and configuration described. The visualization depicts the contributors in the corners with their respective colors. Some of them can be identified as the owner (i.e. most contributions) of some packages. Alan Ezust, for example, is the most indicated person to do the code review of the MacOS code (A). He mainly contributed to previous features as can be seen in the bottom-

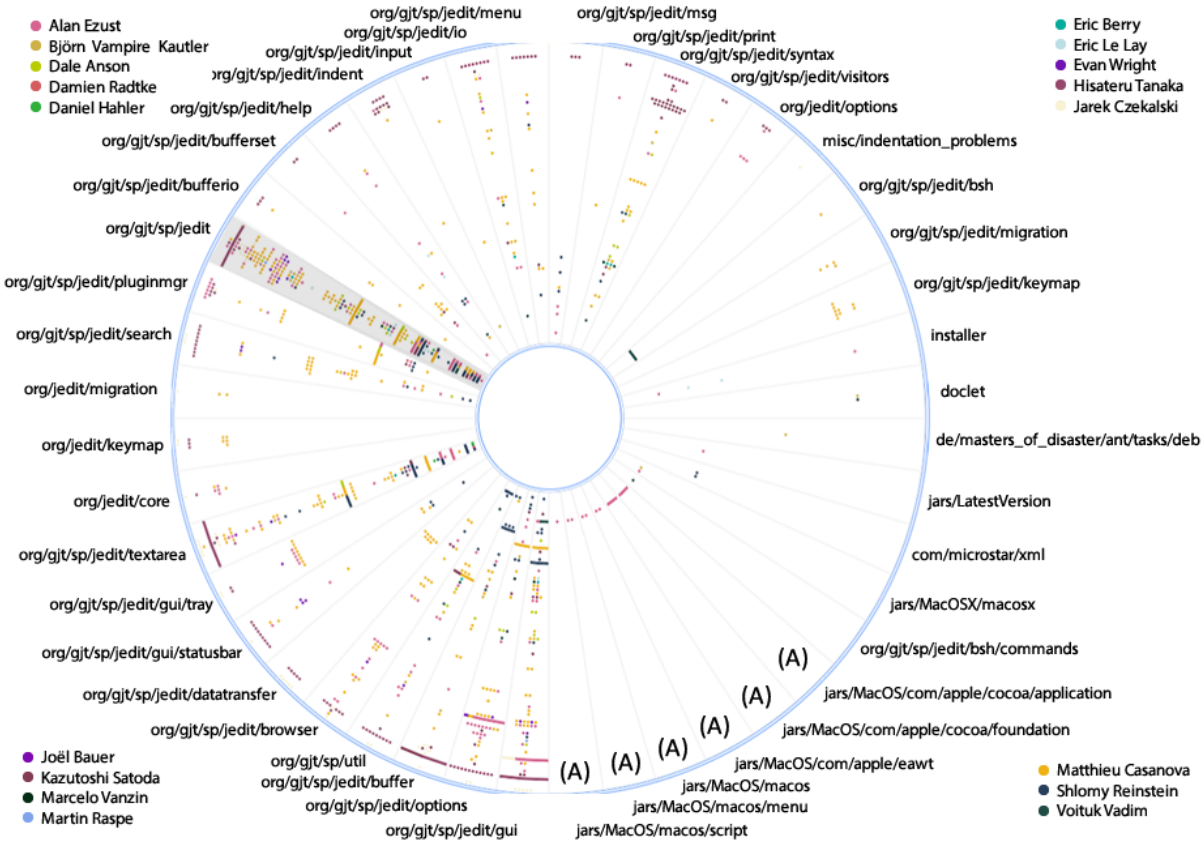


Figure 4.4 EVOWAVE showing who has the knowledge to do the code review

right packages. Another example is the "org/gjt/sp/jedit" package with the highest number of commits (transparent black area). Even though many people changed this package, the EVOWAVE shows that Matthieu Casanova (yellow color) had a considerable number of commits during the whole period. Therefore, he may be indicated to do the code review.

Who is working on the same classes as I am and for which work item?

Developers and managers may ask this question. Developers may want to know the collaborators that are changing a similar artifact, while managers may want to control conflicts among tasks. To answer this question, we set up the timeline for the last 5 months (March 08, 2012 - August 08, 2012), and set Kazutoshi Satoda as the developer who wants to know who is working in the same artifacts. The orientation was set-up from the oldest (inner guideline) to the newest date (outer guideline). The molecules are associated to the change of a java file and their colors with the author of the change. The sectors use the java package as their pooler.

Figure 4.5 presents the EVOWAVE metaphor visualization within the set up de-

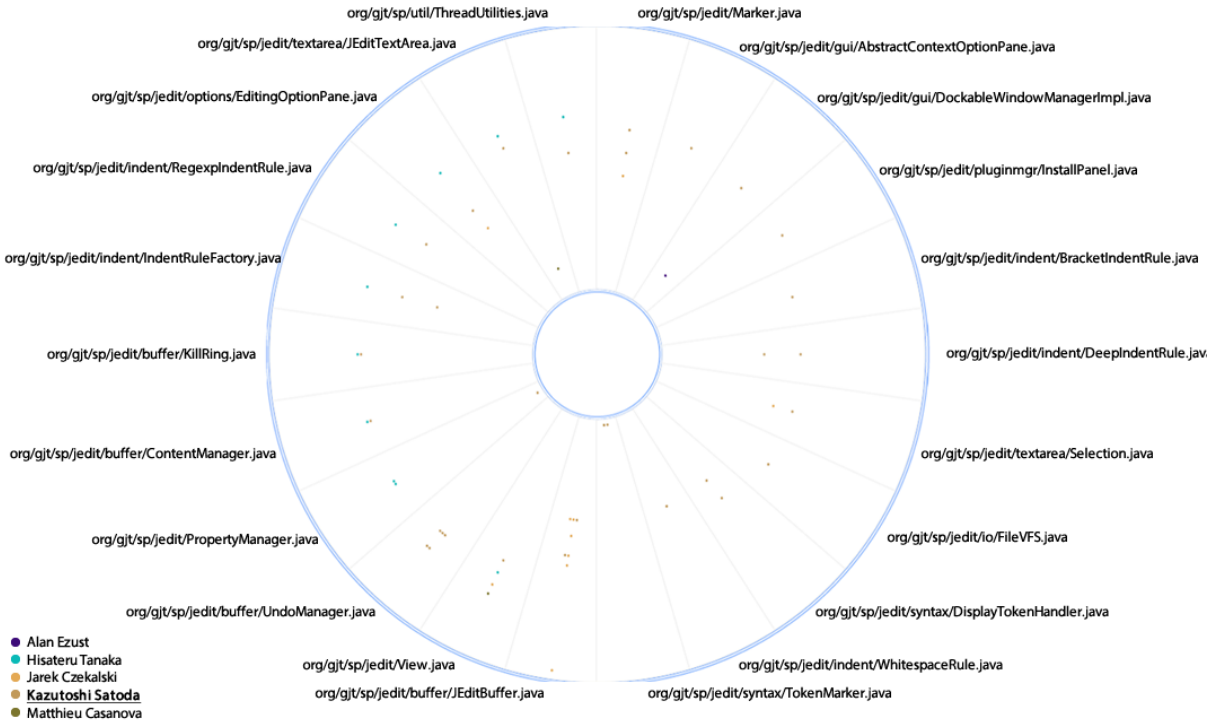


Figure 4.5 EVOWAVE showing who is working on the same classes as Kazutoshi Satoda and in what java files

scribed. The authors who are working on the same classes are displayed in the bottom-left corner. All the classes that were changed in the last 5 months by Kazutoshi Satoda are listed as sectors.

4.1.3 Conclusion

This study was performed using tasks proposed by Fritz (Fritz; Murphy, 2010) using a well known open source software. This study demonstrated that the EVOWAVE metaphor was able to portray a huge amount of data and able to organize the data accurately to extract information. Consequently, we could display data from the software collaboration domain and perform tasks related to this domain. We conclude that the EVOWAVE metaphor can be used to understand the developers contributions during the software development process.

4.2 AN EXPLORATORY STUDY ON LIBRARY DEPENDENCY DOMAIN

In 2015, we conducted an exploratory study to validate the use of EVOWAVE in the analysis of the evolution of how the dependency relation between a system and its dependencies evolves. This study was performed using the EVOWAVE tool instead of the prototype used in the first experiment (Section 4.1). Using the GQM paradigm (Basili; Rombach, 1988), the goal of this study was:

- **To analyze** the EVOWAVE metaphor
- **With the purpose of** validate
- **Regarding** the dependency relationship evolution between a system and its dependencies
- **From the point of view** of EVOWAVE researchers
- **In the context of** software comprehension tasks designed by Kula (Kula et al., 2014) in a real world open source system.

4.2.1 Study Setting

In this exploratory study, we followed the case studies presented by Kula (Kula et al., 2014). They used the Maven 2 Central Repository (Olivera, 2015) to extract the dependency relationships of four projects: FindBugs, FastJson, AtomServer, and SymmetricDS. Since they only walk-through for all scenarios with the FindBugs system, we will focus our EVOWAVE visualization for this system alone.

4.2.1.1 Setup An HTML parser was developed to read page by page of the repository to extract the dependency data from the FindBugs system. First we extract every version of the FindBugs system and when it was released. Then, we extract the dependencies from each one of those versions and when they were released. Finally, we extract all the systems registered in this repository that use one of those dependencies and when they were released. During the analysis of 196,329 HTML pages (5,5GB), the parser extracted 15 versions of the FindBugs system, 294 dependencies from the FindBugs system and 162,510 system versions that use at least one of those dependencies. With this data [1] we setup the metaphor as follows:

- **The period of analysis is** from February 11, 2003 to November 15, 2015.
- **Sectors** are the dependencies from the FindBugs system. Each dependency has sectors inside which represent dependency versions.
- **Windows** represent six months.
- **Molecules** represents the use of only one dependency by FindBugs system itself or another system.
- **Molecule Colors** represent if the system that is using the dependency is adopting (green), remaining (blue), or updating (red) this dependency. The system is adopting the dependency when its first version is using the dependency. The system remains with the same dependency when its previous version uses the same version of the dependency at the time. Finally, the system updates the dependency when its previous version uses a different version of the dependency at the time.

[1] This data was uploaded to the Internet and it can be downloaded at https://wiki.dcc.ufba.br/SoftVis/projects/evowave

Figure 4.6 has two possible visualizations of the EVOWAVE metaphor with the described setup. The visualization (A) is showing the use of all dependencies by the FindBugs system and in (B), the use of all dependencies.

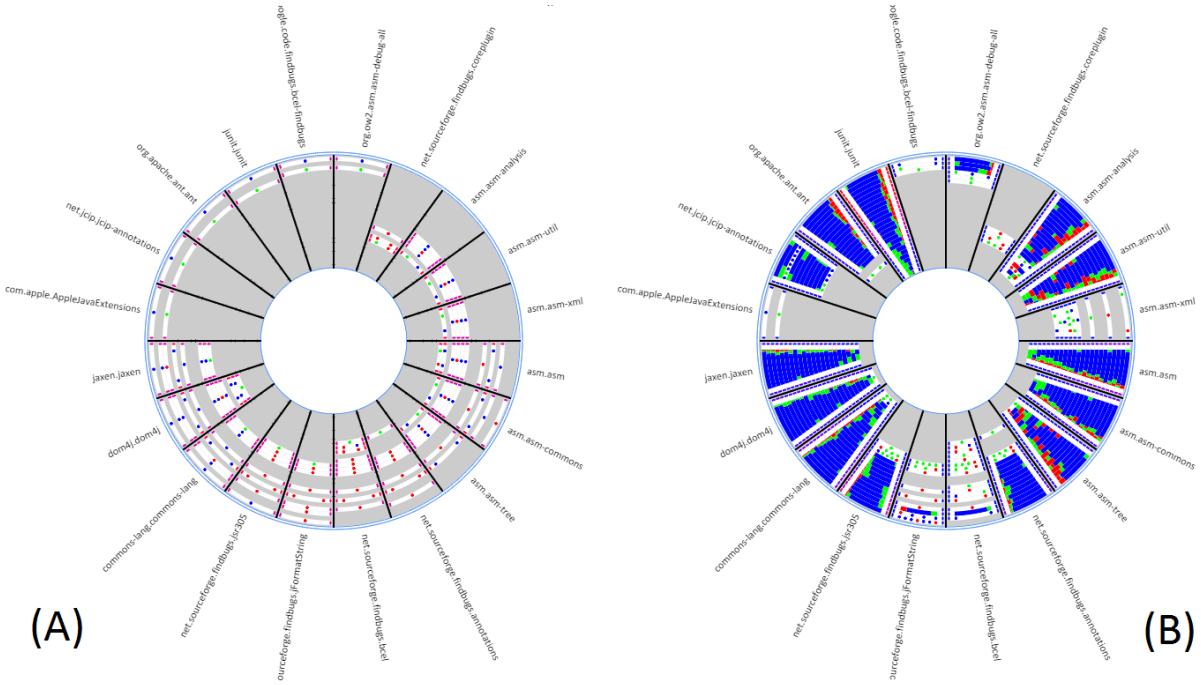


Figure 4.6 Examples of EVOWAVE visualizations for the Library Dependency Domain. The visualization (A) is showing the use of all dependencies and in (B) the use of all dependencies by the FindBugs system.

4.2.1.2 Tasks

The task definitions were extracted from the following scenario:

> "Rusty is a new maintainer to a software project. Rusty notices that some of the system's library dependencies are outdated. Simply upgrading to the latest versions of all dependencies seems natural, however, Rusty does not know where to start. How to help Rusty?" (Kula et al., 2014)

Given this scenario, the first two tasks will help him understand the systems dependency structure to prioritise which libraries should be upgraded and, the last two tasks, will help him identifying suitable candidate library versions for upgrade (Kula et al., 2014). The tasks and the motivation to answer them are described bellow:

1. **Understand the regularity of system dependency changes:** "The evolution history gives an indication of the frequency in relation to the version releases. It would also be useful if a system is more inclined to risk by adopting a newer version or is a either a regular updater or would rather wait until a library version is used by other similar systems before adopting" (Kula et al., 2014).

2. **Understand what important structural dependency events have occurred:** "Dependency relation changes such as dropped and adopted libraries can provide clues for important structural changes. Patterns can be used for understanding various historic events between dependencies" (Kula et al., 2014).

3. **Discover the current "attractiveness" of any library version:** "Understanding the movement of adopters, idlers and updater systems provides visual clues on its 'attractiveness'" (Kula et al., 2014).

4. **Discover if newer releases are viable candidates for updating:** "Assessment of the 'attractiveness' of newer library versions can assist maintainers with the upgrade decisions" (Kula et al., 2014).

4.2.2 Study Execution

In this section, we show how we used EVOWAVE to perform each task aforementioned.

Understand the regularity of system dependency changes

Figure 4.7 displays the visualization filtered only for the dependency usages by the FindBugs system. To understand the regularity of changes in the system dependency, the software engineer needs to look for molecules changes in the sectors. The "com.apple.Apple JavaExtensions", "net.jcip.jcip-annotations", "org.apache.ant.ant", "junit.junit", "com.google.code.findbugs.bcel-findbugs", and "org.ow2.asm.asm-debug-all" dependencies are new for the FindBugs system and were never updated (They are represented as (A) in Figure 4.7). We can reach this conclusion because the molecules in the sectors related to those dependencies are presented on the edge of the visualization and there are no red molecules in it. Unlike those dependencies, the "net.sourceforge.findbugs.annotations" dependency is old and highly changed. The FindBugs system used this dependency from the first version until the middle of 2014. The dependency was updated almost every time a new version of the FindBugs system was released. This behaviour represents an indication of high coupling between those systems. Another important behaviour to notice is that this dependency was not used in some versions. For example, the next window after the window that holds the green molecule for this dependency is empty. But there were new versions released during this period as we can see in the sector "net.sourceforge.findbugs.bcel". The "net.sourceforge.findbugs.annotations" dependency was used after this window again. This behaviour represents that the features provided by this dependency might be replaced by another library. One possible candidate is the "net.sourceforge.findbugs.coreplugin" because it is only used when the "net.sourceforge.findbugs.annotations" dependency is not. We can notice that the regularity of changes in the "dom4j.dom4j" dependency is minimal. It is presented in all versions of the FindBugs system and its version was only updated once.

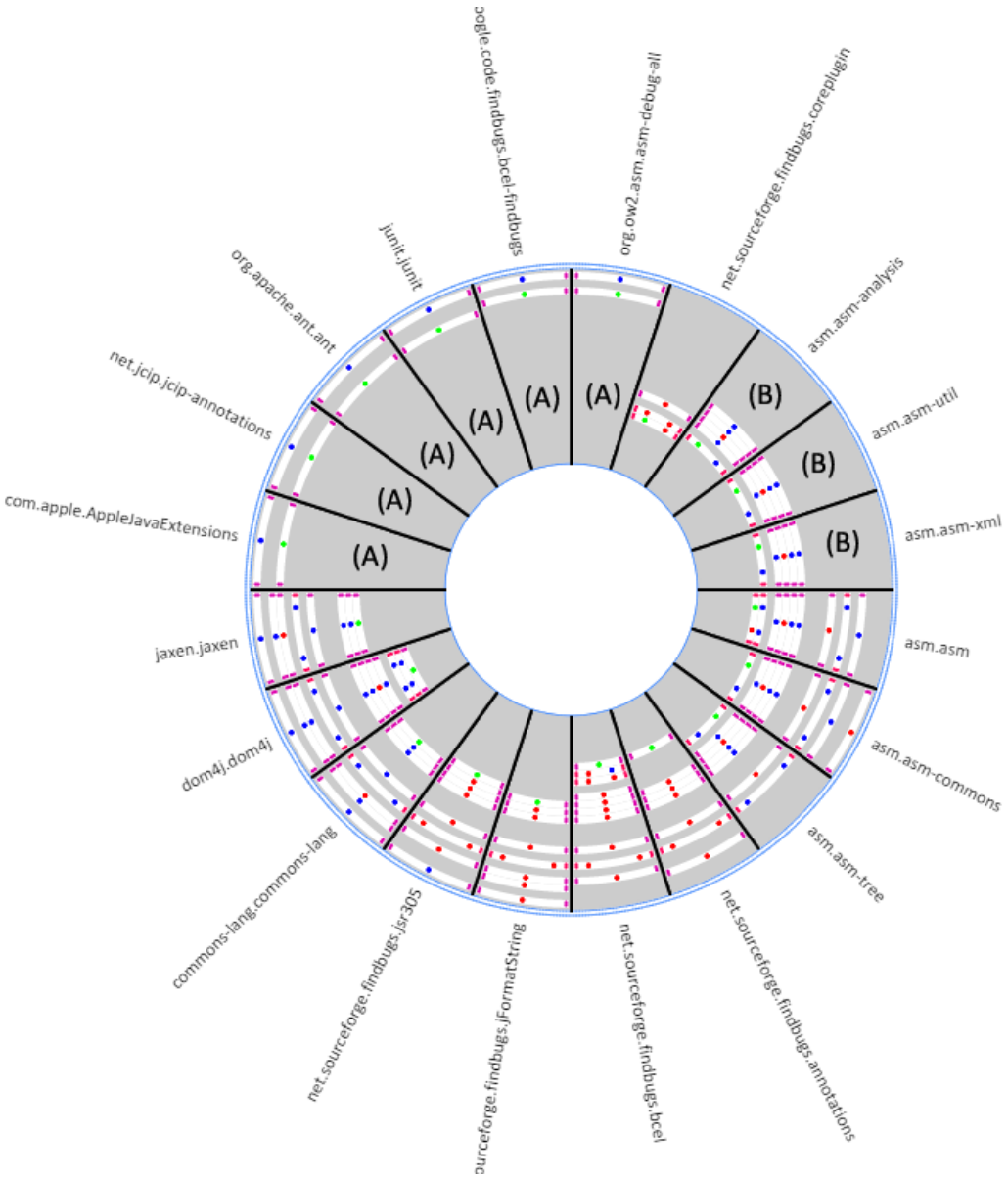


Figure 4.7 The dependency usage of FindBugs project

Understand what important structural dependency events have occurred

The EVOWAVE visualization, as in Figure 4.7, can also be used to address this task by looking for changes in the dependencies. The dependencies "asm.asm-analysis", "asm.asm-util" and "asm.asm-xml", represented as (B) in the Figure, were removed from the project at the same time. This may imply that they were used for some common feature that is using a different library or were removed. Those previously defined as new dependencies, represented as (B) in the Figure, are another important structural event. The dropped library "net.sourceforge.findbugs.bcel" is an important structural event because it was highly used from the beginning and was always updated. This behaviour

indicates high coupling between this dependency and the FindBugs system. Currently the latest versions are not using this library. This may indicate a big system change to portray all the features held in this library, or the system integration to another library with the same features. The dependency "com.google.code.findbugs. bcel-findbugs" was introduced in the system at the same time "net.sourceforge.findbugs.bcel" was removed. Based on the names of the libraries it is possible to notice that this new library might be replacing the old one or maybe the library was simply renamed.

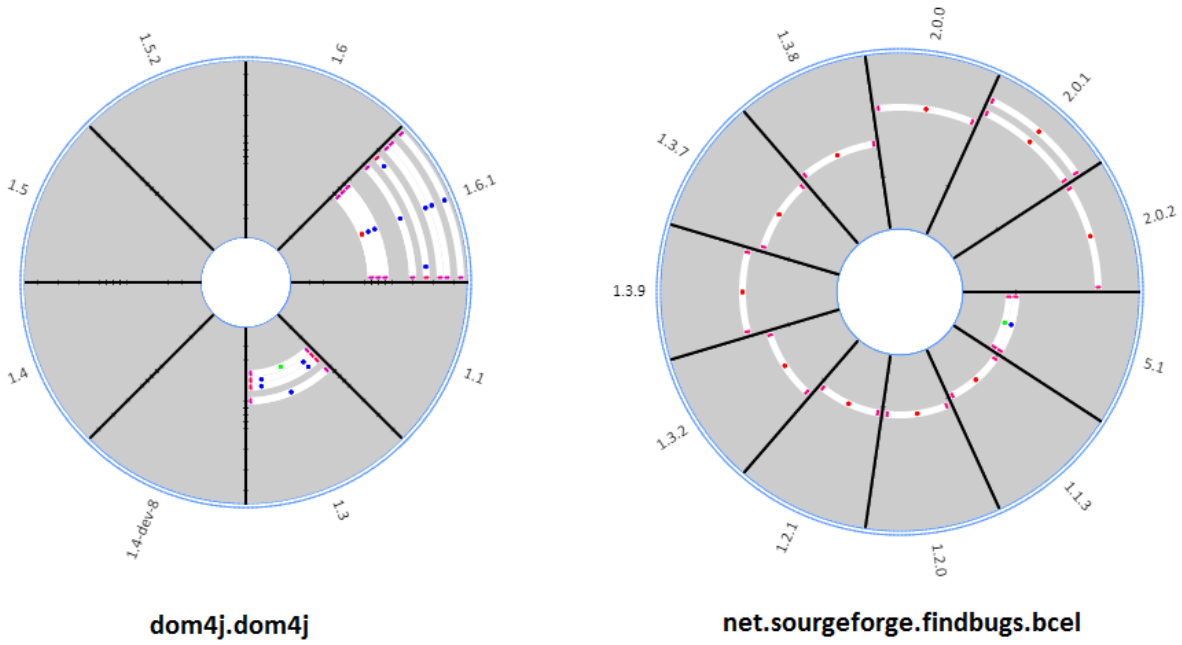


Figure 4.8 Two EVOWAVE visualizations for two dependencies with the usages by the FindBugs system.

To analyze the "net.sourceforge.findbugs.bcel" dependency in more detail, the software engineer can right click on the sector that represents it. Figure 4.8 displays the EVOWAVE visualization for this dependency on the right. A common behaviour of this library usage is the upgrade to a newer version every time there is a new release of the FindBugs system. The latest version that uses this library breaks this pattern by returning to an old version ("2.0.1"). This is an important structural event since the current newer version of this dependency no longer satisfies the requirements of the FindBugs system. Maybe this is why another library called "com.google.code.findbugs. bcel-findbugs" was created. Unlike this dependency, the "dom4j.dom4j" dependency seems to be highly stable for this system. Nevertheless, the change from version "1.3" to version "1.6.1" was an expressive change because there were five versions between them that were not used in the system. The library may have changed a lot during these five versions which implies an expressive change in the system.

Discover the current "attractiveness" of any library version

The latest version of FindBugs uses the "junit.junit" version "4.11". Figure 4.9-A represents the use of all versions from this dependency registered in the Maven repository. We can see that all versions, except for "3.7", have systems that maintain the same version of this dependency because, in the last window, there are blue molecules in the sector of those versions. However, versions "4.11" and "4.12" are the versions with the highest number of projects using them in the last six months since the number of molecules in the last windows of their sectors have the reddest color in the indicator located on the sides. Since the goal is to look for current attractive versions, the visualization was filtered to show only the versions between those versions. Figure 4.9-B displays the visualization filtered. We can see the increasing number of projects that are updating to version "4.12" by looking at the amount of red molecules in the last six months. The current version used by FindBugs system still has a huge amount of projects starting to use, update, and maintain it. Nevertheless, this number has been decreasing over the years. This indicates that the newer version is becoming stable and reliable enough for the FindBugs system to update its dependency to this new version. An interesting fact is that three beta versions of version "4.12" were released almost at the same time as version "4.12" was released and had lower attractiveness. That can be seen while looking at the number of molecules in their sectors. This indicates the lack of trust from other systems in the "junit.junit" beta versions.

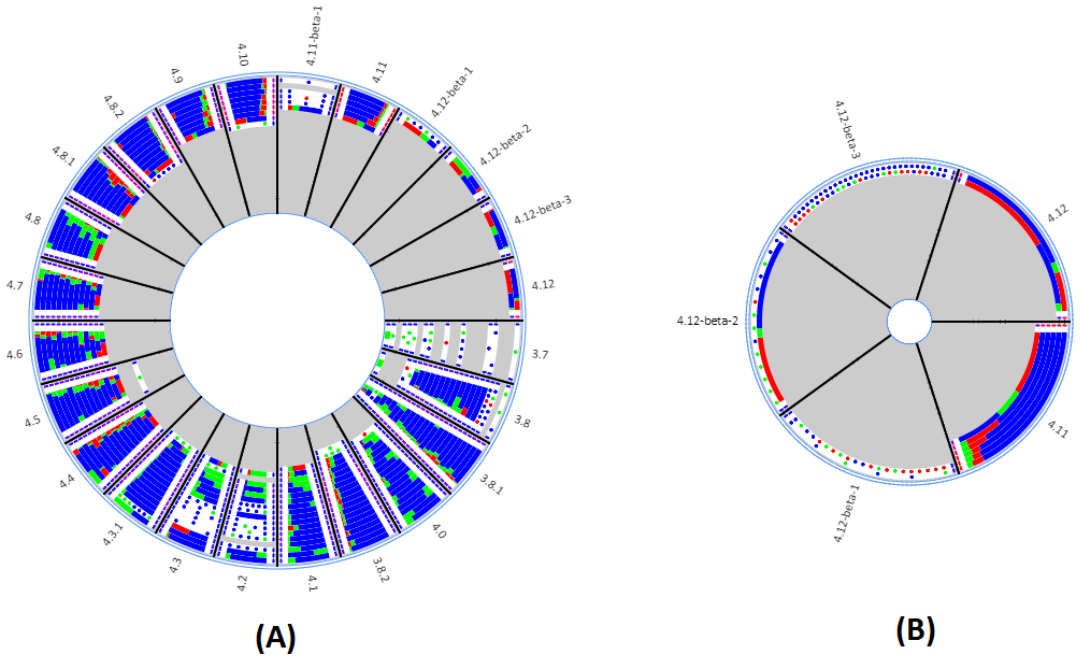


Figure 4.9 Two EVOWAVE visualizations of the "junit.junit" dependency usage.

Discover if newer releases are viable candidates for updating

The FindBugs system can look at four newer releases to update for the "junit.junit" dependency. One of the things the software engineer needs to analyze is the attractiveness of those versions. As was shown in the last task, the attractiveness of beta versions for this dependency is low. This fact eliminates the viability of an update for those versions. The last version to verify the viability is the "4.12". As discuss before, these versions have been really attractive in the last six months. In less then one year, a huge number of projects have been updated to it while there is a decrease in the number of projects updating to the current version used by the FindBugs system. The software engineer of this system should consider updating to the version "4.12" of the "junit.junit" dependency and should not consider the beta versions.

4.2.3 Conclusion

This study was performed using the same data and tasks proposed by Kula (Kula et al., 2014). The author answered those questions using two metaphors he proposed. To answer those questions the software engineer had to learn the concepts of those metaphors proposed by him, and have some experience with them. The rational for this study was to validate the possibility of performing those tasks with only one metaphor, EVOWAVE. Consequently, those tasks could be performed with little interaction in the metaphor leading to the conclusion that the EVOWAVE metaphor can be used to understand how the library dependency evolves during the software development process.

4.3 AN EXPLORATORY STUDY ON LOGICAL COUPLING DOMAIN

In 2015, we conducted an exploratory study to validate the use of EVOWAVE to analyze the logical coupling between system modules during the software development process. This study was performed using the EVOWAVE tool instead of the prototype used in the first experiment (Section 4.1). Using the GQM paradigm (Basili; Rombach, 1988), the goal of this study was:

- **To analyze** the EVOWAVE metaphor
- **With the purpose of** validate
- **Regarding** logical coupling evolution between software modules
- **From the point of view** of EVOWAVE researchers
- **In the context of** a retrospective analysis made by Ambros (D'ambros; Lanza; Lungu, 2009) in a real world open source system.

4.3.1 Study Settings

This exploratory study was inspired by the retrospective analysis made in Ambros (D'ambros; Lanza; Lungu, 2009) to understand the logical coupling between software modules. This

study uses the source code repository to extract the files changed at the same time (i.e. logical coupling) in the ArgoUML system. This system is an UML modeling tool written in Java with more than 17 years of development and at least 200,000 lines of code distributed in 4,222 classes.

4.3.1.1 Setup A parser was developed to read Subversion log files and extract the files that were changed along with a file in the "org.argouml.uml" package. For each commit we extract the following data: the changed file and its package, when the change occurred, and the commit's id. With this data we setup the metaphor as follows:

- **The period of analysis** from September 4, 2000 to January 11, 2015.
- **Sectors** are software modules.
- **Windows** represent six months.
- **Molecules** represent changes in a java file.
- **Molecule Colors** represent the commit of the change. The commits in black made no changes in the selected module, if one was selected.

4.3.1.2 Tasks To guide our study, we considered a study that analyzed changes in the source code to understand the logical coupling between system modules (D'ambros; Lanza; Lungu, 2009). This study uses the logical coupling between modules to understand the dependencies in the system. The task of this study was to make a retrospective analysis of the logical coupling evolution in the ArgoUML system.

4.3.2 Study Execution

At first, we need to understand how the changes are spread among the system modules (i.e. java packages). Figure 4.10 helps us to identify that the java package "org.argouml.uml" underwent many changes during fifteen years of development. We choose to analyze the logical coupling of this package because: 1) of the amount of changes in it; 2) it was highly changed from the beginning to the present date; 3) the package named "uml" should be important for a UML tool.

We filtered the period limiting it from January 2, 2003 to December 30, 2005 in order to analyze the logical coupling of the package "uml" during the period with most changes. Additionally, for more detailed information, we changed the period of the window from six to one month. Figure 4.11 illustrates the visualization with this filter and aggregates all modules into one sector to get a global picture of the logical coupling of the "uml" package. We can understand that most commits caused changes in the "uml" package because the black color has low presence in the visualization. Another information that can be retrieved from this global picture is the number of commits with many files being changed. This information is extracted while looking at the size of the bars with the same color. This is a problem because the bigger the number of changed files, the greater is the logical coupling.

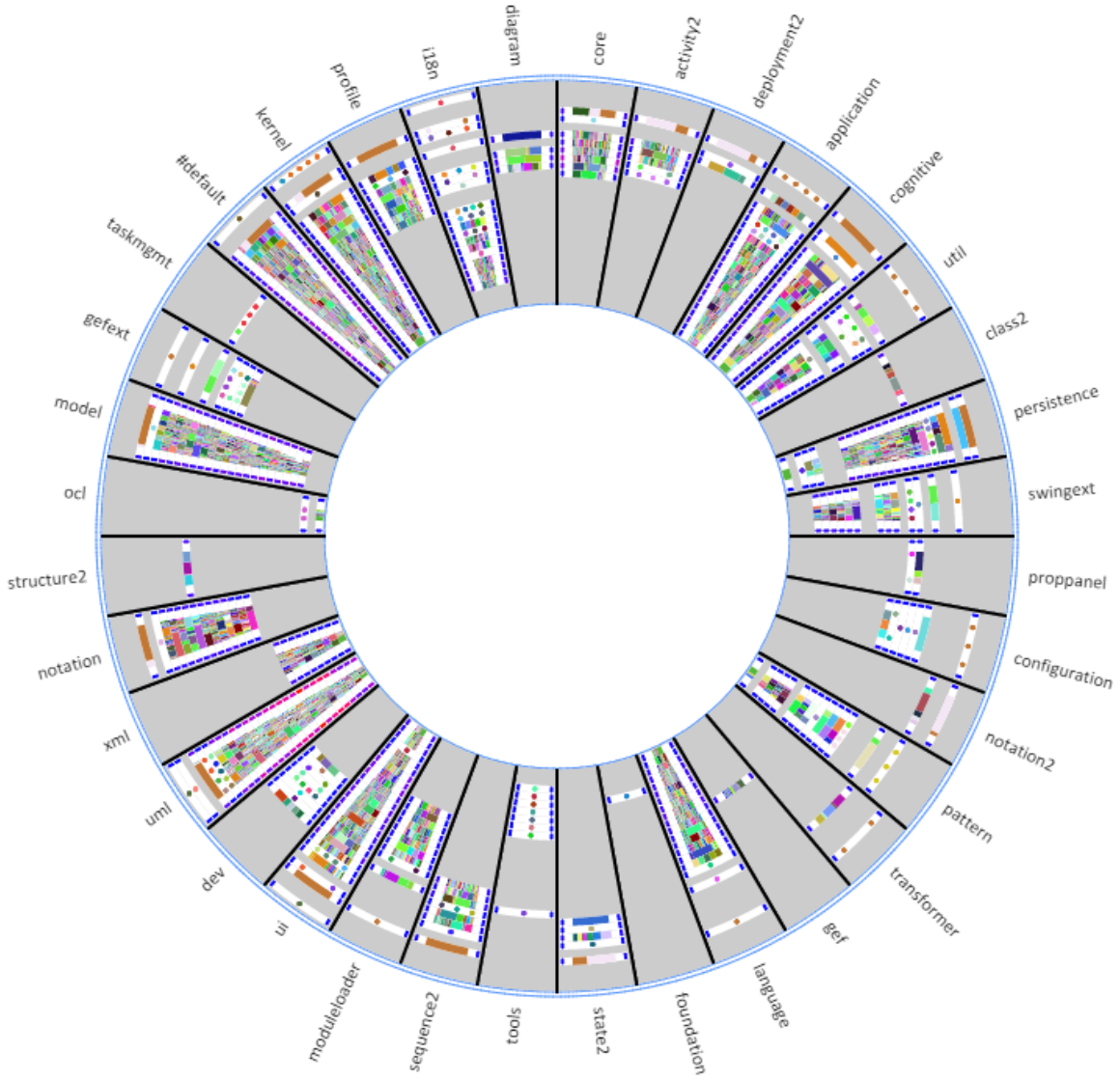


Figure 4.10 EVOWAVE visualization with all module changes during fifteen years of development

To start investigating in more details, we need to analyze the logic coupling of the "uml" package with other system modules. Figure 4.12 illustrates the visualization to analyze the logical coupling of the "uml" package with other packages. First, we analyze packages with less changes (e.g., moduleloader, ocl, tools and, util) because it is easier to understand them due to the low amount of data.

The package "moduleloader" has a low logical coupling with the "uml" package. Nevertheless, in the fifth early month in the "moduleloader" package there were three changes in the same commit that changed almost all modules. Looking deeper into the comment

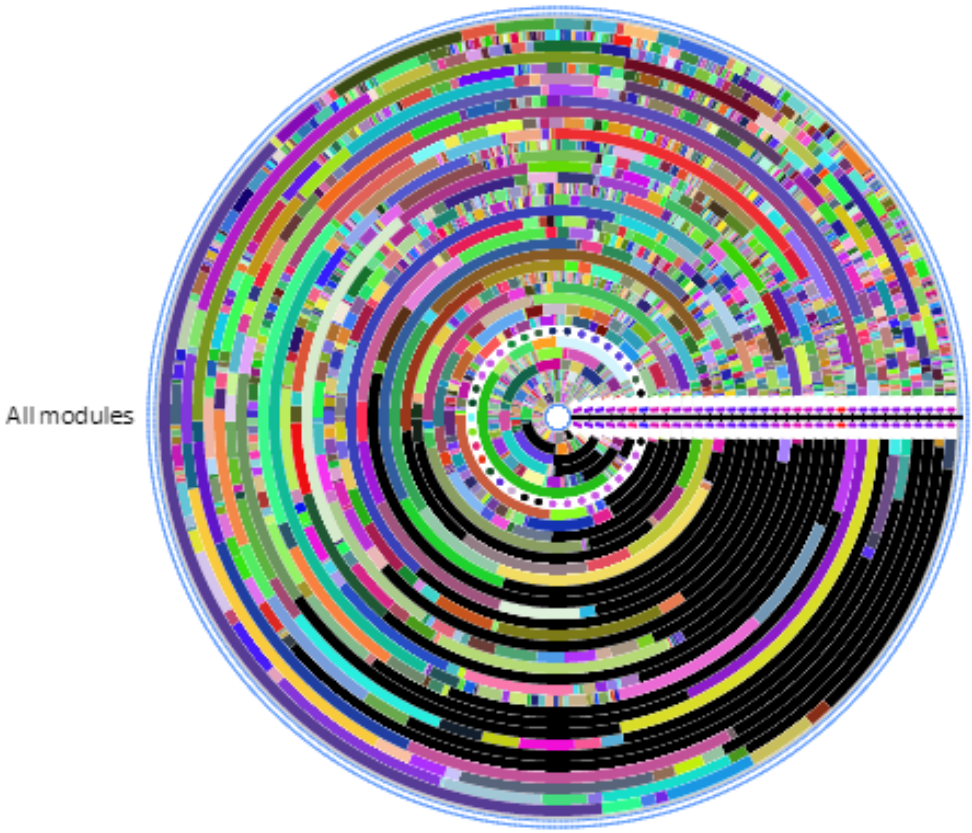


Figure 4.11 EVOWAVE visualization with all module changes from 2003 to 2005, focused on the "uml" package.

of this commit and into the changes, we identified they were related to the copyright style and impacted 1,065 files. In terms of source code these have no impact but let us know that probably every single file must change when there is a new copyright information.

The package "ocl" is highly coupled with the "uml" package. This is observed because there is almost no black color in the "ocl" sector. Among the changes, there are two commits highlights in the forth newer window: the green and red colors. The commit related to the green color is the copyright change that we demonstrate while analyzing the "moduleloader" package. The commit represented by the red color seems to impact many packages. While analyzing the comment and the changes from this commit we identified a major change in the system. The class responsible to be the facade for all communications with the "model" package, changed its signature from "ModelFacade" to "Model.getFacade()". This is a big change in the system because it breaks the signature used in many packages (e.g., ocl, persistence, uml). The logical coupling between the "ocl" package and the "uml" package for this commit is a consequence of their coupling with the model package.

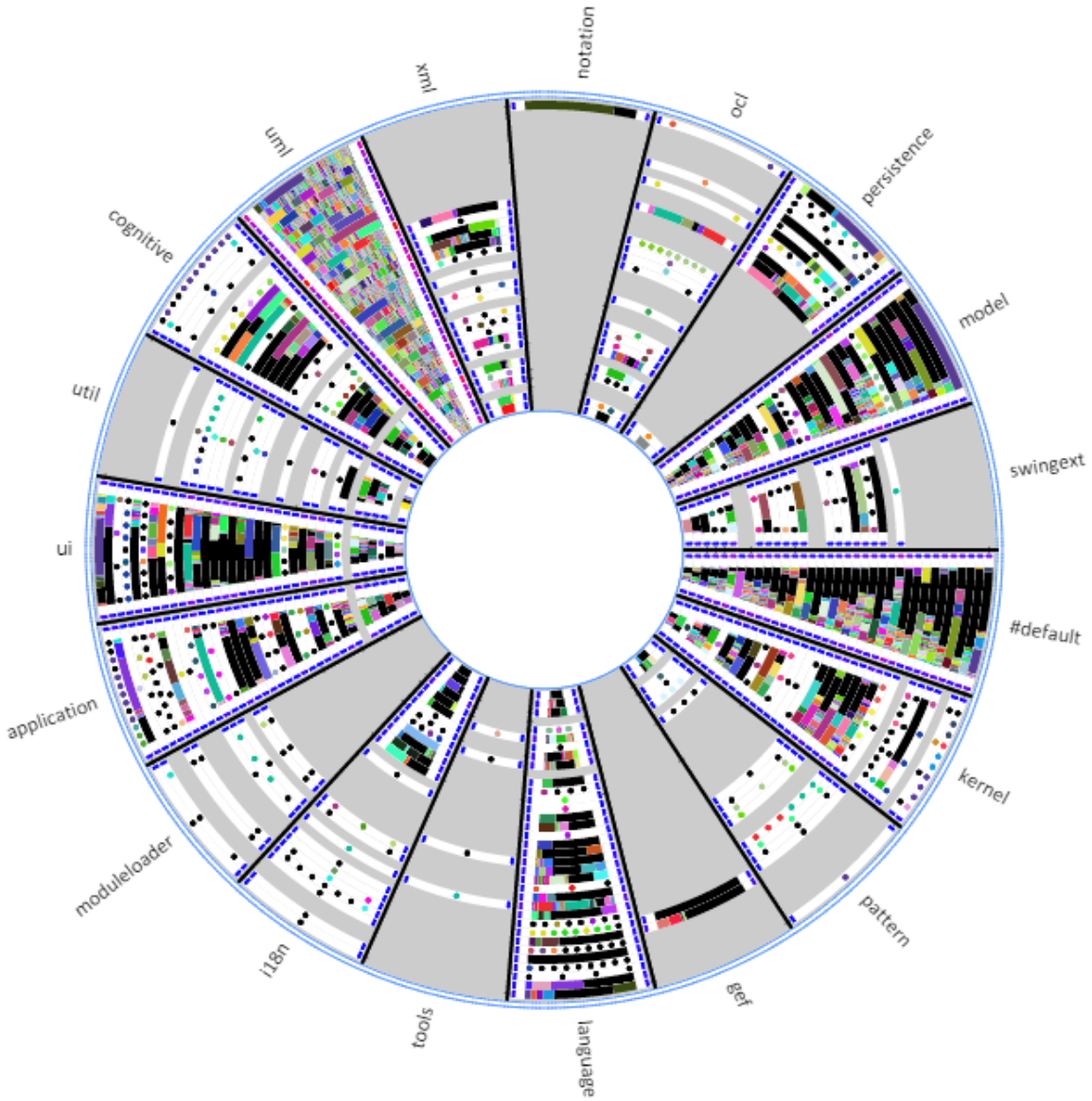


Figure 4.12 EVOWAVE visualization with all changes separed by modules from 2003 to 2005 with the "uml" package in focus.

The "model" package holds a lot of data because it changed many times during this period. In this case, we filtered the visualization to show only the "uml" package and the "model" package in order to better understand their logical coupling. Figure 4.13 represents the visualization with this filter. The first thing to notice is the amount of changes in those packages. Clearly the "uml" package had more changes than the "model" package during this period. Another thing to notice is the frequency of black and other colors. Black molecules are present in practically all months. Nevertheless, there are many colors present in the windows leading to the conclusion that many commits were

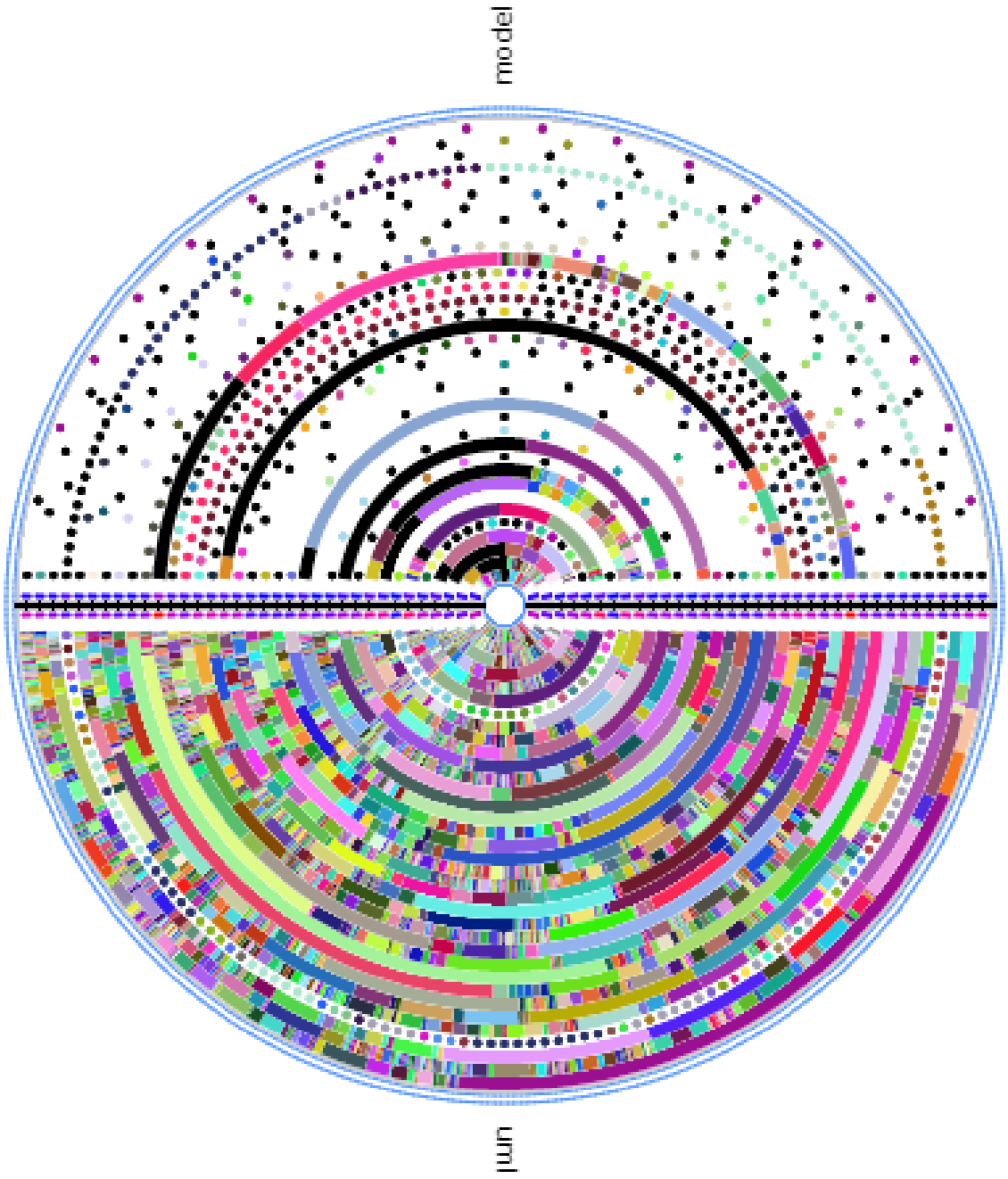


Figure 4.13 EVOWAVE visualization to analyze the logical coupling between the "uml" and "model" packages.

submitted in that month, and are logical coupled with the "uml" package. The eleventh newest window (January, 2005) has the highest number of colors, and was one of the months with most changes. Looking deeper into the commits submitted in that month,

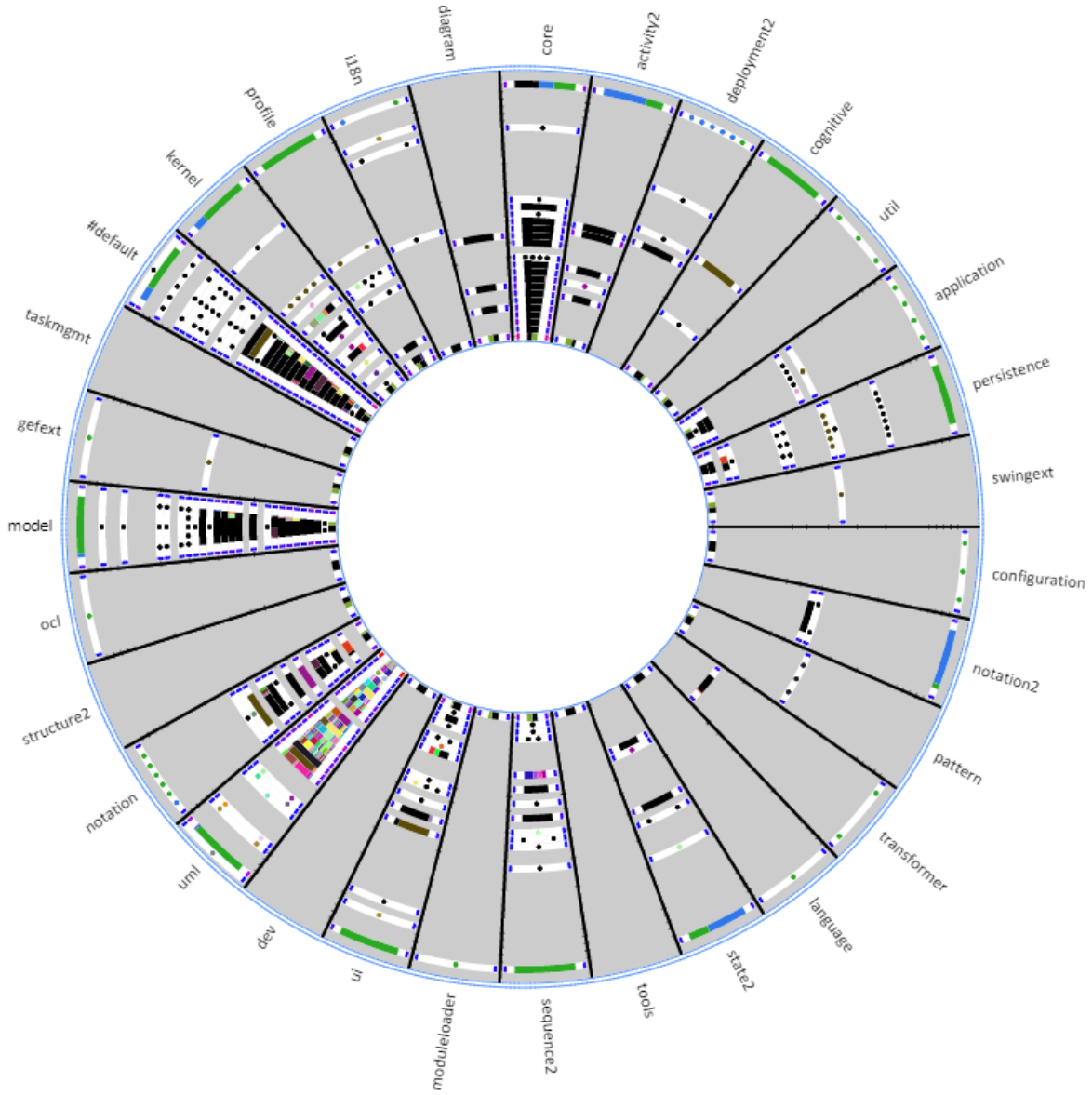


Figure 4.14 EVOWAVE visualization from January 2, 2010 to February 8, 2013

we found that those packages are coupled by an architecture decision to have a Facade pattern to access the "model" package. The Facade pattern can be defined as an entry point to features joined in a module to reduce the need for one module to known another module completely. This pattern being the reason for the coupling of these two packages is a good sign of good architectural decisions throughout the software development.

We filtered the period limiting it from January 2, 2010 to February 8, 2013 in order to analyze the logical coupling of the package "uml" in more recent data. Figure 4.14 illustrates the visualization for this period. The first thing we noticed was the reduction of the logical coupling among the "uml" package and the rest of the system. The reason for this may be refactorings performed during 2005 and 2010 to reduce the coupling

between the "uml" package and the rest of the system. Nevertheless, the green commit at the second newest window stands out in this visualization. It impacts practically all active modules in the system. Looking more deeply into the commit and its changes, we identify that the reason was a change in the logging library from log4j to the native java logging API.

4.3.3 Conclusion

This study was performed using the task proposed by Ambros (D'ambros; Lanza; Lungu, 2009) using the same artifact, the ArgoUML open source project. This study demonstrated that the EVOWAVE metaphor was able to portray a huge amount of data and to guide the software engineer from the visualization to the changes in the code, in order to extract information. Consequently, we could display data from the logical coupling domain and perform a retrospective analysis of the system. We conclude that the EVOWAVE metaphor can be used to understand the software evolution using logical coupling data.

4.4 LIMITATIONS

During the execution of the three studies some limitations regarding the planning and execution of the studies was highlighted. We better describe those limitations bellow:

- **The quantity of domains used**: To perform a study in every software engineering domain would require a huge amount of resources (e.g., time, personal, data available for each domain and, etc.) not available in this work. To minimize this limitation we choose three domains that need different information to perform software engineering tasks. The use of generic concepts that is transversal to most of domains, such as events (molecules), was another way to minimize this limitation. The use of generic concepts increases the capability to represent different domains.

- **The type of experimental method**: Controlled experiments were more indicated then the exploratory studies for this validation. Nevertheless, we did not found a work that proposed a metaphor that visualize multiple domains to design a study were one group uses the EVOWAVE and the other with the other similar metaphor. We tried to design a study with two groups: one using the EVOWAVE and the other group using one tool for each domain. However, the artifact of this study should be the same and it was not possible to find a single artifact that had data from each domain and are storage in a readable format for each tool. The findbugs project for example, has information about the three domains explored here but since it's code is storage in a SVN repository, the evolution radar tool for logical coupling can not read the data (only CVS). We search for the source code but most of them is not available in the Internet. Even if the code was available, a huge effort would be necessary to modify the tools in order to support a common database. To minimize this limitation we choose domains, tasks and artifacts used by previously software evolution works. We used oracles such as the SVN or GIT

log files for Collaboration and Logical Coupling domains and the Maven Repository for the Library Dependency domain to certify the information extracted.

- **The usability validation of the tool**: The studies was performed by the researchers of this work. We had previously knowledge about the metaphor concepts and the tool. This is a limitation because the efficiency and effectiveness of the tool was not validated by subjects outside this work. To minimize this limitation the EVOWAVE concepts has common standards in software visualization. For example, the radial layout is used in many visualization such as the evolution radar (D'ambros; Lanza; Lungu, 2009) and the SDP (Kula et al., 2014). The concept of hierarchical structure used as zoom logical when the user click in the sector, was proposed by Shneiderman (Shneiderman, 1992) in his Treemap metaphor. An average of three user interactions was needed in the tool to perform the studies tasks. This fact reduces the impact of this limitation.

4.5 CHAPTER CONCLUSION

Domains	Tasks \ Works	EVOWAVE	Fritz	Kula	D'Ambros
Software Collaboration Domain	Who is working on what?	OK	OK	NOT OK	NOT OK
	How much work have people done?	OK	OK	NOT OK	NOT OK
	What classes have been changed?	OK	OK	NOT OK	NOT OK
	Who has the knowledge to do the code review?	OK	OK	NOT OK	NOT OK
	Who is working on the same classes as I am and for which work item?	OK	OK	NOT OK	NOT OK
Library Dependency Domain	Understand the regularity of system dependency changes	OK	NOT OK	OK	NOT OK
	Understand what important structural dependency events have occurred	OK	NOT OK	OK	NOT OK
	Discover the current "attractiveness" of any library version	OK	NOT OK	OK	NOT OK
	Discover if newer releases are viable candidates for updating	OK	NOT OK	OK	NOT OK
Logical Coupling Domain	Retrospective Analysis	OK	NOT OK	NOT OK	OK

OK: Task was performed | NOT OK: Task was not performed

Figure 4.15 Table comparing the tasks performed by each work used in the studies and EVOWAVE.

This chapter presented three exploratory studies to validate the use of the EVOWAVE metaphor in multiple domains. Each one was based in previous works and used different

data to perform their tasks. These previously works uses metaphors with specific concepts to successfully perform tasks related to their domain in focus. Those specific concepts are highly associated to a domain property. The work referenced in our third study uses the distance between the class and the center as the logical coupling between the class and the package in focus (Chapter 2 has more information about this work). That is why the metaphor proposed by him is only used in the logical coupling domain. Our metaphor concepts uses properties in common among software engineering domains. For example, in most of these domains there is a event that can become a molecule and there are a structural hierarchy that can be the sectors. The conception of the EVOWAVE concepts thinking in common properties between domains was the main reason that made it able to portray a huge amount of data generated from software developments with up to fifteen years of changes. At Figure 4.15 we present a table comparing the tasks performed by each work used in the studies and EVOWAVE. As we can see, the tasks executed by the works used in our studies are driven by a software engineering domain. No publication was found of these works in different domains.

Chapter 5

This chapter presents the conclusion of this work highlighting the contributions, limitations, and future works.

CONCLUSION

Software evolution is one of the major software engineering topics nowadays. It helps software engineers to better understand the software by analysing decisions made in the past to make better decisions during the software evolution. This analysis deals with a huge amount of data and tasks for different domains. Software visualization has been used to better represent this data in order to help the software engineers with their daily activities.

This dissertation presented a new software evolution visualization metaphor that can represent a huge amount of data from different domains. We started from a mapping study, in the field of software evolution visualization, where we identified the use of a huge amount of metaphors for many domains, and sometimes, for the same domain. The use of different metaphors increases the learning curve needed to use them before actually helping in the software engineer's daily tasks. We searched and identified a well-known phenomenon that can represent a huge amount of evolutionary data. Then, we studied some facts about the phenomenon that could help us in the metaphor concepts. With the facts, we created generic concepts that could be used in many domains and implemented a prototype with them, originating the EVOWAVE metaphor. The metaphor was validated in three different domains helping in tasks proposed by other authors.

5.1 CONTRIBUTIONS

The main contribution of this dissertation was the conception and development of a metaphor for software evolution visualization that can be used in multiple domains. The metaphor uses generic concepts that are transverse between domains without loosing the detailed data. It can be used to visualize the whole development period or just one day. The capacity of representing data from different domains reduces the learning curve necessary to understand how the visualization works. During the development, this dissertation produced the following contributions:

- Specification and creation of a metaphor that can represent different types of data, from the overview to the details, for different domains.
- Development of a tool implementing the metaphor.
- Development of algorithms that extracted data from different types of repositories.
- Experimental studies that validate the use of the metaphor in multiple domains.

5.2 LIMITATIONS

Some limitations were identified during this work. The main limitation were identified during the studies (Section 4.4). In addition to these, the following limitations can be pointed out:

- **The color palette of the molecules**: The palette can be too extensive making some of them difficult to distinguish visually. To minimize this limitation we used the CIELab and Lch color space (Wijffelaars et al., 2008) that successfully produces colors to represent qualitative data.
- **The number of sectors**: It can not be too extensive because the angle will be too small to display any data in it. To minimize this limitation the user can hide sectors using filters. This is a common limitation in the area of software visualization and most solutions use filters.
- **The technology chosen**: It can not display a visualization that exceeds 32,767 pixels in width or height and its area can not be bigger then 268,435,456 pixels. This is not a limitation of the metaphor, therefore we always will be limited of the power of the hardware and the software used. In this case is a limitation of the HTML Canvas implemented in the browser.

5.3 FUTURE WORKS

As future works, the following activities can be cited:

- Improve the EVOWAVE concepts to visualize event sequences (e.g., molecules linked by arrows according to its sequence).
- Improve the EVOWAVE concepts to visualize evolutionary data with geographic information.
- Validate the EVOWAVE to other domains outside the software engineering area.
- Migrate to a desktop application to improve its performance.

BIBLIOGRAPHY

Abramson, D.; Sosic, R. A debugging tool for software evolution. In: *Computer-Aided Software Engineering, 1995. Proceedings., Seventh International Workshop on*. [S.l.: s.n.], 1995. p. 206 –214.

Atlassian. *JIRA Software - Issue Project Tracking for Software Teams*. 2015. Retrieved from https://www.atlassian.com/software/jira.

Ball, T.; Eick, S. Software visualization in the large. *Computer*, v. 29, n. 4, p. 33–43, Apr 1996. ISSN 0018-9162.

Basili, V. R.; Rombach, H. D. The tame project: Towards improvement-oriented software environments. *IEEE Trans. Softw. Eng.*, IEEE Press, Piscataway, NJ, USA, v. 14, n. 6, p. 758–773, jun. 1988. ISSN 0098-5589. Disponível em: <http://dx.doi.org/10.1109/32.6156>.

Bederson, B. B. Fisheye menus. In: *Proceedings of the 13th annual ACM symposium on User interface software and technology*. New York, NY, USA: ACM, 2000. (UIST '00), p. 217–225. ISBN 1-58113-212-3. Disponível em: <http://doi.acm.org/10.1145/354401.354782>.

Bergel, A. et al. Execution profiling blueprints. *Software: Practice and Experience*, John Wiley & Sons, Ltd, p. n/a–n/a, 2011. ISSN 1097-024X. Disponível em: <http://dx.doi.org/10.1002/spe.1120>.

Beyer, D.; Hassan, A. E. Animated visualization of software history using evolution storyboards. In: *Proceedings of the 13th Working Conference on Reverse Engineering*. Washington, DC, USA: IEEE Computer Society, 2006. (WCRE '06), p. 199–210. ISBN 0-7695-2719-1. Disponível em: <http://dx.doi.org/10.1109/WCRE.2006.14>.

Briand, L. The experimental paradigm in reverse engineering: Role, challenges, and limitations. In: *Reverse Engineering, 2006. WCRE '06. 13th Working Conference on*. [S.l.: s.n.], 2006. p. 3–8. ISSN 1095-1350.

Brito, I. S.; Moreira, A. M. Advanced separation of concerns for requirements engineering. In: *JISBD*. [S.l.: s.n.], 2003. p. 47–56.

Brooks, R. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, v. 18, n. 6, p. 543–554, jun. 1983. ISSN 00207373. Disponível em: <http://dx.doi.org/10.1016/s0020-7373(83)80031-5>.

Bugzilla. *Bugzilla - A server designed to help manage software development.* 2015. Retrieved from https://www.bugzilla.org/.

Carneiro, G. *SourceMiner: Um Ambiente Integrado para Visualização Multi-Perspectiva de Software*. Tese (Doutorado) — Federal University of Bahia, 2011.

Carneiro, G. de F.; Magnavita, R.; Mendonça, M. Combining software visualization paradigms to support software comprehension activities. In: *Proceedings of the 4th ACM Symposium on Software Visualization*. New York, NY, USA: ACM, 2008. (SoftVis '08), p. 201–202. ISBN 978-1-60558-112-5. Disponível em: <http://doi.acm.org/10.1145/1409720.1409755>.

Carneiro, G. de F. et al. Identifying code smells with multiple concern views. In: *Software Engineering (SBES), 2010 Brazilian Symposium on*. [S.l.: s.n.], 2010. p. 128–137.

Caserta, P.; Zendra, O. Visualization of the Static Aspects of Software: A Survey. *Visualization and Computer Graphics, IEEE Transactions on*, PP, n. 99, 2010. Disponível em: <http://dx.doi.org/10.1109/TVCG.2010.110>.

Corbi, T. A. Program understanding: Challenge for the 1990s. *IBM Systems Journal*, v. 28, n. 2, p. 294–306, 1989. ISSN 0018-8670.

Cornelissen, B. et al. Understanding execution traces using massive sequence and circular bundle views. In: *Program Comprehension, 2007. ICPC '07. 15th IEEE International Conference on*. [S.l.: s.n.], 2007. p. 49–58. ISSN 1092-8138.

D'ambros, M.; Lanza, M.; Lungu, M. Visualizing co-change information with the evolution radar. *IEEE Trans. Softw. Eng.*, IEEE Press, Piscataway, NJ, USA, v. 35, n. 5, p. 720–735, set. 2009. ISSN 0098-5589. Disponível em: <http://dx.doi.org/10.1109/TSE.2009.17>.

Detten, M.; Platenius, M.; Becker, S. Reengineering component-based software systems with archimetrix. *Software Systems Modeling*, Springer-Verlag, p. 1–30, 2013. ISSN 1619-1366. Disponível em: <http://dx.doi.org/10.1007/s10270-013-0341-9>.

Deursen, A. van et al. Viewpoints in software architecture reconstruction. In: BAD HONNEF. *Proceedings 6th Workshop on Software Reengineering (WSR)*. [S.l.], 2004.

Diehl, S. *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2007. ISBN 3540465049.

Encyclopaedia-britannica-online. *Napoleon I: statistical map of russian campaign.* 2013. Disponível em: <http://www.britannica.com/EBchecked/media/70821/Statistical-map-of-Napoleons-Russian-campaign-of-1812-The-size>.

Erlikh, L. Leveraging legacy system dollars for e-business. *IT Professional*, IEEE Educational Activities Department, Piscataway, NJ, USA, v. 2, n. 3, p. 17–23, maio 2000. ISSN 1520-9202. Disponível em: <http://dx.doi.org/10.1109/6294.846201>.

Ernst, E. *Higher-Order Hierarchies*. [S.l.: s.n.], 2003. 303–328 p.

Facebook. *Visualizing Friendships*. 2013. Disponível em: <https://www.facebook.com/note.php?note_id = 469716398919¿.

Fontana, F. A.; Zanoni, M. A tool for design pattern detection and software architecture reconstruction. *Information Sciences*, v. 181, n. 7, p. 1306 – 1324, 2011. ISSN 0020-0255. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0020025510005955>.

Fritz, T.; Murphy, G. C. Using information fragments to answer the questions developers ask. In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*. New York, NY, USA: ACM, 2010. (ICSE '10), p. 175–184. ISBN 978-1-60558-719-6. Disponível em: <http://doi.acm.org/10.1145/1806799.1806828>.

Fuhr, A. et al. Model-driven software migration into service-oriented architectures. *Comput. Sci.*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, v. 28, n. 1, p. 65–84, fev. 2013. ISSN 1865-2034. Disponível em: <http://dx.doi.org/10.1007/s00450-011-0183-z>.

Furnas, G. W. Generalized fisheye views. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. New York, NY, USA: ACM, 1986. (CHI '86), p. 16–23. ISBN 0-89791-180-6. Disponível em: <http://doi.acm.org/10.1145/22627.22342>.

Gamma, E. et al. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN 0-201-63361-2.

Godfrey, M.; Tu, Q. Growth, evolution, and structural change in open source software. In: *Proceedings of the 4th International Workshop on Principles of Software Evolution*. New York, NY, USA: ACM, 2001. (IWPSE '01), p. 103–106. ISBN 1-58113-508-4. Disponível em: <http://doi.acm.org/10.1145/602461.602482>.

Greevy, O.; Lanza, M.; Wysseier, C. Visualizing live software systems in 3d. In: *In Proceedings of the ACM Symposium on Software Visualization*. [S.l.]: ACM Press, 2006. p. 47–56.

Jay, G. et al. Cyclomatic complexity and lines of code: Empirical evidence of a stable linear relationship. *JSEA*, v. 2, n. 3, p. 137–143, 2009.

jedit. *jEdit - Programmer's Text Editor*. 2015. Retrieved from http://jedit.org/.

Johnson, C.; Hansen, C. *Visualization Handbook*. Orlando, FL, USA: Academic Press, Inc., 2004. ISBN 012387582X.

Kuhn, A. et al. Software cartography: thematic software visualization with consistent layout. *J. Softw. Maint. Evol.*, John Wiley & Sons, Inc., New York, NY, USA, v. 22, n. 3, p. 191–210, abr. 2010. ISSN 1532-060X. Disponível em: <http://dx.doi.org/10.1002/smr.v22:3>.

Kula, R. et al. Visualizing the evolution of systems and their library dependencies. In: *Software Visualization (VISSOFT), 2014 Second IEEE Working Conference on*. [S.l.: s.n.], 2014. p. 127–136.

Lanza, M. *Combining Metrics and Graphs for Object Oriented Reverse Engineering*. 1999.

Lanza, M. The evolution matrix: Recovering software evolution using software visualization techniques. In: *Proceedings of the 4th International Workshop on Principles of Software Evolution*. New York, NY, USA: ACM, 2001. (IWPSE '01), p. 37–42. ISBN 1-58113-508-4. Disponível em: <http://doi.acm.org/10.1145/602461.602467>.

Lanza, M. Codecrawler - polymetric views in action. In: *Automated Software Engineering, 2004. Proceedings. 19th International Conference on*. [S.l.: s.n.], 2004. p. 394–395. ISSN 1938-4300.

Lehman, M. M. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, v. 68, n. 9, p. 1060–1076, 1980.

Lehman, M. M.; Ramil, J. F. Rules and tools for software evolution planning and management. *Ann. Softw. Eng.*, J. C. Baltzer AG, Science Publishers, Red Bank, NJ, USA, v. 11, n. 1, p. 15–44, nov. 2001. ISSN 1022-7091. Disponível em: <http://dx.doi.org/10.1023/A:1012535017876>.

Lethbridge, T. C.; Singer, J.; Forward, A. How software engineers use documentation: The state of the practice. *IEEE Softw.*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 20, n. 6, p. 35–39, nov. 2003. ISSN 0740-7459. Disponível em: <http://dx.doi.org/10.1109/MS.2003.1241364>.

Letovsky, S. Cognitive processes in program comprehension. In: *Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers*. Norwood, NJ, USA: Ablex Publishing Corp., 1986. p. 58–79. ISBN 0-89391-388-X. Disponível em: <http://dl.acm.org/citation.cfm?id=21842.28886>.

Lungu, M. Towards reverse engineering software ecosystems. In: *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*. [S.l.: s.n.], 2008. p. 428 –431. ISSN 1063-6773.

Magnavita, R.; Novais, R.; Mendonça, M. Using evowave to analyze software evolution. In: *Proceedings of the 17th International Conference on Enterprise Information Systems*. [S.l.: s.n.], 2015. p. 126–136. ISBN 978-989-758-097-0.

Mayrhauser, A. von; Vans, A. M. Program comprehension during software maintenance and evolution. *Computer*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 28, n. 8, p. 44–55, ago. 1995. ISSN 0018-9162. Disponível em: <http://dx.doi.org/10.1109/2.402076>.

Mazza, R. *Introduction to Information Visualization*. 1. ed. [S.l.]: Springer Publishing Company, Incorporated, 2009. ISBN 1848002181, 9781848002180.

Ncsa. *Streaming Video and Sample Animation Images at Reduced Resolution from the 1995 IMAX Film "Stormchasers"*. 2013. Disponível em: <http://redrock.ncsa.illinois.edu/AOS/imax.html>.

Novais, R. et al. An interactive differential and temporal approach to visually analyze software evolution. In: *Visualizing Software for Understanding and Analysis (VISSOFT), 2011 6th IEEE International Workshop on*. [S.l.: s.n.], 2011. p. 1–4.

Novais, R. et al. On the proactive and interactive visualization for feature evolution comprehension: An industrial investigation. In: *Software Engineering (ICSE), 2012 34th International Conference on*. [S.l.: s.n.], 2012. p. 1044–1053. ISSN 0270-5257.

Novais, R. L. *Combinando Estratégias de Análise Visual de Evolução de Software em Diferentes Níveis de Detalhe*. Tese (Doutorado) — Federal University of Bahia, 2013.

Novais, R. L. et al. Software evolution visualization: A systematic mapping study. *Inf. Softw. Technol.*, Butterworth-Heinemann, Newton, MA, USA, v. 55, n. 11, p. 1860–1883, nov. 2013. ISSN 0950-5849. Disponível em: <http://dx.doi.org/10.1016/j.infsof.2013.05.008>.

Nova/wgbh. *Stormchasers*. 1995. M.F. Films.

Olivera, F. R. *MVNrepository*. 2015. Retrieved from http://mvnrepository.com/.

Pennington, N. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, v. 19, n. 3, p. 295 – 341, 1987. ISSN 0010-0285. Disponível em: <http://www.sciencedirect.com/science/article/pii/0010028587900077>.

PHILOSOPHICAL Magazine. Taylor & Francis., 1906. ISSN 0031-8086. Disponível em: <http://books.google.com.br/books?id=x5EOAAAAIAAJ>.

Pigoski, T. M. *Practical Software Maintenance: Best Practices for Managing Your Software Investment*. New York, NY, USA: John Wiley & Sons, Inc., 1996. ISBN 0471170011.

Processingjs. *A port of the Processing Visualization Language*. 2015. Retrieved from http://processingjs.org/.

Rajlich, V.; Bennett, K. A staged model for the software life cycle. *Computer*, v. 33, n. 7, p. 66–71, Jul 2000. ISSN 0018-9162.

Riel, A. J. *Object-Oriented Design Heuristics*. 1st. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1996. ISBN 020163385X.

Rufiange, S.; Melancon, G. Animatrix: A matrix-based visualization of software evolution. In: *Software Visualization (VISSOFT), 2014 Second IEEE Working Conference on*. [S.l.: s.n.], 2014. p. 137–146.

Shneiderman, B. Tree visualization with tree-maps: 2-d space-filling approach. *ACM Trans. Graph.*, ACM, New York, NY, USA, v. 11, n. 1, p. 92–99, jan. 1992. ISSN 0730-0301. Disponível em: <http://doi.acm.org/10.1145/102377.115768>.

Shneiderman, B. The eyes have it: A task by data type taxonomy for information visualizations. In: *Proceedings of the 1996 IEEE Symposium on Visual Languages*. Washington, DC, USA: IEEE Computer Society, 1996. (VL '96), p. 336–. ISBN 0-8186-7508-X. Disponível em: <http://dl.acm.org/citation.cfm?id=832277.834354>.

Shneiderman, B.; Mayer, R. Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Computer & Information Sciences*, Springer, v. 8, n. 3, p. 219–238, 1979.

Soloway, E.; Adelson, B.; Ehrlich, K. Knowledge and Processes in the Comprehension of Computer Programs. *The Nature of Expertise*, A. Lawrence Erlbaum Associates, Hillside, NJ, p. 129–152, 1988.

Souza rodrigo, B. S.-T. M.; Manoel, M. Skyscrapar: an augmented reality visualization for software evolution. *II Brazilian Workshop on Software Visualization*, p. 17–24, 2012. ISSN 0098-5589.

Staples, M. L.; Bieman, J. M. 3–d visualization of software structure. In: Zelkowitz, M. V. (Ed.). Elsevier, 1999, (Advances in Computers, v. 49). p. 95 – 141. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0065245808602843>.

Steinbrückner, F.; Lewerentz, C. Representing development history in software cities. In: *Proceedings of the 5th international symposium on Software visualization*. New York, NY, USA: ACM, 2010. (SOFTVIS '10), p. 193–202. ISBN 978-1-4503-0028-5. Disponível em: <http://doi.acm.org/10.1145/1879211.1879239>.

Teyseyre, A.; Campo, M. An overview of 3d software visualization. *Visualization and Computer Graphics, IEEE Transactions on*, v. 15, n. 1, p. 87–105, Jan 2009. ISSN 1077-2626.

Tufte, E. R. *The visual display of quantitative information*. Cheshire, CT, USA: Graphics Press, 1986. ISBN 0-9613921-0-X.

Voinea, L.; Telea, A. Multiscale and multivariate visualizations of software evolution. In: *Proceedings of the 2006 ACM symposium on Software visualization*. New York, NY, USA: ACM, 2006. (SoftVis '06), p. 115–124. ISBN 1-59593-464-2. Disponível em: <http://doi.acm.org/10.1145/1148493.1148510>.

Ware, C. *Information Visualization, Second Edition: Perception for Design (Interactive Technologies)*. 2. ed. Morgan Kaufmann, 2004. Hardcover. ISBN 1558608192. Disponível em: <http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/1558608192>.

Wettel, R.; Lanza, M. Codecity: 3d visualization of large-scale software. In: *Companion of the 30th international conference on Software engineering*. New York, NY, USA: ACM, 2008. (ICSE Companion '08), p. 921–922. ISBN 978-1-60558-079-1. Disponível em: <http://doi.acm.org/10.1145/1370175.1370188>.

Wijffelaars, M. et al. Generating color palettes using intuitive parameters. In: *Proceedings of the 10th Joint Eurographics / IEEE - VGTC Conference on Visualization*. Chichester, UK: The Eurographs Association & John Wiley & Sons, Ltd., 2008. (EuroVis'08), p. 743–750. Disponível em: <http://dx.doi.org/10.1111/j.1467-8659.2008.01203.x>.

Wilhelmson, R. Visualization of storm and tornado development for an omnimax film and for the cave. 1996.