On the Use of Software Visualization to Analyze Library Dependency Evolution: an Exploratory Study

Rodrigo Magnavita[1], Renato Novais [1, 2], Thiago Mendes[1], Manoel Mendonça[2]

[1]Fraunhofer Project Center @ UFBA
Salvador, BA.
[2]Instituto Federal da Bahia
Salvador, BA.

{rodrigo.magnavita, manoel.mendonca}@ufba.br,
{renato, thiagosouto}@ifba.edu.br

Abstract. *Software evolution produces a large amount of data. Software engineers may benefit from these data while carrying out daily activities. The challenge is to understand all the data and recovery valuable information. One of these activities is to maintain library dependencies of a project. This is a complex task, since it is usual a system and its library dependencies evolving separately. It is necessary to be careful when updating the libraries of a project. The maintainers need to know the history about a system´s past upgrade decisions and which dependencies were adopted at the same time. Software Evolution Visualization (SEV) can be a promising approach to this end. In this paper, we present an exploratory study that uses a SEV tool, called EVOWAVE, to analyze Library Dependency Evolution. EVOWAVE is able to visualize different types of data generated in software evolution using both overview-based and detail-based approaches. In the study, we performed an analysis of FindBug's library dependency history. The results highlighted the benefits of the SEV tool on analyzing a large amount of data that encompasses more than 10 years.*

1. Introduction

Large projects may depend on several third-party libraries. They are important to the project as they solve part of the projects' requirements. Both projects and libraries evolve, making it challenger to manage the configuration of a project. In this sense, to maintain library dependencies of a project is a key – and complex – task. Usually, a system and its library dependencies evolve separately. It is necessary to be careful when updating the libraries of a project. The maintainers need to comprehend the history about a system´s past upgrade decisions and which dependencies were adopted at the same time.

Software visualization (SoftVis) has been used to deal with software comprehension activities, such as the ones related to library dependency. It is expected that, when one improves the software comprehensibility, software maintenance tasks are smoother performed, and the overall quality of the systems increases. SoftVis helps people to understand software through visual elements, reducing complexity to analyze the large amount of data generated during the software evolution [Diehl 2007]. Some examples of what this data can be are: software metrics, stakeholders, bugs, features, and library dependency.

In the context of software evolution visualization (SEV), researchers generally use two different approaches. The first presents the big picture of the software, providing an overview of the entire software history [Kuhn et al. 2010] [Voinea and Telea 2006] [Lungu 2008], while the second shows snapshots of the software evolution in detail [D'Ambros et al. 2009] [Novais et al. 2011] [Novais et al. 2012] [Bergel et al. 2011]. Both approaches are important since each one 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 proposed visualizations [Novais et al. 2013].

In the last two years, we have been working on the specification, development and evaluation of a novel SEV tool, called EVOWAVE [Magnavita 2016]. EVOWAVE realizes a novel visualization metaphor [Magnavita et al. 2015]. It is able to visualize different types of data generated in software evolution using both overview and detail approaches. EVOWAVE can represent a huge number of events at a glance. Several mechanisms of interactions allow the user to explore the visualization in detail. EVOWAVE has one perspective, implemented in a circular layout. This open source tool can be applied to different software engineering tasks and contexts. To validate its benefits, we conducted three exploratory studies using EVOWAVE in three different software engineering contexts: software collaboration [Magnavita et al. 2015], library dependency, and logical coupling.

In this paper, we present the library dependency study. As in the other two studies, we selected a close related work [Kula et al. 2014], and replicated the study they conducted. Our goal was both to show the benefits of EVOWAVE in a new context (Library Dependency) considering an already published work, and also to comparably highlight the benefits of our tool.

Kula et al.'s work [Kula et al. 2014] visualizes how the dependency relation between a system and its dependencies evolve from two perspectives. Those two perspectives are realized by two different views. They demonstrated on real-world systems how maintainers can benefit from their visualizations through four case scenarios. In our case, we use the EVOWAVE, a multiple domain software evolution visualization metaphor, to perform the same study on the analysis of FindBug's library dependency history. To this end, we first had to extract a large data set from Maven Central Repository [Olivera 2015]. This data set was used to fulfill the EVOWAVE metaphor. As a results, this paper contributes by: i) showing the study on the FindBug's library dependency history using a single perspective software evolution visualization; and ii) providing an experimental package that includes the large data set we collected[1].

This paper is organized as follow. Section 2 presents the background in order to support the understanding of the conducted study. Section 3 describes the study we conducted to evaluate EVOWAVE in the Library Dependency context. Section 4 presents related work. Finally, Section 5 concludes this work.

2. Background

This section initially shows a background on the related topics of this work: library dependency and software visualization. At the end, we present an overview about the EVOWAVE tool.

[1] EVOWAVE website – http://wiki.ifba.edu.br/evowave

2.1. Library Dependency

Libraries are collections of components that solve certain recurring problems. Each of these problems is from a certain domain. Examples for such library domains include Graphical User Interfaces (GUI), database access, logging, XML parsing, communication, etc. Specially in the Java programming language, libraries are used quite intensively. The Java 6 runtime environment (JRE) alone contains about 16,000 classes, and there are thousands of 3rd party libraries available for many different purposes [Quante 2008].

Kula et al. (2014) defined a terminology and model for reasoning about dependencies between evolving systems and libraries: (i) **Versions** - refers to both system and library releases as versions. Conventions of releases are usually project specific; (ii) **Dependency Relations** - A dependency relation is when a system starts using a library as its dependent; and (iii) **Dependency Relations Change Types** - are system versions in terms of a change in the dependency relations: (a) An **adopter** system version starts using a library. An adopter system version starts using a library for the first time (i.e., it has not used previous versions); (b) An **idler** is a system version that depends upon the same library version as its immediate predecessor; and (c) An **updater** is a system version of which the previous version depended upon a different library version.

System maintainers face several challenges stemming from a system and its library dependencies evolving separately. Maintaining the libraries of the project is a complex task [Kula et al. 2014]. It is necessary to be careful when updating the libraries of a project. The maintainers need to know the history about a system's past upgrade decisions and which dependencies were adopted at the same time, may reveal the respective system structure. For this reason, novice dependencies or maintainers of undocumented systems may want to know the significant dependency changes, such as dropped and adopted libraries. In addition, for more experienced maintainers, it is useful to know the upgrade decisions of other systems regarding a common library they depend on. It may represent opportunities for upgrading to a newer version of a library as well as opportunities for migrating to a different library altogether.

2.2. Software visualization

Software visualization (SoftVis) is a subarea of information visualization [Mazza 2009], which visualizes abstract data generated in the software development process. SoftVis researchers are concerned with visualizing the structure, behavior, and evolution of the software [Diehl 2007]. The structure is all artifacts that were generated statically during the software development process. Source code, requirements, and test cases are examples of structures. Behavior refers to software behavior during its execution (at runtime). 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. Several works have been developed in this area.

In 1999, Lanza introduced the concept of polymetric views [Lanza 1999] to visualize the software structure. With his visualization tool, he was able to support the reverse engineering of software systems by visualizing the system structure with its relationships extracting this information from the source code. The polymetric views are

simple interactive graphs, enriched by various software metrics. Still in the 90s, Staples proposed a tri-dimensional exploration to visualize software structure [Staples and Bieman 1999]. The approach, named Change Impact Viewer, is a tri-dimensional matrix where the base (x and z axis) represents the system classes and the height (y axis) are the functions implemented by the class. Newer works started to explore the third dimension. CodeCity, for example, is one of the most famous 3D visualization tool [Wettel and Lanza 2008]. It represents the system as a 3D interactive urban environment. The city metaphor provides an overview of the system's structural organization by drawing the classes as buildings and the packages as districts.

One important challenge in visualizing the software behavior is the scalability. The 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, it is 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. The first one shows the system's structural decomposition and the nature of its interactions during the trace. 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. 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 software comprehension activities [Carneiro et al. 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].

As was briefly explained, software visualization can be used in different contexts and domains. The goal is to improve the understandability of the system at hand, in order to take better decisions aiming to improve its quality.

2.3. EVOWAVE

EVOWAVE is a new visualization tool 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. This section presents the concepts related to the proposed metaphor, which realizes EVOWAVE. Later, we explain how those concepts can be mapped to software properties and the tool's aspects of implementation.

2.3.1 EVOWAVE Concepts

Figure 1 presents the EVOWAVE concepts, which are explained bellow. **Layout**. EVOWAVE has a circular layout with two circular guidelines (inner and outer), as shown in Figure 1-A. They represent a software life cycle period between two selected dates. This period, named timeline (Figure 1-A), gives an overview of the software history. It 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 and users' configuration.

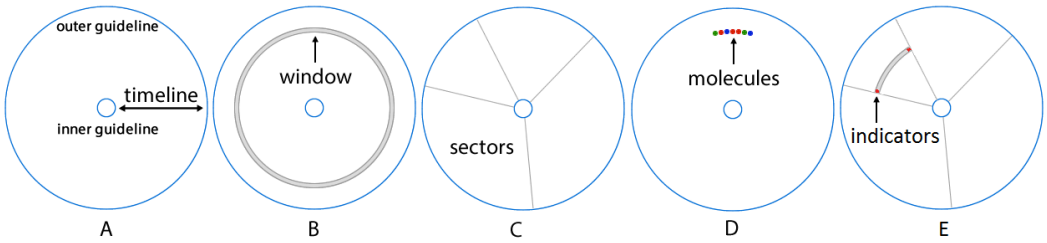

Figure 1. The EVOWAVE concepts.

To give some orientation to the path between the guidelines, the inner can be mapped to the oldest date and the outer to the newest date, or vice versa.

Windows. A window is a group of consecutive short periods (Figure 1-B). It is circular in shape and its length depends on the number of grouped periods. It can be used to compare a subset of short periods, making it possible to carry out a detailed analysis regarding the overall context.

Sectors. A sector is a visual element drawn between the two circular guidelines according to its angle (Figure 1-C). It may have different angles. This concept is used to group events that share some characteristic (e.g., classes of the same packages). Considering, for example, a sector representing a package, it is possible to interact and navigate getting detail on demand about the inner packages.

Molecules. Molecules are depicted as circular elements inside sectors and windows (Figure 1-D). Each molecule has an event associated to it. When we have more molecules than the display size limits, we gathered and drawn them as a quadrilateral polygon that fills the region where the molecules are. Molecules can represent any change on software history, such as file changes, team changes or bug reports.

Indicators. The number of molecules indicator is drawn as a rectangle located on the frontier of the sectors for each window (Figure 1-E). 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.

2.3.2 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 tasks at hand. Find bellow the EVOWAVE characteristics (visual attributes) that can be mapped to software history attributes (real). Figure 2 is one example of EVOWAVE showing a dependency usage of FindBugs project. For sake of understanding, this example is decorated (mockup) with labels pointing to the main concepts of the metaphor.

Timeline. The timeline defines the period of analysis through two dates. We can map two software versions and analyze what happened between them. If we map the

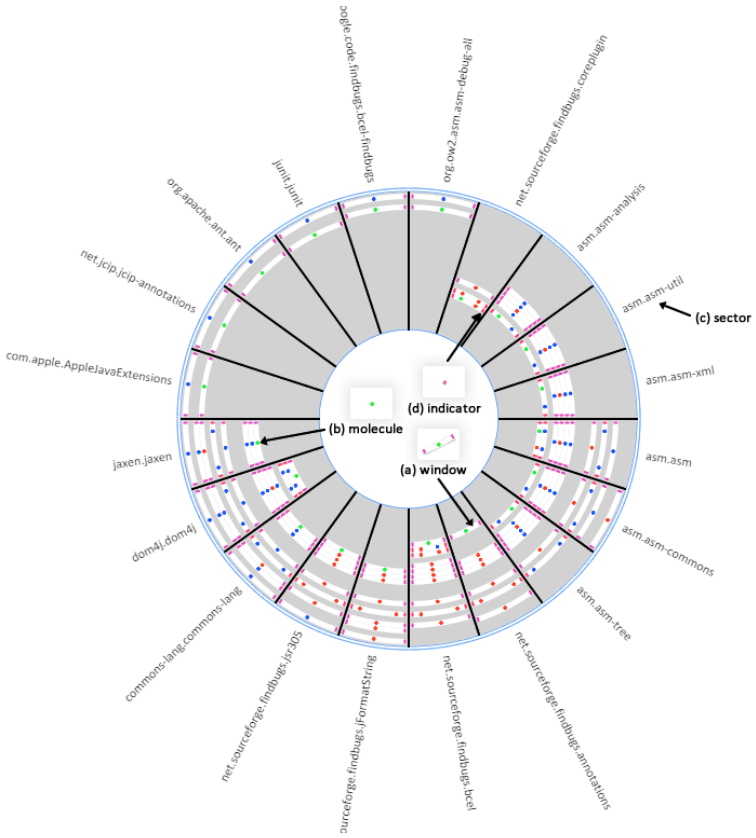

Figure 2. Evowave showing one example of FindBugs project's dependency usage.

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.

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. Some examples of software properties that can be mapped to a pooler are: the package of a changed class event; the file type of a changed file event; and the author of a bug report event.

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 where each one has a different level in the hierarchy.

The Angle of a Sector. The angle defines how much of some software property the sector has relatively to the others. If the pooler is a 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.

The Color of a Molecule. The color can 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 case, we can select two specific colors to decorate 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.

It is important to remark that EVOWAVE is highly configurable. In this sense, the user can select any visual attribute in the available set, and map to real attributes he/she has. In our opinion, the task at hand will guide this mapping.

2.3.3. Aspects of Implementation

Evowave architecture is designed to be simple and extensible. The client side uses a well-known JavaScript (JS) 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 (JS Object Notation). The JSON 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 extra 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. Every service provided by the server returns the data according to the visualization metadata. More details on the algorithms used can be found in [Magnavita 2016].

3. An Exploratory Study on Library Dependency Context

In 2015, we conducted an exploratory study to validate the use of EVOWAVE tool (Section 2.3) in the analysis of the evolution of how the dependency relation among a system and its dependencies evolves.

Using the GQM paradigm [Basili and Rombach 1988], the goal of this study was:

- **To analyze** the EVOWAVE metaphor;
- **With the purpose of** characterize;
- **Regarding the** dependency relationship evolution among 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.

3.1. Study Setting

In this exploratory study, we followed the case studies presented by Kula et al. [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 focused our evaluation for this system alone.

3.1.1. Setup

The information regarding FindBugs repository dependency is public available in Maven repository HTML pages. Therefore, we had to develop a HTML parser 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 (**5,5 GB**), 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[2], 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** represent 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.

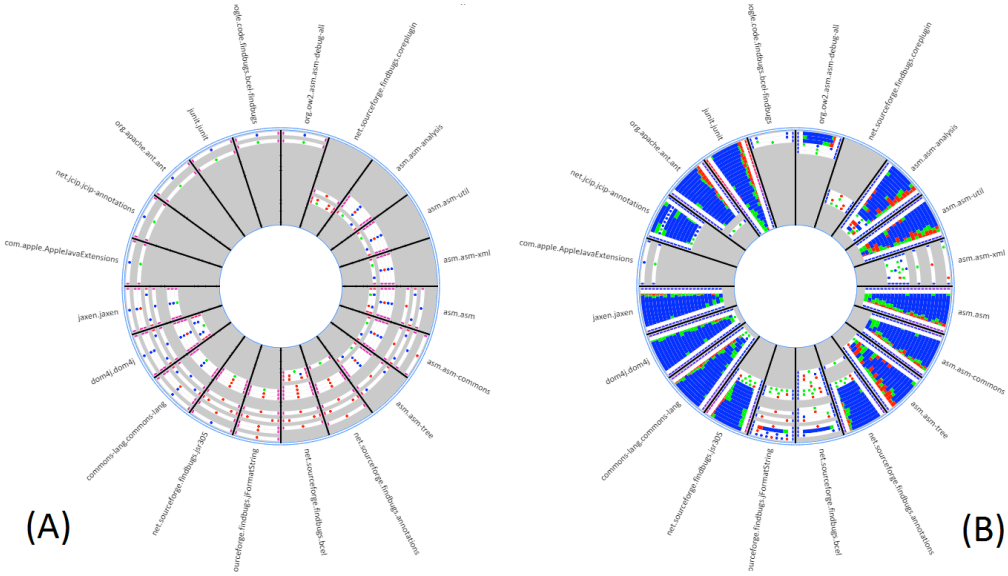

Figure 3. Examples of EVOWAVE visualizations for the Library Dependency Context: (A) The visualization is showing the use of all dependencies by FindBugs; (B) the use of all dependencies by any system.

Figure 3 has two examples of visualizations of the EVOWAVE metaphor with the described setup. In visualization (A), each molecule is a FindBugs system release that used the dependence represented in the sector. In visualization (B), each molecule

[2] This data was uploaded to the Internet and it can be downloaded at EVOWAVE website.

represents a release of any system registered in the Maven repository that used the dependence represented in the sector.

3.1.2. 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].

Four tasks were defined based on this scenario. The first two tasks help Rusty to understand the systems dependency structure to prioritize which libraries should be upgraded.

The last two tasks help him identifying suitable candidate library versions for upgrade [Kula et al. 2014]. The tasks and the motivation to answer them are described below:

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

3.2. 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 shows 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. In this case, we found different behaviors, namely: (i) the "com.apple.AppleJavaExtensions", "net.jcip.jcip-annotations", "org.apache.ant.ant", "junit.junit", "com.google.code.findbugs.bcel-findbugs", and "org.ow2.asm.asm-debug-all" dependencies (They are represented as (A) in Figure 4) are new (green) for the FindBugs system and were never updated (blue). We can reach this conclusion because the molecules in the sectors related to those dependencies are presented on the edge (outer guideline) of the visualization and there are no red molecules in it; (ii) the

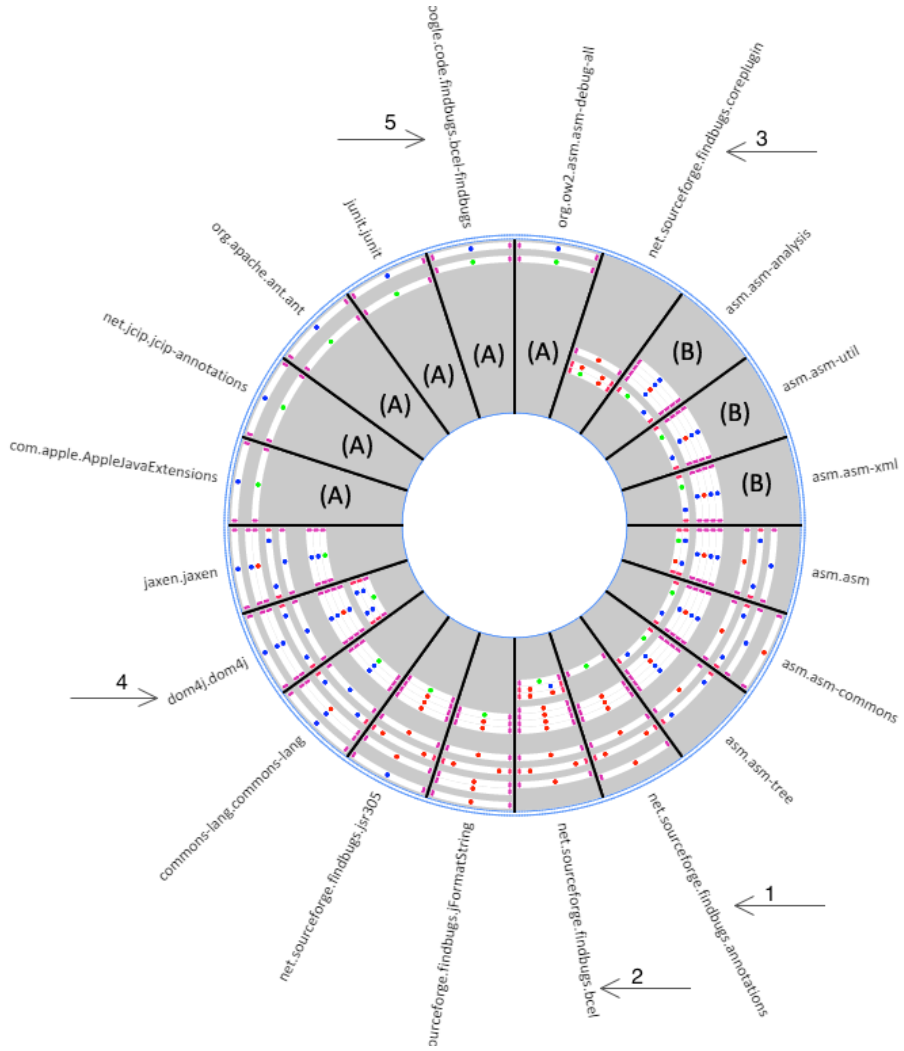

Figure 4. The dependency usage of FindBugs project.

"net.sourceforge.findbugs.annotations" (Figure 4-arrow 1) 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 behavior represents an indication of high coupling between those systems; (iii) another important behavior 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. However, there were new versions released during this period as we can see in the sector "net.sourceforge.findbugs.bcel" (Figure 4-arrow 2); (iv) the "net.sourceforge.findbugs.annotations" dependency was used after this window again. This behavior represents that the features provided by this dependency might be replaced by another library. One possible candidate is the "net.sourceforge.findbugs.coreplugin" (Figure 4-arrow 3), because it is only used when the "net.sourceforge.findbugs.annotations" dependency is not; (v) we can notice that the regularity of changes in the "dom4j.dom4j" (Figure 4-arrow 4) dependency is minimal. It is presented in all versions of the FindBugs system and its version was only updated once.

Understand what important structural dependency events have occurred

The EVOWAVE visualization, as in Figure 4, 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 4, were removed from the project at the same time. This may imply that they were removed or used for some common feature that is using a different library. Those previously defined as new dependencies, represented as (B) in the Figure 4, are an important structural dependency event. The dropped library "net.sourceforge.findbugs.bcel" (Figure 4-arrow 2) is an important structural event because it was highly used from the beginning and was always updated. This behavior 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" (Figure 4-arrow 5) was introduced in the system at the same time "net.sourceforge.findbugs.bcel" (Figure 4-arrow 2) was removed. Based on the names of the libraries, it is possible to observe that this new library might be replacing the old one or maybe the library was simply renamed.

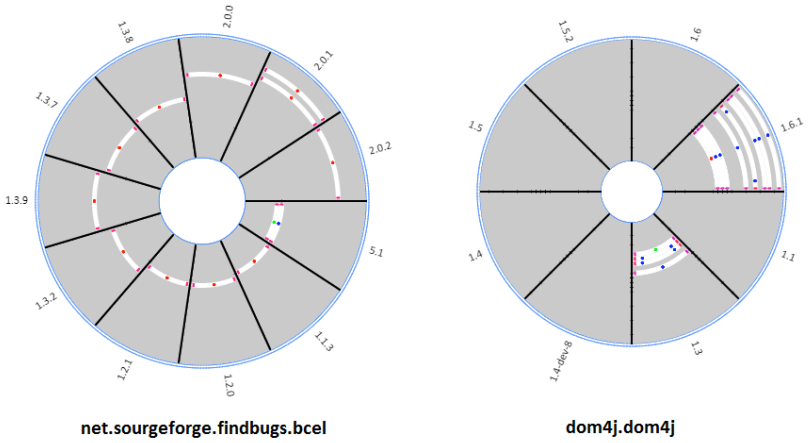

Figure 5. Two EVOWAVE visualizations for two dependencies, with the usages by the FindBugs system.

To analyze the "net.sourceforge.findbugs.bcel" dependency in more detail, the software engineer can right click on the sector that represents it. Figure 5 displays the EVOWAVE visualization for this dependency on the left (Sectors now represents FindBug's versions). A common behavior 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 (molecule closer to the outer guideline) 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 (Figure 5-right) seems to be highly stable for this system. Nevertheless, the change from version "1.3" to version "1.6.1" was very expressive, since 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

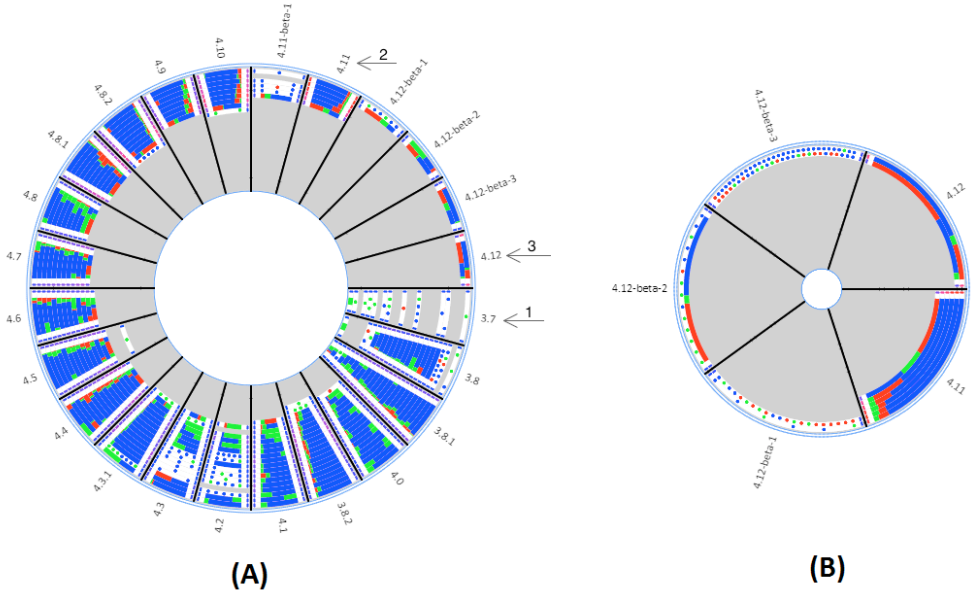

Figure 6. Two EVOWAVE visualizations of the "junit.junit" dependency usage.

The latest version of FindBugs uses the "junit.junit" version "4.11". Figure 6-A represents the use of all versions of this dependency registered in the Maven repository. We can see that all versions, except for "3.7" (Figure 6-A-arrow 1), have systems that maintain the same version of this dependency, since, in the last window, there are blue molecules in the sector of those versions. However, versions "4.11" (Figure 6-A-arrow 2) and "4.12" (Figure 6-A-arrow 3) 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 frontier. Since the goal is to look for current attractive versions, the visualization was filtered to show only the versions between those versions. Figure 6-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 ("4.11") still has a huge amount of projects starting to use, updating, and maintaining 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. Another interesting fact is that three beta versions of version "4.12" were released almost at the same time as version "4.12" and had lower attractiveness (Figure 6-B). This 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.

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 discussed before,

this version has been attractive in the last six months. In less than 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. Software engineers of this system should consider updating to the version "4.12" of the "junit.junit" dependency and should not consider the beta versions.

3.3. Conclusion

This study was performed using the same data and tasks proposed by Kula et al. [Kula et al. 2014]. The authors answered those questions using two metaphors they proposed. To answer those questions, software engineers had to learn the concepts of both metaphors proposed, and have some experience with them. The rationale for our study was to validate the possibility of performing those tasks with only one metaphor. 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. Related Work

In our literature review, we found two stands out related work for being similar to this work. They use a similar layout, but their proposal focus on different tasks for a unique domain.

In [D'Ambros et al. 2009], the authors proposed a visualization-based approach, named Evolution Radar, which integrates logical coupling information at different levels of abstraction. It shows the dependencies between a module in focus and all the other modules of a system. 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. In [Kula et al. 2014], the authors proposed to visualize how the dependency relationship in a system and its dependencies evolves from two perspectives. 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. 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.

5. Final Remarks

Large projects depend on several third-party libraries. Both projects and libraries need to evolve, and to understand how the dependencies evolve is a key activity. In this work, we used the EVOWAVE, a multiple domain software evolution visualization metaphor, to perform an exploratory study on the analysis of FindBug's library dependency information. This study showed we were able to evaluate the EVOWAVE in another context (i.e. library dependency). We used the same study procedures as in [Kula et al. 2014]. In addition to the fact it is one more context, we can highlight other outcomes of our tool as it uses a single metaphor and is a fresh open source tool, web-based, and highly configurable.

The exploratory study conducted helped us to see the outcomes and shortcomings of EVOWAVE. As outcomes, it indicated the EVOWAVE usefulness on

the library dependency analysis of a large software system. As shortcomings, it indicated the need of tool improvement. More mechanisms of interaction are necessary to speedup the analysis, specially by end users. The exploratory study itself has limitations, since it is conducted by the authors' perspective. As future work, we plan to conduct other experimental studies in order to evaluate EVOWAVE in the context of library dependency, however considering end users.

References

[Basili and Rombach 1988] Basili, V. R. and Rombach, H. D. (1988). The tame project: Towards improvement-oriented software environments. IEEE Trans. Softw. Eng., 14(6):758–773.

[Bergel et al. 2011] Bergel, A., Bañados, F., Robbes, R., and Binder, W. (2011). Execution profiling blueprints. Software: Practice and Experience, pages n/a–n/a.

[Carneiro et al. 2008] Carneiro, G., Magnavita, R., and Mendonça, M. (2008). Combining software visualization paradigms to support software comprehension activities. In Proceedings of the 4th ACM Symposium on Software Visualization, SoftVis '08, pages 201–202, New York, NY, USA. ACM.

[Carneiro et al. 2010] Carneiro, G., Silva, M., Mara, L., Figueiredo, E., Sant'Anna, C., Garcia, A., and Mendonca, M. (2010). Identifying code smells with multiple concern views. In Software Engineering (SBES), 2010 Brazilian Symposium on, pages 128–137.

[Cornelissen et al. 2007] Cornelissen, B., Holten, D., Zaidman, A., Moonen, L., van Wijk, J., and van Deursen, A. (2007). Understanding execution traces using massive sequence and circular bundle views. In Program Comprehension, 2007. ICPC '07. 15th IEEE International Conference on, pages 49–58.

[D'Ambros et al. 2009] D'Ambros, M., Lanza, M., and Lungu, M. (2009). Visualizing cochange information with the evolution radar. IEEE Trans. Softw. Eng., 35(5):720–735.

[Diehl 2007] Diehl, S. (2007). Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[Kuhn et al. 2010] Kuhn, A., Erni, D., Loretan, P., and Nierstrasz, O. (2010). Software cartography: thematic software visualization with consistent layout. J. Softw. Maint. Evol., 22(3):191–210.

[Kula et al. 2014] Kula, R., De Roover, C., German, D., Ishio, T., and Inoue, K. (2014). Visualizing the evolution of systems and their library dependencies. In Software Visualization (VISSOFT), 2014 Second IEEE Working Conference on, pages 127–136.

[Lanza 1999] Lanza, M. (1999). Combining metrics and graphs for object oriented reverse engineering.

[Lungu 2008] Lungu, M. (2008). Towards reverse engineering software ecosystems. In Software Maintenance, 2008. ICSM 2008. IEEE International Conference on, pages 428 –431.

[Magnavita 2016] Magnavita, R. (2016). EVOWAVE: A Multiple Domain Metaphor for Software Evolution Visualization. Dissertation, Universidade Federal da Bahia.

[Magnavita et al. 2015]Magnavita, R., Novais, R., and Mendonça, M. (2015).Using evowave to analyze software evolution. In Proceedings of the 17th International Conference on Enterprise Information Systems, pages 126–136.

[Mazza 2009] Mazza, R. (2009). Introduction to Information Visualization. Springer Publishing Company, Incorporated, 1 edition.

[Novais et al. 2011] Novais, R., Lima, C., de F Carneiro, G., Paulo, R., and Mendonça, M. (2011). 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, pages 1–4.

[Novais et al. 2012] Novais, R., Nunes, C., Lima, C., Cirilo, E., Dantas, F., Garcia, A., and Mendonca, M. (2012). On the proactive and interactive visualization for feature evolution comprehension: An industrial investigation. In Software Engineering (ICSE), 2012 34th International Conference on, pages 1044–1053.

[Novais et al. 2013] Novais, R. L., Torres, A., Mendes, T. S., Mendonça, M., and Zazworka, N. (2013). Software evolution visualization: A systematic mapping study. Inf. Softw. Technol., 55(11):1860–1883.

[Olivera 2015] Olivera, F.R. (2015). Mvnrepository. Retrieved from http://mvnrepository.com/.

[ProcessingJS 2015] ProcessingJS (2015). A port of the processing visualization language. Retrieved from http://processingjs.org/.

[Quante 2008] Quante, J. (2008). Using library dependencies for clustering. In 10th Workshop Software Reengineering, 5-7 May 2008, Bad Honnef, pages 171–175.

[Staples and Bieman 1999] Staples, M. L. and Bieman, J. M. (1999). 3-D visualization of software structure. volume 49 of Advances in Computers, pages 95 – 141. Elsevier.

[Voinea and Telea 2006] Voinea, L. and Telea, A. (2006). Multiscale and multivariate visualizations of software evolution. In Proceedings of the 2006 ACM symposium on Software visualization, SoftVis '06, pages 115–124, New York, NY, USA. ACM.

[Wettel and Lanza 2008] Wettel, R. and Lanza, M. (2008). Codecity: 3d visualization of large-scale software. In Companion of the 30th international conference on Software engineering, ICSE Companion '08, pages 921–9.