# The Medium of Visualization for Software Comprehension

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

**Leonel Merino**

von Chile

Leiter der Arbeit:
Prof. Dr. O. Nierstrasz
Institut für Informatik
Universität Bern

# The Medium of Visualization for Software Comprehension

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

**Leonel Merino**

von Chile

Leiter der Arbeit:
Prof. Dr. O. Nierstrasz
Institut für Informatik
Universität Bern

Von der Philosophisch-naturwissenschaftlichen Fakultät angenommen.

Bern, 29.06.2018.

Der Dekan:
Prof. Dr. Gilberto Colangelo

This dissertation can be downloaded from scg.unibe.ch.



Attribution–ShareAlike

*To Cuqui, my wife.*

*To my children: Magdalena, Agustín, and Josefina.*

# Acknowledgments

I am immensely grateful to Oscar Nierstrasz who gave me the opportunity to join the Software Composition Group. Thanks for giving me the time and freedom to grow as a researcher, and pursue my interests. Thanks Oscar!

I want to express my immense gratitude to Iris Keller who was so friendly from the first moment, and helped me and my family with all the administrative work. Thanks Iris!

I am grateful to Nevena, Andrea, Andrei, Boris, Claudio, Haidar, Jan, Pascal, Manuel, and Yuriy, for being always friendly, and for their support and advice. Thank you!

I want to thank Mircea for motivating me to do research in software visualization, and I thank Mohammad for being critical to the value of software visualization. Thanks Mircea and Mohammad!

I am thankful to Daniel Keim and the *DBVIS* group in Konstanz for hosting me and my family during a lovely winter in 2017. We have great memories from the Christmas party at Daniel's home. Thank you Daniel!

I am grateful to Dominik, Silas, and Mario, who gave me the honor to mentor their bachelor theses. I had a great time discussing with you. Thanks!

I want to thank Alexandre Bergel for being a great host during my stay in the Chilean summer of 2018. Your enthusiasm and positiveness have always been encouraging. I enjoyed our discussions during the lunch in Santiago. Thanks Alex!

I am grateful to Fabian Beck for accepting to be the external examiner in my PhD committee. I would also like to thank Paolo Favaro for chairing the examination.

I want to thank Bettina Choffat for helping me to organize my PhD defense.

I am thankful to Pooja, Nitish, and Reza, for your interest in my work. Thanks!

Last but not least. I want to thank my wife Cuqui, for being always with me. I love to share my life with you, and with Magdalena, Agustín, and Josefina, who are a permanent source of joy. I thank to God for all of you.

# Abstract

Although abundant studies have shown how visualization can help software developers to understand software systems, visualization is still not a common practice since developers *(i)* have little support to find a proper visualization for their needs, and once they find a suitable visualization tool, they *(ii)* are unsure of its effectiveness. We aim to offer support for identifying proper visualizations, and to increase the effectiveness of visualization techniques.

In this dissertation, we characterize proposed software visualizations. To fill the gap between proposed visualizations and their practical application, we encapsulate such characteristics in an ontology, and propose a meta-visualization approach to find suitable visualizations. Amongst others characteristics of software visualizations, we identify that the *medium* used to display them can be a means to increase the effectiveness of visualization techniques for particular comprehension tasks. We implement visualization prototypes and validate our thesis via experiments.

We found that even though developers using a physical 3D model medium required the least time to deal with tasks that involve identifying outliers, they perceived the least difficulty when visualizing systems based on the standard computer screen medium. Moreover, developers using immersive virtual reality obtained the highest recollection.

We conclude that the effectiveness of software visualizations that use the city metaphor to support comprehension tasks can be increased when city visualizations are rendered in an appropriate medium. Furthermore, that visualization of software visualizations can be a suitable means for exploring their multiple characteristics that can be properly encapsulated in an ontology.

# Contents

# 1 Introduction

By automating time-consuming, laborious, and repetitive tasks software systems support humans to be more productive. However, the benefit that a software system brings depends on its quality. While a high quality system can boost the productivity of users, low quality systems can even obstruct their work. There are six main characteristics of the quality of a software system[1]. A high quality system not only must include the required *(i) functionality*, but it also has to be *(ii) reliable*. That is, the system has to be up and running under specified circumstances (*e.g.,* failures) for a defined period of time. The system must also be designed to *(iii) ease its use*, to encourage adoption. The system must *(iv) efficiently* use the available resources (*e.g.,* disk space, memory, network). The system must encourage *(v) portability*; that is, being able to adapt to changes in the operating environment and user requirements. Once the system is being used in a productive environment, it must facilitate *(vi) maintenance* tasks, which can involve fixing an identified fault or adding new functionality.

Nonetheless, the quality of software systems varies, and some systems that exhibit low quality are hard to maintain. One reason is that a prerequisite to undertake a maintenance task is that developers need to *understand*

[1] `http://www.iso.org/iso/en/ISOOnline.frontpage`

the software system. When understanding artifacts of a software system, developers need to analyze various data related to the system. A prominent characteristic of such data is their source, which usually restricts the number of available tools. Usually developers start understanding aspects of a system such as the system's structure by analyzing the source code. Developers also might need to analyze the log of the system's execution to understand the system's behavior at run time. The version control system contains valuable data of the evolution of a system, which can help developers, for instance, to identify changes that need to be reverted to remove a bug of the system. Typically, these data are structured in a format intended to be read by machines instead of humans, which also hinders the ability of developers to analyze the data. The complexity of the data set, which depends on its source and format (*e.g.,* number of lines of code and dependencies amongst system components in the source code), makes program understanding a complex task. All in all, analyzing software data requires great effort from developers.

Usually, programming environments used by developers include tools to support the common basic tasks that arise during program understanding such as browsing, editing, and debugging. These tools are mostly based on text, and programmers spend a lot of time reading data (*e.g.,* reading source code). We believe that existing text based tools limit the abilities of developers to analyze software data. In particular, we believe that visual representations of attributes of data can bring great support to developers who can use them to analyze the many attributes and relationships within a data set. Data visualization is the field that focus on proposing tools and techniques to augment the human capabilities to analyze the many attributes and relationships of a data set through a visual representation. The intersection of the software engineering and the data visualization fields gathers the research on *software visualization*.

## 1.1 Software Visualization

*"Software visualization is the use of interactive computer graphics, typography, graphic design, animation, and cinematography to enhance the interface between the software engineer or the computer science student and their programs"* [PBS93].

Many software visualization tools have been proposed to tackle multiple specific tasks that arise during development. To name a few: *Code-*

*Crawler* [LD03] helps developers analyze software metrics during reverse engineering, *Jive* [Rei03] supports understanding the runtime of a program, *SHriMP* [LMSW03] offers views of the structure of a program, *CVSscan* [VTvW05] enables developers to get insight into the changes of a system from the Control Version System, *Softwarenaut* [LLG06] supports exploring hierarchical decompositions of a system, *CodeCity* [WL07a] eases comprehension by mapping a system's aspects using a city metaphor, *Rigi* [KMM07] helps explore dependency relations at the class and package level of Java systems, *Zinsight* [DPH10] enables user exploration, analysis and understanding of traces containing many events, *Stench Blossom* [MHB10] promotes smell detection by an interactive ambient visualizations, *Code Bubbles* [Rei14] supports developers in debugging tasks, *CodeSurveyor* [HMA15] supports code comprehension in large codebases by allowing developers to view large-scale software at all levels of abstraction using a map metaphor, *TypeV* [FSWH16] helps developers to analyze the evolution of a software system by visualizing abstract syntax trees, *vizSlice* [AJdS+16] supports understanding of large scale software slices, and *SoL Mantra* [TKII17] helps to identify library update opportunities. While some of the proposed visualization tools do not explicitly target program comprehension tasks, we consider that all of them support in a degree the understanding of software systems, and therefore, can be of help during maintenance.

We list, inspired by a previous taxonomy [MMC02], the main characteristics of the proposed software visualization to identify their strengths and limitations. The requirements for a software visualization tool are characterized by the particular needs of developers. The needs involve a *task* that belongs to an *audience* who have particular *data*. Similarly, the proposed visualization approach involves a visualization *technique* that is displayed using a *medium*, which is packaged in a *tool*.

**Task**

There are many particular software engineering tasks in which visualization has been proposed to support developers. These tasks can be classified using various criteria. For instance, the tasks can be classified into one of the following three categories: *structure*, *behavior*, and *evolution* [Die07]. Developers, for instance, can visualize the system's *(i)* structure to analyze the dependencies obtained from the source code, *(ii)* behavior based on execution traces in a

running system, and *(iii)* evolution based on the meta data gathered from a version control system.

The tasks can also be categorized into one of multiple problem domains [LM10]: *(i) changes* such as debugging, refactoring, and testing; *(ii) elements* such as performance, concurrency and intent and implication; and *(iii) element relationships* such as dependencies, architecture, and control flow.

Sometimes tasks are also classified by identifying the cognitive process of moving from questions to answers [SMDV06]. This classification includes: *(i) finding* initial software entities that might lead developers to formulate a concrete question, *(ii) building* on those points by identifying relationships between entities, *(iii) understanding* a group of entities and relationships, and *(iv) questioning* how various groups relate each other.

Also tasks can be classified by the subject [FM10] of the need: *(i) people* (*e.g.,*, who is working on what), *(ii) code* (*e.g.,*, changes to the code), *(iii) progress* (*e.g.,*, work item progress), *(iv) build* (*e.g.,*, broken builds), *(v) test* (*e.g.,*, test case analysis), *(vi) web* (*e.g.,*, web related concerns)

The analysis of the tasks using these various classifications can help researchers in the software visualization field to identify tasks with little visualization support. The analysis also can help developers (who are willing to adopt visualization) to find suitable tools.

**Audience**

The expected user of a software visualization tool plays a specific role in the software life-cycle. We observe that a software visualization that is effective to support the concerns that arise with a certain audience can be ineffective to the concerns of another one. Therefore, we believe that the role of the user has to be taken into account when looking for a suitable visualization for a particular concern. Although most software visualization tools target the programmer's audience, there are some tools that target other audiences. For instance, tools in which the expected user plays a role such as project manager or architect.

**Data**

A software visualization provides developers a tangible representation of *software data*. We call software data the various types of

data that relate to a software system. For instance, source code, execution logs, and meta-data from the version control system can be visualized to analyze the structure, behavior, and evolution respectively. Although all these software data can relate to the same system, they can greatly differ in their characteristics (*e.g.,* format, source). The choice of which visualization technique to use partially depends on the characteristics of data. For instance, some visualization techniques can represent only certain data types (*e.g.,* qualitative, quantitative, nominal, ordinal, discrete, continuous, hierarchical).

**Technique**

The technique used in a software visualization tool defines the graphical attributes that are chosen to represent the properties of software data. The many visualization techniques that have been proposed are sometimes referenced using multiple names, which makes tracking their adoption difficult. Sometimes a proposed technique is the result of combining already known techniques. Some taxonomies have been proposed to classify visualization techniques based on various criteria. However, due the lack of a unified catalog of visualization techniques, reflecting on the relationship of visualization techniques with other characteristics (*e.g.,* tasks, data) is still hard. A few taxonomies have been proposed to classify visualization techniques [Kei02]. In our experience these taxonomies provide a good abstraction that eases the analysis of such relationships.

**Medium**

The medium corresponds to the means used to display the visualization. Commonly, software visualizations are displayed using the standard computer display medium [MGN16a]. The popularity of the standard computer display can be a consequence of its *(i)* high availability in most computers (where most programming is done), and *(ii)* high quality graphics available in the current technology. However, various other media are available for displaying visualizations such as wall-displays, multi-touch tables, immersive virtual reality, and physical 3D models. Only a few software visualization approaches have included a medium different than the computer screen. Though in the past researchers had limited access to these rather extravagant media due their high cost, today many of them have become accessible and are already available in the ecosystem of developers.

Indeed, several studies have shown qualitative evidence of the impact of the medium in the information visualization field. One study [Dat02] found that 3D visualizations in immersive virtual reality are “intuitive”, and “easy to use”. Another study [RBLN04] found that they are “useful”. A study [BM07] indicates that key characteristics of the success of immersive virtual reality in the gaming industry are its ability to “engage” and “entertain”. Also, a more recent investigation [KJ18] found that users “connect”, and obtain a better “overview” in immersive virtual reality than using 3D visualizations displayed on the standard computer screen. Some other studies also have found statistically significant differences through qualitative evaluations. For instance, one study [SDP+09] found that users obtain a better experience using 3D visualizations in immersive virtual reality, which users find “intuitive” and “natural”, compared to 3D visualizations displayed on the standard computer screen. Another investigation [WFRFN] showed that users who visualize data in immersive virtual reality required less “effort” and “navigation” to find information, and “engage” better and perceive themselves to be more “accurate”, than when they used 3D visualizations on the standard computer screen. We conjecture that some of the benefits reported of using immersive virtual reality in *information visualizations* can be transfered to *software visualizations*.

We hypothesize that the medium, defined as “a means or channel of communication or expression”[2], plays a role in the the effectiveness of software visualizations. In consequence, we propose that software visualizations take advantage of such media to increase their effectiveness. In a communication the medium is the means used to send a *message* from a *sender* to a *receiver*. Indeed, amongst the main usages of visualizations are proposed: *discovery* and *communication* [Mun14]. We observe that when using visualizations for

*(1)* discovering, the sender is the data, the message is the data’s insight (that developers are expected to decode), and the receiver is the developer (who is using the visualization to retrieve the message); and

[2] "medium, n. and adj." OED Online. Oxford University Press, June 2017. Web. 2 August 2017.

*(2)* communicating already discovered insights (the message), the sender is the developer herself, and the receiver is her audience (*e.g.,* a development team).

We believe that the *medium* used in a visualization to discover and communicate insights of a data set plays an important role in the effectiveness of software visualization.

**Tooling**

Many software visualization prototypes have been proposed to alleviate various software development concerns [MGN16a, Die07]. These prototypes are heterogeneous by nature. They are written in multiple languages and use various libraries. However, only some of them are publicly available. We ask about the evidence presented to support the claimed effectiveness of such tools. We ask how *effectiveness* is defined in such evaluations and whether the strategies and methods used in them are appropriate.

## 1.2 Problem Statement

The main drawbacks to adopt a proposed software visualization approach are the lack of *(i)* means to find a suitable one for a particular software engineering task, and *(ii)* evidence of their effectiveness.

A developer who is willing to adopt a visualization approach to deal with a particular software engineering task might struggle to find a suitable software visualization. There are several variables that must be taken into account to find a suitable visualization (*e.g.,* task, data, audience, medium), however, this is obstructed by the lack of organization amongst proposed software visualizations. Moreover, once a suitable visualization is found, developers are usually unsure of the effectiveness of visualizations.

Effectiveness is defined as having "the power of acting upon the thing designated".[3] Consequently, to assess the effectiveness of software visualizations we must know in advance their designated requirements. However, identifying the requirements of software visualizations is difficult due the lack of organization amongst the proposed visualization approaches. For those approaches in which we identify a list of requirements, we investigate how thorough are the evaluations that bring evidence of effectiveness.

[3] "effective, adj. and n." OED Online. Oxford University Press, June 2017. Web. 27 October 2017.

We believe that the main strength of software visualizations is that they enable communication. However, the medium designated to transport the message between the sender and the receiver is usually not considered relevant and limited to a single one (*i.e.,* the standard computer screen). Moreover, we ask whether conducted evaluations have assessed this aspect, which could be another reason that hinders software visualization adoption. We believe that a sound evaluation of software visualization tools across multiple media can help us to identify specific tasks, audiences, and data, in which a given visualization technique is particularly effective.

## 1.3 Our Approach

We present a taxonomy in which we classified proposed software visualizations. The taxonomy not only allows practitioners to find suitable visualizations for particular concerns, but also allows researchers in the software visualization field to identify problems domains that require more attention. We observe that many software visualizations have been proposed to deal with multiple development concerns. To ensure their effectiveness, we expand our investigation, and analyze the software visualization evaluations.

We found several visualization tools that have proven to be effective in evaluations, however, whether these tools are available, and can be put into action is unclear. To deal with this problem, we curated a catalog of 70 actionable software visualization tools. We report on their execution environment, last date of maintenance, level of maturity, employed visualization technique, and the medium used to display them. We observe that most software visualizations are displayed on the standard computer screen, and conjecture that the standard computer screen might not be an appropriate medium to display some visualization techniques.

We argue that the effectiveness to support particular tasks of software visualizations can be increased when displayed on a more appropriate medium. We focus on the 3D city visualization technique (widely used), that we display with immersive virtual reality, and with printed physical 3D models as media. We conduct a controlled experiment and report the results. We found that physical models boost completion time of tasks that involve identifying outliers, however, such tasks are considered the least difficult when the visualization is displayed on the standard computer

screen. Moreover, we observe that immersive virtual reality excels at promoting recollection.

### 1.3.1 Thesis statement

We formally state our thesis as follows:

*To increase the effectiveness of software visualization tools, we need to consider the impact of the medium in user performance and experience.*

## 1.4 Contributions

There are several contributions from the conducted research. The technical contributions include the implementation of prototype tools used for testing hypotheses as well as data sets extracted in literature reviews. The conceptual contributions involve a taxonomy for characterizing software visualization approaches, and a discussion of the impact of the medium in the effectiveness of visualization. The main contributions of this dissertation are:

### 1.4.1 A taxonomy of proposed software visualization tools, and a discussion of the need of the explicit inclusion of the medium as a key attribute that promotes the communication in software visualizations

We collected relevant studies in the software visualization field that we classified into dimensions of a proposed taxonomy. Amongst other attributes, we characterized the tasks for which visualizations have been proposed, and the media used to display such software visualizations. When grouping the tasks into problem domains, we found a disconnect between the problem domains on which visualization have focused and the domains that get the most attention from developers.

Although the medium has been considered as a main attribute since foundational software visualization taxonomies [RC93, PBS93, MMC02], we observe that the medium has not been a main concern among most proposed visualizations. In consequence, we observe that most software visualization approaches use the standard computer screen, and only a

few have included another medium. We believe that there are certain tasks in which the choice of the medium used to display a software visualization can increase the effectiveness of the tool. In consequence, we think that researchers who propose a visualization tool to deal with software development concerns must explicitly support the medium of choice.

The results of this investigation were published in a full peer-reviewed paper in an international conference [MGN16a], and then extended in a journal [MGN17].

### 1.4.2 A discussion of meta-visualization as a means for identifying suitable visualization tools

We investigated means for filling the gap between proposed software visualizations and their practical application. We focused on live visualization example objects that are annotated with a set of questions that they can help to investigate. We instrumented a prototype application to monitor the use of the examples in an active community, and found that developers sometimes spend great effort finding suitable examples that they can tailor for their specific needs. To deal with such problem, we proposed a meta-visualization approach. It it, a tag-iconic cloud-based representation connects frequent keywords (extracted from questions) to icons that represent visualization examples. Through usage scenarios we show preliminary results of its usability.

The results of this study were published in a short peer-reviewed paper in an international conference [MGN+16b].

### 1.4.3 A discussion of the software visualization evaluation

We also investigated the characteristics of the evaluations used in proposed software visualization approaches. We reviewed the literature to collect attributes of such evaluations. Amongst others, we extracted evaluation strategies, data collection methods, and statistical tests for analyzing results in experiments. We found that even though a fair number of studies used experiments to evaluate a proposed visualization tool, most of the proposed software visualizations do not explicitly include an evaluation. We also found that only a few studies conducted surveys to investigate the needs of practitioners. We observe that due the nature of software visualization tools, they are rather different than usual software engineering tools. In

consequence, experiments focusing mostly on completion *time* and *correctness* can offer a limited assessment of a visualization tool. We argue that an effective software visualization should not only boost time and correctness but also other variables such as recollection, usability, engagement, and other impressions that might promote the communicate ability of a software visualization. We conclude by providing researchers in the field with guidelines to design and conduct the evaluation of proposed software visualization tools.

The results of this study were published in a journal [MGAN].

### 1.4.4 A discussion of the architectural design choices and lessons learned from implementing the city metaphor in virtual reality and observing its use

We hypothesized that immersive virtual reality can promote user engagement, and therefore benefit the understanding of software systems by making users willing to spend more time on it. We developed a prototype tool to test this hypothesis that uses immersive virtual reality to display a visualization based on the city metaphor. We conducted a formative experiment with six participants to identify potential strengths of the medium. We analyzed engagement in terms of navigation, emotions, and time perception. We found that this medium boosts the concentration of developers who are able to focus on particular tasks. We investigated whether this medium could boost recollection, and so benefit the understanding of a software system.

The results of this study were published in a tool demo peer-reviewed paper in an international conference [MGAN17].

### 1.4.5 An experiment to evaluate the impact of the medium in the effectiveness of 3D software visualizations

We conducted an experiment to evaluate the impact of the medium in the effectiveness of a software visualization technique. In the experiment, we compared the visualization technique displayed on three media: *(i)* standard computer screens (SCS), *(ii)* immersive virtual reality (I3D), and *(iii)* physical 3D models (P3D). We defined a set of comprehension tasks, and selected a popular software visualization technique which is based

in the city metaphor. We conducted the experiment with 27 participants who were split into three groups. Each of them used the visualization in only one medium. We found that even though developers using P3D required the least time to identify outliers, they perceived the least difficulty when visualizing systems based on SCS. Moreover, developers using I3D obtained the highest recollection.

The results of this study were published in the proceeding of a conference [MFB+17].

### 1.4.6 The artifacts of our research are publicly available

Although reproducibility is a key component of science, we observe there are many publications in which involved artifacts are not publicly available. Instead, we decided to ensure the reproducibility of the results of our research by making the involved artifacts publicly available:

- CityVR source code[4]
- Replication package of the experiment[5]

## 1.5 Outline

This dissertation is organized as follows:

**Chapter 2** presents an overview of the related work in software visualization. The chapter introduces a taxonomy in which we classify proposed software visualizations, and discusses the gaps found in terms of the tasks on which visualizations were focused, and the attributes (*e.g.,* the medium), which we hypothesize might impact the effectiveness of software visualizations.

**Chapter 3** elaborates on the evidence of the effectiveness of proposed software visualization tools via a systematic review of the literature in which we extracted the main concepts of the evaluations used in each case. We discuss quantitative and qualitative analyses of the results, and propose guidelines for designing and conducting evaluations of software visualization tools.

[4] `http://scg.unibe.ch/research/cityvr`
[5] `http://scg.unibe.ch/research/mediavis`

**Chapter 4** discusses our attempts to fill the gap between proposed software visualizations and their practical application. We elaborate on *(i)* a meta-visualization approach that link live visualization examples to keywords extracted from questions that visualizations support, *(ii)* a curated catalog of 70 actionable software visualizations, and *(iii)* an ontology approach to encapsulate the characteristics of software visualizations.

**Chapter 5** introduces a prototype: *CityVR*, a gameful software visualization tool displayed with immersive virtual reality for the understanding of software systems. We discuss lessons learned during its design and implementation, and report on a formative experiment in which we assess user engagement as a means to enhance communication.

**Chapter 6** describes an experiment that evaluates the impact of the medium in the effectiveness of 3D software visualization. We discuss how for certain tasks there is a specific medium in which the effectiveness of the visualization technique is increased. We also outline complementary attributes that need to be included when evaluating the effectiveness of software visualizations.

**Chapter 7** concludes the dissertation and outlines future work.

# 2
# State of the Art

## 2.1 Introduction

Software visualization provides enormous advantages for the development process; to name a few, it supports project managers in communicating insights to their teams [TGG08], it guides testers when exploring code for anomalies [DPKM06], it helps analysts to make sense of multivariate data [MLN15], and it aids new developers in open software communities [PJ09]. However, visualization is not yet commonly used by developers. More than a decade ago researchers wondered *why is software visualization not widely used?* [Fal02]. They observed that one of the reasons is that efforts in software visualization are out of touch with the needs of developers [Rei05]. Several attempts have tried to fill in the gap and encourage developers to adopt visualization. For instance, Maletic *et al.* [MMC02] proposed a taxonomy of software visualization to support various tasks during software development; Schots *et al.* [SW14] extended this taxonomy by adding the resource requirements of visualizations, and providing evidence of their utility; Storey *et al.* [SvG05] proposed a framework to assess visualization tools; Kienle *et al.* [KM07] performed a literature survey to identify quality attributes and functional requirements for software visual-

ization tools; Padda *et al.* [PSM07] proposed some visualization patterns to guide users in understanding the capabilities of a given visualization technique; Sensalire *et al.* [SOT08a] classified the features that users require in software visualization tools; and Merino *et al.* [MGN+16b] proposed meta-visualization of keywords that represent development concerns connected to visualization examples for helping developers to find suitable visualizations. However, the lack of organization among visualization approaches is still an important barrier to finding and using them in practice [SW14]. In fact, developers are still unaware of existing visualization techniques to adopt for their particular needs. A few studies have tried to address this issue by investigating to which software engineering tasks particular visualization techniques have been applied [GHM05, PAM14, SLB14]. Nevertheless, we believe these studies are still too coarse-grained to match a suitable visualization to their concrete needs.

When developers perform a particular programming task they ask some questions such as *"what code is related to a change?"* or *"where is this method called?"* Several studies have investigated such questions and classified them into groups [SMDV06, KDV07, FM10]. Indeed, such questions reflect developer needs, and we believe that mapping them to existing types of visualization can help developers to adopt visualization in their daily work. In particular, we would like to answer the following research questions:

*RQ.1)* What are the characteristics of visualization techniques that support developer needs?

*RQ.2)* How well are developer needs supported by visualization?

We believe answering these questions, *(1)* helps practitioners to find suitable visualizations for their specific needs, and *(2)* assists researchers in the field to identify needs with little visualization support.

In particular, we review 86 design studies from which we extracted task, need, audience, data source, representation, medium and tool. We characterize them according to the subject, process and problem domain that we discuss in relation to the proposed visualization techniques. We found that one third of the studies combined various visualization techniques, but most of them belong to one of the following three types: *(1)* techniques that use *geometric transformations* to explore structure and distribution *e.g.,* Parallel Coordinates, *(2) stacked* techniques that are tailored to present data partitioned in a hierarchical fashion *e.g.,* Treemap, and *(3) pixel-oriented*

techniques that are suitable for displaying large amounts of data *e.g.,* Table lens. We found that software visualizations address problem domains that receive diverse levels of attention from developers. That is, many visualizations have been proposed to tackle problems in domains that are highly important for developers such as history and debugging, but also in domains that are reported less frequently among developers such as dependencies and concurrency. In contrast, there is little support for needs in contract and policy domains, which are fairly important for developers.

The remainder of the chapter is structured as follows: Section 2.2 describes the methodology that we followed to collect relevant literature and select design studies proposed in the software visualization field; Section 2.3 presents our results by classifying them based on their *task*, *need*, *audience*, *data source*, *representation*, *tool*, and *medium* [MMC02]; Section 2.4 discusses our research questions and threats to validity of our findings, and Section 2.5 concludes.

## 2.2 Methodology

We applied the Systematic Literature Review (SLR) approach, a rigorous and auditable research methodology for *Evidence-Based Software Engineering* (EBSE). The method offers a means for evaluating and interpreting relevant research to a topic of interest. We followed Keele's comprehensive guidelines [Kee07], which make it less likely that the results of the literature survey will be biased.

### 2.2.1 Data sources and search strategy

We sought papers that are relevant to the aim of our investigation, *i.e.,* that propose a visualization technique useful to solve a specific problem in software development. Although such papers are expected to be found across multiple software engineering venues, we decided to collect them from the complete set of papers published by SOFTVIS [SOF16] and VISSOFT [VIS16]. We opted for these two venues because we believe their fifteen editions and hundreds of papers dedicated specially to software visualization offer a sound body of literature reflected in the good (*B*) classification that they obtain in the CORE ranking [COR16] (which considers citation rates, paper submission and acceptance rates among other indicators). Although we observe that publications in better ranked

venues might be of higher quality, we believe that analyzing a collection of studies that have been accepted for publication according to fairly similar criteria will support a more objective comparison, and will provide a suitable baseline for future investigations. Figure 2.1 summarizes the number of papers collected as well as those included in this study.

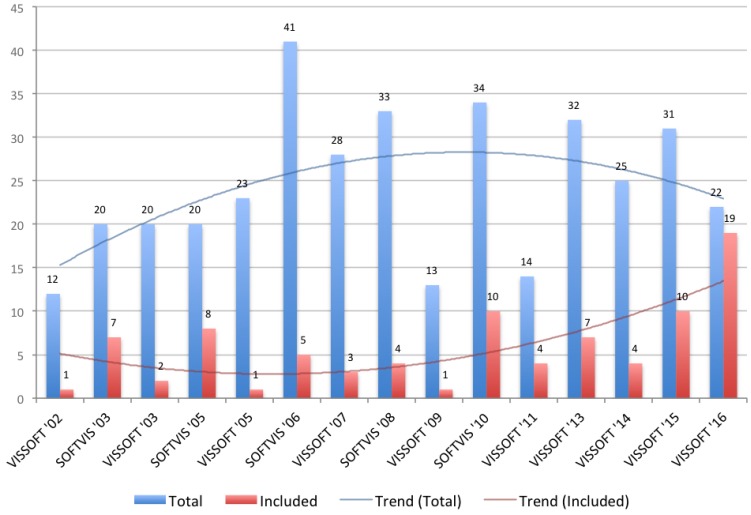



Figure 2.1: The 86 included papers from the collection of 368 papers published in SOFTVIS/VISSOFT venues.

### 2.2.2 Included and excluded studies

We searched for problem-driven studies in which we could identify the role of the user, specific development needs, a proposed visualization technique, and an evaluation demonstrating utility. We excluded short papers of one or two pages (like posters, keynotes and challenges) which due to limited space are unlikely to contain enough detail. We also excluded short papers for which a longer version exists. Of the 295 remaining papers we selected design study papers that describe how a visualization is suitable for tackling a particular problem in software development. We included such papers in our study and excluded papers in the other categories proposed by

Munzner [Mun08] (evaluation, model, system and technique) because we considered them unlikely to provide a visualization to tackle a problem in software development. In the proposed categories a visualization paper can be classified into one of five categories:

*a*) *Evaluations* describe how a visualization is used to deal with tasks in a problem domain. Evaluations are often conducted via user studies in laboratory settings in which participants solve a set of tasks while variables are measured.

*b*) *Design studies* show how existing visualization techniques can be usefully combined to deal with a particular problem domain. Typically, design studies are evaluated through case studies and usage scenarios.

*c*) *Systems* elaborate on the architectural design choices of a proposed visualization tool and the lessons learned from observing its use.

*d*) *Techniques* focus on novel algorithms that improve the effectiveness of visualization. Techniques are often evaluated using benchmarks that measure performance. For instance, edge crossing in graph layout.

*e*) *Models* include *Commentary* papers in which an expert in the field advocate a position and argue to support it; *Formalism* papers present new models, definitions or terminology to describe techniques; and *Taxonomy* papers propose categories that help researchers to analyze the structure of a domain.

We classified the types of papers by first reading the abstract, second the conclusion, and finally, in the cases where we still were not sure of their main contribution, reading the rest of the paper. Although some papers might exhibit characteristics of more than one type, we classified them focusing on their primary contribution. Figure 2.2 shows the outcome of our classification. We identified 86 design study papers and included them in the study. Although more than two thirds of the papers came from VISSOFT, selected papers that we classify as design studies are moderately balanced.

A frequent critique of visualization papers is their lack of evaluation. Indeed, Figure 2.3 shows that papers that take evaluation as their main focus are unusual. The chart also shows an important increment in the number of design study papers in VISSOFT'16, while only a few correspond to model and system papers. Traditionally, the number of papers

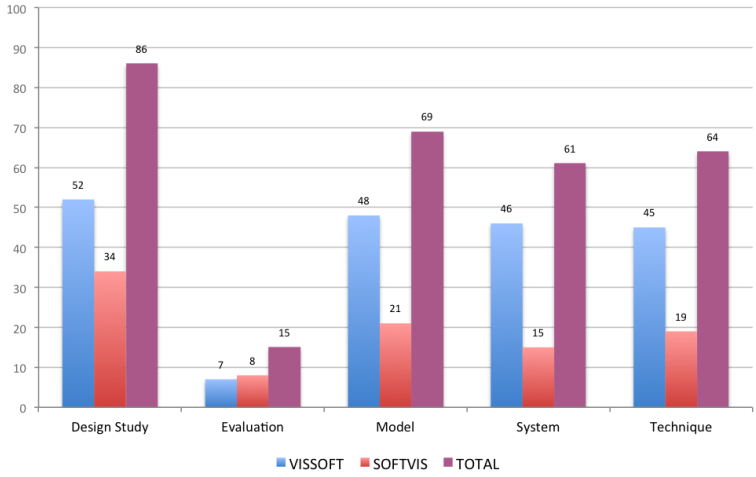



Figure 2.2: Classification of the 295 SOFTVIS/VISSOFT papers by type.

in SOFTVIS editions (2003-2010) was consistently higher than in VISSOFT workshops (2002-2011). The trend of the publications once they merged in the VISSOFT conferences (2013-2016) seems more influenced by SOFTVIS.

Figure 2.4 shows a visualization of the universe of 368 papers published in SOFTVIS/VISSOFT [MSGN16]. In this visualization, rectangles represent papers, their height encodes the number of pages (a 5-page paper is depicted by a square), and the color is used to identify its venue (VISSOFT in blue, and SOFTVIS in red). We used the intensity of the color to represent the publication year, thus the darker the color the newer the paper. Edges connect authors (gray circles) to papers (rectangles). Paper and author nodes are distributed using a force-directed layout. The 86 selected design study papers are distinguished by a black border and a label on top. In the visualization the topology of the community is exposed. A few large groups of collaborators that agglomerate many publications (for which we labeled a main contributor) contrast with the large number of groups that have few of them. We identify two main groups: *(1)* a cohesive one where we labeled the author *"Telea, A."*, and *(2)* another less cohesive but larger one, where we labeled the author *"Lanza, M."*. Although the graph does

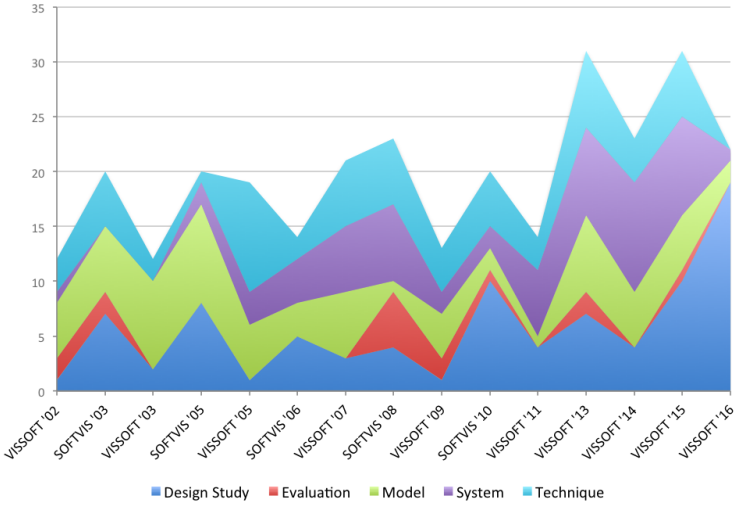



Figure 2.3: Evolution of SOFTVIS/VISSOFT papers by type. From the bottom upwards: Design Study, Evaluation, Model, System, and Technique.

not show the usual topology of a community (due the lack of collaboration), we notice that in VISSOFT'16 both main groups collaborated in a recent publication [SBFB16]. The visualization facilitates the observation that in small groups only one color predominates, thus their publications are not intermingled between SOFTVIS and VISSOFT. Moreover, we observe that the selected papers are scattered among groups of different size, venues and years of publication.

### 2.2.3 Data Extraction

Table 2.1 presents the attributes that we extracted from each paper: *(1)* task; *(2)* need; *(3)* audience; *(4)* data source; *(5)* representation; *(6)* medium; and *(7)* tool.

We scanned the papers and identified recurrent sections that are likely to contain the data we sought. In our experience, attributes such as task, need, audience and data source are frequently described in the evaluation section, while the representation, medium and tool are typically found in another section dedicated to describe the architectural decisions and

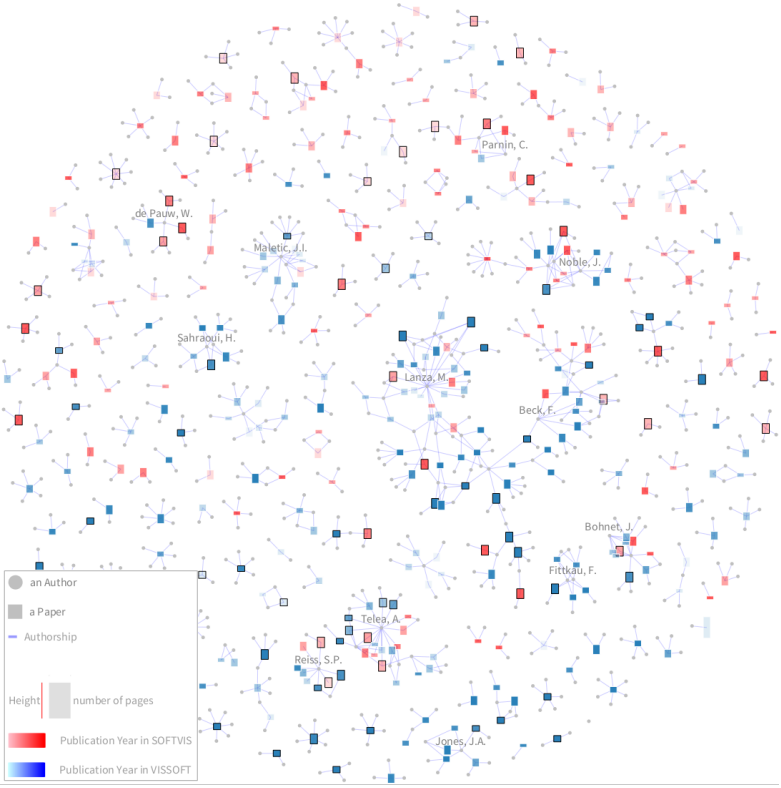



Figure 2.4: Overview of the complete publication record of SOFTVIS/VISSOFT. The 86 selected papers (out of 368) correspond to design studies.

implementation of the prototype. Consequently, we extracted the task by identifying frequent terms used to describe development concerns such as programming, testing, debugging, maintenance, reverse-engineering. For the need we looked for questions that are used to specify what can be answered with the visualization. When there were no explicit questions, we extracted the goal that motivated the need for a proposed visualization. The audience was detected by identifying roles that users play in development such as programmer, engineer, tester. We extracted the data source by identifying the origin of the software artifacts that are visualized, such as

Table 2.1: Data extracted from papers.

| Attribute | Description |
|---|---|
| Task | *why* the visualization is needed (*e.g.,* testing) |
| Need | *which* questions motivated the visualization |
| Audience | *who* will use the visualization (*e.g.,* analyst) |
| Data source | *what* source of data is visualized (*e.g.,* source code) |
| Representation | *what* technique is used to represent the data (*e.g.,* pixel-oriented ) |
| Medium | *where* to render the visualization (*e.g.,* wall-disp.) |
| Tool | *which* tool is used for evaluation (*e.g.,* lviz) |

source code and running system. For the representation we reflected on the description of visualization techniques, analyzed figures, and looked for their description. We extracted the medium by recognizing in the description the technology required to display the visualization such as wall display, standard monitor. We also extracted attributes of tools from the description of the artifact used in the evaluation such as tool name, and availability. When we were not able to identify an attribute, we searched for common terms already found in other studies. When we still did not find a description, we reported it as *not identified*.

We validated the quality of the extracted data by asking the authors of the included studies to review the data of their papers. In particular, for each study we prepared a message that includes the extracted data and classification. We sent the message to the main author of each study, and when their address was not longer valid or we did not receive an answer after some weeks, we sent the message to co-authors. In the few cases that the same person was the main author of several studies, we only sent the message to the co-authors to balance the workload. Unfortunately, we could not contact the authors of 10 studies as their reported e-mail addresses were no longer valid (dash-marked in Tables 2.2 and 2.3). Among the 76 remaining studies, 43 of them (*i.e.,* 57%) contributed to our survey (check-marked in Tables 2.2 and 2.3). Eight studies (S1, S3, S19, S30, S55, S62, S67, S68) completely agreed with our classification (we appreciate the rigorous feedback from S30, S55, S67, and S68, which effectively improved this work); nine studies (S5, S17, S31, S37, S46, S47, S48, S57, S81) agreed with the extracted data and also provided further information

*e.g.,* specified a category for the data that we classified as *not identified*; and twenty-six studies (S2, S6, S9, S10, S11, S14, S16, S20, S23, S24, S25, S27, S33, S41, S42, S43, S44, S45, S50, S59, S61, S70, S71, S73, S75, S77) partially agreed and reclassified some attributes. We observe that the classification of the *representation* used in studies is the greatest source of disagreement. Most authors were not aware of the proposed classification, and preferred to specify a category such as *graph*, *tree*, *glyph*.

## 2.3 Results

In this section we describe various characteristics of the 86 papers.

### 2.3.1 Task

Table 2.4 shows the classification of the papers based on the type of tasks [MMC02] they tackled. Figure 2.5 shows the distribution of the types of tasks presented in each edition of the venues. We sorted the venues chronologically starting by SOFTVIS editions followed by VISSOFT ones. We think it provides a better understanding of their various contributions. We observe that even though we selected papers from all editions of SOFTVIS and VISSOFT, we included only few papers from the first editions of VISSOFT. This can be a consequence of the lower percentage of design study papers in VISSOFT than in SOFTVIS (see Figure 2.1). We also detected that papers tackling testing appear for the first time only in the two last editions of SOFTVIS and then reappear in VISSOFT'14. Although most of the reviewed studies tackled programming tasks (as shown in Table 2.4) they concentrate on SOFTVIS'03 and VISSOFT'15-'16, showing little presence in the rest of the editions. We reflect that the result provides a good overview of the degree of attention that each development concern has had, but since many different visualization techniques are proposed within each type, it provides little help to practitioners to find a suitable visualization for their specific needs. Only the authors of two studies considered multiple categories to classify the task (*[S43]*, *[S61]*). The authors of three other studies proposed to include software comprehension as a category (*[S67]*, *[S27]*, *[S68]*). The authors of one study, while agreeing with our classification (programming), proposed system design and architecture as subcategories.

Table 2.2: The included papers in the study [S1-S50]. The ones reviewed by their authors have a check mark. The ones for which we did not find a valid e-mail address are marked with a dash.

| Id | Reference | Year | Rev. |
|---|---|---|---|
| [S1] | Merge-tree: Visualizing the integration of commits into Linux, **Wild, E. et al.** | 2016 | ✓ |
| [S2] | Visualizing Project Evolution Through Abstract Syntax Tree Analysis, **Feist, M.D. et al.** | 2016 | ✓ |
| [S3] | Visually Exploring Object Mutation, **Schulz, R. et al.** | 2016 | ✓ |
| [S4] | Jsvee & Kelmu: Creating and Tailoring Program Animations for Computing Education, **Sirkiae, T.** | 2016 | |
| [S5] | Towards Visualization of Feature Interactions in Software Product Lines, **Illescas, S. et al.** | 2016 | ✓ |
| [S6] | Perquimans: A Tool for Visualizing Patterns of Spreadsheet Function Combinations, **Middleton, J. et al.** | 2016 | ✓ |
| [S7] | Metrics visualization technique based on the origins and function layers for OSS-based development, **Ishizue, R. et al.** | 2016 | |
| [S8] | DAHLIA 2.0: A Visual Analyzer of Database Usage in Dynamic and Heterogeneous Systems, **Meurice, L. et al.** | 2016 | |
| [S9] | A Visualization Framework for Parallelization, **Wilhelm, A. et al.** | 2016 | ✓ |
| [S10] | An Interactive Microarray Call-Graph Visualization, **Shah, M.D. et al.** | 2016 | ✓ |
| [S11] | On using Tree Visualisation Techniques to support Source Code comprehension, **Bacher, I. et al.** | 2016 | ✓ |
| [S12] | Visualizing Modules and Dependencies of OSGi-based Applications, **Seider, D. et al.** | 2016 | |
| [S13] | vizSlice: Visualizing Large Scale Software Slices, **Alomari, H. et al.** | 2016 | |
| [S14] | Visualization Tool for 3D Graphics Program Comprehension and Debugging, **Podila, S. et al.** | 2016 | ✓ |
| [S15] | CuboidMatrix: Exploring Dynamic Structural Connections in Software Components., **Schneider, T. et al.** | 2016 | |
| [S16] | Walls, Pillars and Beams: A 3D Decomposition of Quality Anomalies, **Tymchuk, T. et al.** | 2016 | ✓ |
| [S17] | Critical Section Investigator: Building Story Visualizations with Program Traces, **Shah, M.D. et al.** | 2016 | ✓ |
| [S18] | Visualizing the Evolution of Working Sets, **Minelli, R. et al.** | 2016 | |
| [S19] | MetaVis: Exploring Actionable Visualization, **Merino, L. et al.** | 2016 | ✓ |
| [S20] | Kayrebt: An Activity Diagram Extraction and Visualization Toolset Designed for the Linux Codebase, **Georget, L. et al.** | 2015 | ✓ |
| [S21] | XVIZIT: Visualizing Cognitive Units in Spreadsheets, **Hodnigg, K. et al.** | 2015 | |
| [S22] | Vestige: A Visualization Framework for Engineering Geometry-Related Software, **Schneider, T. et al.** | 2015 | |
| [S23] | Hierarchical Software Landscape Visualization for System Comprehension: A Controlled Experiment, **Fittkau, F. et al.** | 2015 | ✓ |
| [S24] | Interactive Tag Cloud Visualization of Software Version Control Repositories, **Greene, G.J. et al.** | 2015 | ✓ |
| [S25] | Blended, Not Stirred: Multi-concern Visualization of Large Software Systems, **Dal Sasso, T. et al.** | 2015 | ✓ |
| [S26] | Pixel-Oriented Techniques for Visualizing Next-Generation HPC Systems, **Cottam, J. et al.** | 2015 | |
| [S27] | SMNLV: A Small-Multiples Node-Link Visualization Supporting Software Comprehension, **Abuthawabeh, A. et al.** | 2015 | ✓ |
| [S28] | Live Visualization of GUI Application Code Coverage with GUITracer, **Molnar, A.J.** | 2015 | |
| [S29] | Advancing Data Race Investigation and Classification through Visualization, **Koutsopoulos, N. et al.** | 2015 | — |
| [S30] | Visual Clone Analysis with SolidSDD, **Voinea, L. et al.** | 2014 | ✓ |
| [S31] | Polyptychon: A Hierarchically-Constrained Classified Dependencies Visualization, **Daniel, D.T. et al.** | 2014 | ✓ |
| [S32] | ChronoTwigger: A Visual Analytics Tool for Understanding Source and Test Co-evolution, **Ens, B. et al.** | 2014 | |
| [S33] | Visualizing the Evolution of Systems and Their Library Dependencies, **Kula, R.G. et al.** | 2014 | ✓ |
| [S34] | The visualizations of code bubbles, **Reiss, S.P. et al.** | 2013 | — |
| [S35] | Visualizing software dynamicities with heat maps , **Benomar, O. et al.** | 2013 | |
| [S36] | DEVis: A tool for visualizing software document evolution, **Junji Zhi et al.** | 2013 | — |
| [S37] | SourceVis: Collaborative software visualization for co-located environments , **Anslow, C. et al.** | 2013 | ✓ |
| [S38] | SYNCTRACE: Visual thread-interplay analysis, **Karran, B. et al.** | 2013 | — |
| [S39] | Automatic categorization and visualization of lock behavior, **Reiss, S.P. et al.** | 2013 | |
| [S40] | Chronos: Visualizing slices of source-code history, **Servant, F. et al.** | 2013 | |
| [S41] | Visual support for porting large code bases, **Broeksema, B. et al.** | 2011 | ✓ |
| [S42] | Visualising concurrent programs with dynamic dependence graphs, **Lonnberg, J. et al.** | 2011 | ✓ |
| [S43] | Visual exploration of program structure, dependencies and metrics with SolidSX, **Reniers, D. et al.** | 2011 | ✓ |
| [S44] | MosaiCode: Visualizing large scale software: A tool demonstration , **Maletic, J.I. et al.** | 2011 | ✓ |
| [S45] | An interactive ambient visualization for code smells, **Murphy-Hill, E. et al.** | 2010 | ✓ |
| [S46] | Exploring the inventor's paradox: applying jigsaw to software visualization, **Ruan, H. et al.** | 2010 | ✓ |
| [S47] | Towards anomaly comprehension: using structural compression to navigate profiling call-trees, **Lin, S. et al.** | 2010 | ✓ |
| [S48] | Heapviz: interactive heap visualization for program understanding and debugging, **Aftandilian, E.E. et al.** | 2010 | ✓ |
| [S49] | Trevis: a context tree visualization analysis framework., **Adamoli, A. et al.** | 2010 | |
| [S50] | Dependence cluster visualization, **Islam, S.S. et al.** | 2010 | ✓ |

### 2.3.2 Need

In Table 2.8 and 2.9 we present the developer needs that we identified from studies. Although some studies tackle more than one need we report the most representative one (the complete set of needs is available online[1]). On the one hand, we found that 90% of studies (*i.e.,* 77) describe

[1] `http://scg.unibe.ch/research/visualisation-review`

Table 2.3: The included papers in the study [S51-S86]. The ones reviewed by their authors have a check mark. The ones for which we did not find a valid e-mail address are marked with a dash

| Id | Reference | Year | Rev. |
|---|---|---|---|
| [S51] | Embedding spatial software visualization in the IDE: an exploratory study, **Kuhn, A. et al.** | 2010 | |
| [S52] | Visualizing windows system traces, **Wu, Y. et al.** | 2010 | — |
| [S53] | Zinsight: a visual and analytic environment for exploring large event traces, **de Pauw, W. et al.** | 2010 | |
| [S54] | Representing development history in software cities, **Steinbrückner, F. et al.** | 2010 | |
| [S55] | Case study: Visual analytics in software product assessments, **Telea, A. et al.** | 2009 | ✓ |
| [S56] | Representing unit test data for large scale software development, **Cottam, J.A. et al.** | 2008 | |
| [S57] | A catalogue of lightweight visualizations to support code smell inspection, **Parnin, C. et al.** | 2008 | ✓ |
| [S58] | Streamsight: a visualization tool for large-scale streaming applications, **de Pauw, W. et al.** | 2008 | |
| [S59] | Stacked-widget visualization of scheduling-based algorithms, **Bernardin, T. et al.** | 2008 | ✓ |
| [S60] | "A Bug's Life" Visualizing a Bug Database, **D'Ambros, M. et al.** | 2007 | |
| [S61] | Visualizing Dynamic Memory Allocations, **Moreta, S. et al.** | 2007 | ✓ |
| [S62] | A Visualization for Software Project Awareness and Evolution , **Ripley, R.M. et al.** | 2007 | ✓ |
| [S63] | Experimental evaluation of animated-verifying object viewers for Java, **Jain, J. et al.** | 2006 | — |
| [S64] | Execution patterns for visualizing web services, **de Pauw, W. et al.** | 2006 | |
| [S65] | Visualizing live software systems in 3D, **Greevy, O. et al.** | 2006 | |
| [S66] | Visual exploration of function call graphs for feature location in complex software systems, **Bohnet, J. et al.** | 2006 | |
| [S67] | Multiscale and multivariate visualizations of software evolution, **Voinea, L. et al.** | 2006 | ✓ |
| [S68] | CVSscan: visualization of code evolution, **Voinea, L. et al.** | 2005 | ✓ |
| [S69] | Jove: Java as it happens, **Reiss, S.P. et al.** | 2005 | |
| [S70] | Methodology and architecture of JIVE, **Gestwicki, P. et al.** | 2005 | ✓ |
| [S71] | Visual Exploration of Combined Architectural and Metric Information, **Termeer, M. et al.** | 2005 | ✓ |
| [S72] | Visual data mining in software archives, **Burch, M. et al.** | 2005 | |
| [S73] | The war room command console: shared visualizations for inclusive team coordination, **O'Reilly, C. et al.** | 2005 | ✓ |
| [S74] | Visualizing structural properties of irregular parallel computations, **Blochinger, W. et al.** | 2005 | |
| [S75] | Visualization of mobile object environments, **Frishman, Y. et al.** | 2005 | ✓ |
| [S76] | Towards understanding programs through wear-based filtering, **DeLine, R. et al.** | 2005 | |
| [S77] | Program animation based on the roles of variables, **Sajaniemi, J. et al.** | 2003 | ✓ |
| [S78] | Visualizing Java in action, **Reiss, S.P.** | 2003 | |
| [S79] | EVolve: an open extensible software visualization framework, **Wang, Q. et al.** | 2003 | — |
| [S80] | Visualization of program-execution data for deployed software, **Orso, A. et al.** | 2003 | |
| [S81] | A system for graph-based visualization of the evolution of software, **Collberg, C. et al.** | 2003 | ✓ |
| [S82] | Interactive locality optimization on NUMA architectures, **Mu, T. et al.** | 2003 | — |
| [S83] | Graph visualization for the analysis of the structure and dynamics of extreme-scale supercomputers, **Zhou, C. et al.** | 2003 | — |
| [S84] | KScope: A Modularized Tool for 3D Visualization of Object-Oriented Programs, **Davis, T.A. et al.** | 2003 | |
| [S85] | Self-Organizing Maps Applied in Visualising Large Software Collections, **Brittle, J. et al.** | 2003 | — |
| [S86] | Revision Towers, **Taylor, C.M.B. et al.** | 2002 | |

envisioned user needs by explicitly posing questions that can be answered using the proposed visualization, such as *"what is the software doing when performance issues arise?" [S69]*, *"what does this called method do?" [S76]*. On the other hand, in 10% of studies (*i.e.,* 9) there was no explicit question formulation. In such cases, we identified the goals that the proposed visualization achieves, examples of them being *"to assist designers of scheduling-based, multi-threaded, out-of-core algorithms" [S59]*, *"to get a better insight into the control or data flow inside a program" [S20]*. Although questions allow users to assess whether a visualization is useful, we observe that uncategorized questions hinder the reuse of a visualization. We tackle this issue with a classification of needs based on problem domains. A detailed analysis is provided in Section 2.4.

Table 2.4: Classification of papers based on the tasks.

| Task | Reference | # |
|---|---|---|
| Debugging | S14, S22, S29, S34, S42, S48, S50, S53, S59, S66, S69-S70, S75, S78, S80 | 15 |
| Maintenance | S2-S3, S5, S8-S9, S13, S30, S33, S35, S37-S38, S41, S43-S46, S57, S60-S61, S64, S67-S68 | 22 |
| Programming | S4, S6-S7, S11, S17-S19, S21, S24, S26, S28, S39, S51-S52, S61, S63, S71, S74, S76-S77, S79, S81-S83, S86 | 25 |
| Reverse Engineering | S10, S12, S15, S20, S23, S25, S27, S31, S40, S43, S47, S54, S65, S67, S72, S84-S85 | 17 |
| Software Process Management | S1, S16, S36, S55, S62, S73 | 6 |
| Testing | S32, S49, S56, S58 | 4 |

### 2.3.3 Audience

Software developers play specific roles such as *interaction designer*, *solution architect*, *GUI designer*, *requirements analyst*, *release coordinator*. In contrast, as shown in Table 2.5, 85% of the studies (*i.e.,* 73) envisioned a generic audience described as *developer* (42), *user* (19), *programmer* (17), or *engineer* (5). In the remaining studies the role of the user was more specific such as *project manager* (9), *architect* (7), *maintainer* (5), *tester* (4), or *designer* (2). Less frequent roles were *bug triager*, *HPX developer*, *operation staff*, *performance analyst*, *quality assurance engineer*, and *reviewer*. Some studies envisioned roles of users from other fields such as *business owner* and *student*. One study envisioned managers as well as developers pursuing the same questions *"(1) when were the changes made? (2) what kind of changes have been made? and (3) how does visit / download time vary over time?" [S36]*. Another study envisioned that their tool would be suitable for *"everyone involved in software development" [S62]*. We observed that a better understanding of the scope of the role that an audience plays would *(1)* help researchers to propose solutions focused on the particular problems that roles cope with, and *(2)* facilitate adoption of visualization by practitioners.

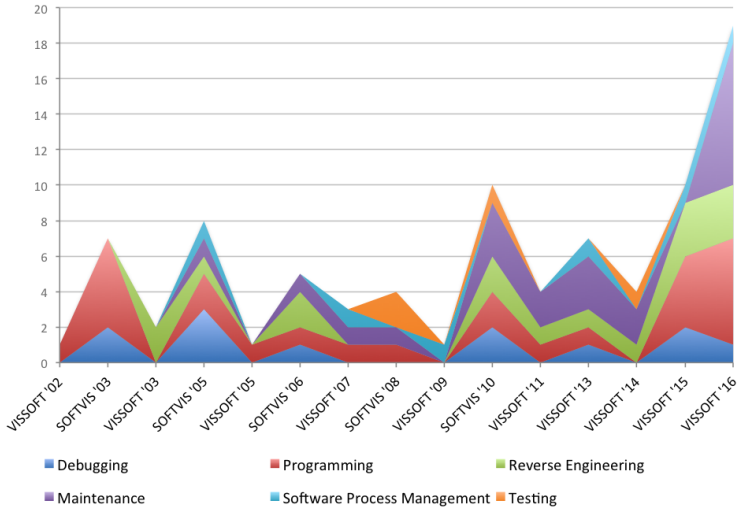



Figure 2.5: Distribution of papers by task in each venue. Bottom-up: Debugging, Programming, Reverse Engineering, Maintenance, Software Process Management and Testing.

### 2.3.4 Data source

Table 2.6 presents various sources of data that are visualized in the studied papers. The most frequent data were gathered from *(1) running system*. Most studies visualized traces of execution, metrics (*e.g.,* CPU usage) and user interactions. Some studies visualized events among applications to analyze operating systems and distributed architectures. A few studies visualized memory accesses and behavior of live objects; *(2) source code* that provided the input to build models of systems for the visualization of dependencies, metrics, structure and inheritance. A few studies visualized annotations used to define features, the scope of variables and program slices. We found that the most frequent language supported was Java, followed by C/C++, which was supported by half of the studies. Other languages with little support include Smalltalk and Pascal; and *(3) version control system*. Most studies visualized meta-data from the commit (*e.g.,* author, date, message), and less frequently changes of code (*e.g.,* added and removed files). Less frequently we found non-traditional sources such

Table 2.5: Classification of papers based on the audience.

| Audience | Reference | # |
|---|---|---|
| Developer | S2, S5, S8-S9, S11-S13, S18-S19, S22-S23, S25-S26, S30-S31, S33, S35-S37, S40-S41, S43-S44, S46-S47, S49-S51, S54, S56-S58, S60, S64-S68, S71, S73, S75-S76, S80 | 42 |
| User | S1, S5, S8, S12-S13, S15, S17-S18, S21, S34, S38, S52, S60, S74, S78-S79, S82-S83, S85 | 19 |
| Programmer | S3-S4, S7, S10, S14, S24, S39, S45, S48, S59, S61, S64, S69-S70, S75, S81, S86 | 17 |
| Project Manager | S2, S12, S32, S44, S54-S55, S57, S68, S73 | 9 |
| Architect | S9, S30, S31, S44, S55, S68, S71 | 7 |
| Manager | S7, S16, S24, S36, S62, S86 | 6 |
| Student | S4, S14, S28, S42, S63, S77 | 6 |
| Engineer | S5, S50, S65, S70, S72 | 5 |
| Maintainer | S13, S33, S50, S68, S80 | 5 |
| Analyst | S13, S16, S53, S75 | 4 |
| Leader | S7, S41, S43, S73 | 4 |
| Tester | S7, SS60, S64, S68 | 4 |
| Researcher | S2, S6, S84 | 3 |
| Designer | S7, S64 | 2 |
| New Team Member | S20, S68 | 2 |
| Practitioner | S6, S28 | 2 |
| Quality Assurance Engineer | S29, S60 | 2 |
| Bug Triager [S2], Business Owner [S64], Coders [S62], End-User [S60], Everyone involved in development [S62], HPX Developer [S26], Linux Kernel Developer [S20], Operation Staff [S64], Performance Analyst [S53], Reviewer [S7], Software Manager [S36], Test Manager [S32] | | 1 |

as *spreadsheets*, *bug tracking systems* (*e.g.,* Bugzilla), *build automation tools* (*e.g.,* Maven), *databases* and *documentation*.

We observe that visualizations have focused on sources of complex data that are difficult to analyze by other means, but this also shows that

sources of complex data are not limited to the traditional ones. We also noticed that studies focus mainly on describing how they modeled data rather than specifying the source and type of data. We observe that detailed descriptions of data that include not only the source but the format as well as other characteristics can help developers to adopt visualizations. For instance, users who are aware of a technique for visualizing a stack trace gathered from a running system can decide whether their context is similar enough to adopt the visualization.

### 2.3.5 Representation

Describing the representation used in a visualization is a complex task. Authors proposing a visualization use various strategies to describe the applied techniques. Some used verbose descriptions *[S62, S65]* by specifying dimensions, metaphors, marks, and properties of them. Others *[S68, S71]* opted for concise but sometimes vague descriptions. We classify the visualization techniques used in the studies according to the popular taxonomy proposed by Keim [Kei02]. This taxonomy provides a concise list of categories upon which abundant research has relied. In it, visualization techniques can belong to one of four categories (examples are shown in Figure 2.6): *(1) Stacked* techniques that are tailored to present data in a hierarchical fashion (*e.g.,* Treemaps and Cone Trees); *(2) Iconic* techniques that map the data attributes to the features of an icon (*e.g.,* CocoViz *[S46]*); *(3) Geometrically-Transformed* techniques that aim at finding interesting transformations of data attributes (*e.g.,* Scatter-plots and Parallel Coordinates); *(4) Dense Pixel* techniques that map each data attribute to a colored pixel and group the pixels belonging to each attribute into adjacent areas (*e.g.,* Vampir *[S7]*) ; and *(5) Standard 2D/3D* techniques such as Bar Charts, X-Y Plots;

Table 2.7 presents these categories. We note that approximately half of the studies (*i.e.,* 44) combine techniques from several categories. The most frequent combination occurred between Treemaps and Node-link diagrams (Stacked and Geometrically-Transformed). Combinations of other types of techniques occurred with less frequency (less than four studies). Frequent types are Geometrically-Transformed (GT), Dense Pixel (DP) and Stacked (ST). We observe that GT is frequent since node-link techniques, that belong to this category, are commonly used by visualizations that explore relationships. The DP type contains techniques suitable for depicting mas-

Table 2.6: Classification of papers based on the data source.

| Data Source | | | Reference | # |
|---|---|---|---|---|
| Running System (41) | Trace | Execution | S3, S9-S10, S14-S15, S17, S20, S22, S28-S29, S38-S39, S42, S49, S56, S59, S65, S69-S70, S74, S78-S80 | 22 |
| | | Metric | S17, S47, S66, S71 | 4 |
| | | Interaction | S62, S76 | 2 |
| | Application events | | S23, S52, S53, S58, S64, S75, S83 | 7 |
| | Memory accesses | | S9, S20, S48, S61, S80, S82 | 6 |
| | Live objects | | S9, S18-S19, S63 | 4 |
| Source Code (31) | Dependency | | S4, S8, S12-S13, S27-S28, S31, S37, S43-S44, S46, S50, S54-S55, S63, S65-S66, S77, S81, S84-S85 | 22 |
| | Metric | | S3, S12, S30, S37, S41, S43-S46, S51, S55, S71 | 12 |
| | Structure | | S8, S27, S31, S37, S43, S44, S46, S55, S65-S66, S79 | 11 |
| | Inheritance | | S81, S84-S85 | 3 |
| | Annotation [S5], Scope [S11], Slice [S13] | | | 1 |
| Version Control System (16) | Meta-data | | S1-S2, S26, S32, S34-S36, S40, S67-S68, S72-S73, S86 | 13 |
| | Code Changes | | S2, S8, S16, S32, S34-S36, S55, S67-S68 | 10 |
| Spreadsheet (2) | | | S6, S21 | 2 |
| Others (4) | Bug Tracking System [S60], Build Automatic Tool [S33], Database [S8], Documentation [S71] | | | 1 |

sive data sets such as Heatmap. ST also includes popular techniques for hierarchical data such as Treemap.

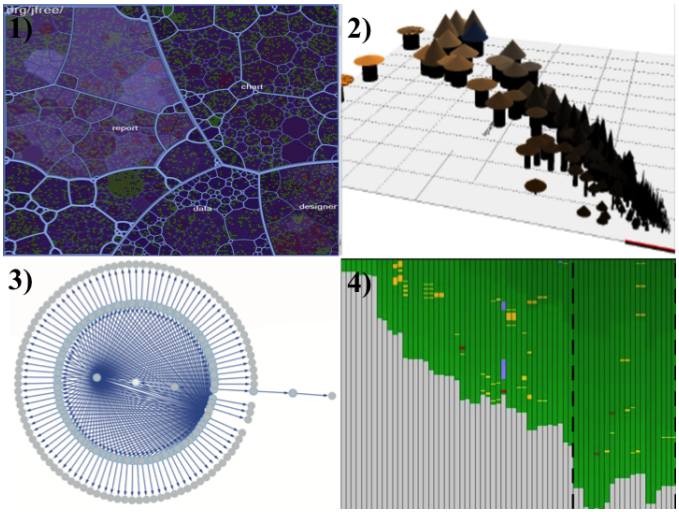



Figure 2.6: Examples of visualizations for each type (1) Stacked, (2) Iconic, (3) Geometrically-Transformed, and (4) Dense Pixel.

### 2.3.6 Tool

Tables 2.8 and 2.9 summarize the tools collected from the papers. Normally, they are developed as prototypes to evaluate a proposed visualization. All studies, among the 77 that explicitly identified a tool (*i.e.,* 90%), introduced a new visualization tool. Notice that the tool named *Jive* that was used in two studies *[S70, S78]* corresponds to a different tool. A few (*i.e.,* 26%) made their tool and source code publicly available. As one can expect, few prototypes were maintained and extended over time. The most notable cases are *Jive [S70]*, and two tools used for teaching programming: *jGrasp [S63]* and *PlanAni [S77]*. If we consider tools for which current information is available, their average lifespan is 3.7 years[2]. We acknowledge that this value represents only a lower bound, since it does not consider possible earlier presentations of the tools. Various studies often used different visualization frameworks. The most frequent ones are *OpenGL* (11) used over multiple years, and *D3.js* (9) and *Roassal* (6) used only recently. Also, three studies used *Java3D* in more than a decade ago. *GraphViz* was used

[2]We measured lifespan as the time between a tools' first appearance in a publication and the last update to the projects' repository.

Table 2.7: Classification of papers based on the representation.

| Representation Type | Technique | Reference | # |
|---|---|---|---|
| Geometrically-Transformed | Node-link diagram (Tree Layout) | S1, S6, S9, S11, S14, S17, S23, S27, S31, S36-S38, S40, S42-S43, S48, S55-S56, S64 | 16 |
| | Node-link diagram (Force-directed Layout) | S5, S10, S12, S18-S19, S31-S32, S65, S69, S72, S74, S81 | 11 |
| | Hierarchical Edge Bundle | S30, S41, S43, S55 | 4 |
| | Parallel Coordinate | S13, S46 | 2 |
| | Scatterplot | S32, S46 | 2 |
| Stacked | Treemap | S1, S5, S11, S12, S13, S35, S43, S55, S80 | 9 |
| | Icicle Treemap | S11, S17 | 2 |
| | Sunburst | S49 | 1 |
| Pixel | Heatmap | S22, S26, S34-S35, S39, S50, S53, S57, S59, S60-S61, S67-S68, S73 | 14 |
| | Matrix | S3, S15-S16, S44, S52, S72, S82 | 7 |
| | Table Lens | S41, S43, S50, S80 | 4 |
| Iconic | | S4, S7, S8, S15-S16, S19, S24-S25, S35-S36, S45-S47, S51, S56, S60, S62, S67, S69, S71, S83, S85 | 22 |
| Dense | | | |
| Standard | | S3-S4, S9, S14, S21, S28, S33, S58, S59, S63-S64, S66, S70, S75-S77, S79 | 17 |

in four studies. The rest of the studies use multiple frameworks, and in twenty-three there is no explicit information about any frameworks used.

### 2.3.7 Medium

In our previous work [MGN16a], we included the medium as one of the dimensions of the proposed software visualization taxonomy. Although the authors of a previous taxonomy [MMC02] envisioned a future where

Table 2.8: Visualization tools and needs extracted from papers [S1-S50].

| Ref. | Tool | Year | Framework | Questions and Goals that Motivate Visualization |
|---|---|---|---|---|
| [S1] | Linvis | 2016 | Python, Flask | How and by whom commits arrive and merge into the Linux repository? |
| [S2] | TypeV | 2016 | JavaScript, TypeScript, D3.js | What is a developer's contribution to a repository? |
| [S3] | Object Evolution Blueprint | 2016 | Smalltalk, Roassal | How the value of variables change during the execution of a program? |
| [S4] | Jsvee | 2016 | JavaScript | How program code behaves when it is executed? |
| [S5] | ECCO | 2016 | JavaScript, D3.js | How features are implemented and interact? |
| [S6] | Perquimans | 2016 | JavaScript, D3.js | How are spreadsheet users building formulae? |
| [S7] | MAF and OC | 2016 | JavaScript, HTML5 | To show the origins and function layers in development |
| [S8] | DAHLIA | 2016 | Not identified | How the database elements are mapped in the Java code? |
| [S9] | Parceive | 2016 | JavaScript, D3.js | To assist users in identifying scenarios that benefit from parallelization |
| [S10] | Not identified | 2016 | Java, Processing | How execution time is spent in the program? |
| [S11] | Not identified | 2016 | JavaScript, D3.js, ace.js, esprima.js, estraverse.js,escope.js | What is the static structure of a source code document? |
| [S12] | Not identified | 2016 | JavaScript, D3.js | To analyze software structure and dependencies |
| [S13] | vizSlice | 2016 | JavaScript, D3.js | What parts of the software can be affected by a change? |
| [S14] | Not identified | 2016 | JavaScript, D3.js | Is the data transferred correctly from CPU to GPU? |
| [S15] | CuboidMatrix | 2016 | Smalltalk, Pharo, Roassal | How code critiques are distributed in the software? |
| [S16] | A Roassal 3D visualization | 2016 | Smalltalk, Pharo, Roassal3D | What are quality evolution anomalies and what caused them? |
| [S17] | CSI | 2016 | Java, Processing | How call trees behave for critical sections in a Java program? |
| [S18] | Not identified | 2016 | Smalltalk, Pharo, Roassal | How developers navigate and interact with code during development? |
| [S19] | MetaVis | 2016 | Smalltalk, Pharo, Roassal | What visualizations are suitable to answer development questions? |
| [S20] | Kayrebt | 2015 | C, C++, GraphViz, Qt | What is actually compiled by the compiler? |
| [S21] | XVIZIT | 2015 | Java FX, Control FX, GraphStream | What would be affected if I were to change a cell? |
| [S22] | Vestige | 2015 | C++ OpenGL | How the computation reached that result? |
| [S23] | ExploreViz | 2015 | JavaScript, tree.js | What are the consequences of a failure in a certain application? |
| [S24] | ConceptCloud | 2015 | Play web framework | How often and by whom certain files have changed together? |
| [S25] | Blended City | 2015 | Smalltalk, Pharo, Roassal | What happened to a software system in a given time frame? |
| [S26] | Vampir | 2015 | Not identified | How different are work queues on different threads? |
| [S27] | SMNLV | 2015 | Java 8, Graphisto Toolkit, abego TreeLayout, NetBeans Visual Library | To check guidelines and re-engineering of existing software, |
| [S28] | GUITracer | 2015 | Java 6 using AWT, Swing | What source code runs once a GUI event is fired? |
| [S29] | RaceView | 2015 | C, Eclipse Visualization Zest | How a specific code location can be reached via function calls? |
| [S30] | SolidSDD | 2014 | C, C++ , OpenGL | How are clones distributed across system structure? |
| [S31] | Polyptychon | 2014 | JavaScript, D3.js | Are there any patterns in the dependency structure? |
| [S32] | ChronoTwigger | 2014 | OpenGL, GLUT, VR Juggler | How source and test files develop together over time? |
| [S33] | Not identified | 2014 | R, GGPlot2 | When should I update my library dependencies |
| [S34] | Code Bubbles | 2013 | Not identified | How are Java programs based on working sets developed? |
| [S35] | VERSO | 2013 | Not identified | How programmers behave during the evolution of a program? |
| [S36] | DEVis | 2013 | Java, G4P | When and what kind of changes have been made? |
| [S37] | SourceVis | 2013 | Java, MT4j, OpenCloud, JFreeChart | What is the structure and properties of software? |
| [S38] | SYNCTRACE | 2013 | Not identified | Where and when a thread waits or releases? |
| [S39] | Not identified | 2013 | Not identified | Which locks interact with one another and how complex is it? |
| [S40] | Chronos | 2013 | Java | When, how, by whom, and why was this code changed or inserted? |
| [S41] | PortAssist | 2011 | C++, Qt, OpenGL | Which rewrite activities conflict with each other? |
| [S42] | Atropos | 2011 | Java, Apache Commons BCEL, Matrix software visualisation framework | How do operations executed in a Java program relate to each other? |
| [S43] | SolidSX | 2011 | OpenGL, GLUT, FTGL, wxWidgets | How metrics correlate with the dependencies in the system? |
| [S44] | MosaiCode | 2011 | C++, Qt | How metrics have changed? Where are run time bottlenecks? |
| [S45] | Stench Blossom | 2010 | Java, Eclipse | What code smells are present in the code I am working with? |
| [S46] | Jigsaw | 2010 | Not identified | What entities are likely to depend on this package? |
| [S47] | ProfVis | 2010 | Java, HProf, Processing | What parts of the program could be modified to improve performance? |
| [S48] | Heapviz | 2010 | Prefuse toolkit | What is the shape of the data structures, and how are they connected? |
| [S49] | Trevis | 2010 | Trevis, GraphViz | To study the calling contexts where the program spent most time |
| [S50] | Decluvi | 2010 | Java | What is the dependence structures/clusters in your program? |

software visualizations would use a variety of media, we found few studies exploiting this dimension, shown in Table 2.10. Almost 56% of the reviewed studies do not mention the expected medium on which the vi-

Table 2.9: Visualization tools and needs extracted from papers [S51-S86].

| Ref. | Tool | Year | Framework | Questions and Goals that Motivate Visualization |
|---|---|---|---|---|
| [S51] | CodeMap | 2010 | Not identified | What is the purpose of the application? Who are the collaborators? |
| [S52] | Iviz | 2010 | OpenJDK 1.6.0 | How the operating system works? |
| [S53] | Zinsight | 2010 | Not identified | How did we get to these events? |
| [S54] | CrocoCosmos | 2010 | jMonkeyEngine | How the component content changes over time? |
| [S55] | Not identified | 2009 | Not identified | How metrics evolve in time over the entire software system? |
| [S56] | SeeTest | 2008 | Stencil visualization environment | How did the changes from yesterday affect project's stability? |
| [S57] | NosePrints | 2008 | Not identified | How widespread and how difficult a problem may be to fix? |
| [S58] | Streamsight | 2008 | dot | How the system and applications evolve? |
| [S59] | Lumiere | 2008 | OpenGL, C++ | How are concurrent tasks scheduled by the algorithm? |
| [S60] | Bug Watch | 2007 | Not identified | How the bugs are distributed in the system over time? |
| [S61] | MemoView | 2007 | C++, OpenGL, FLTK | How does fragmentation depend on time and pool? |
| [S62] | Palantír | 2007 | Java | When was the artifact changed? |
| [S63] | jGrasp | 2006 | Not identified | To understand concepts of dynamic programming implementation |
| [S64] | IBM Web Services Navigator tool | 2006 | Not identified | How different IT resources interact sequentially with one another? |
| [S65] | TraceCrawler | 2006 | CCJun | How the system behaved during the execution of a feature? |
| [S66] | Call Graph Analyzer | 2006 | GraphViz | Which the important functions for feature understanding are? |
| [S67] | CVSgrab | 2006 | Python, wxWidgets, OpenGL, C | How metrics correlate during evolution of a given set of items? |
| [S68] | CVSscan | 2005 | Python, wxWidgets, OpenGL, C | What code lines were added, removed, or altered, when and by whom? |
| [S69] | Jove | 2005 | Not identified | What the software is doing when performance issues arise? |
| [S70] | Jive | 2005 | Not identified | What is the runtime object structure of a Java program? |
| [S71] | MetricView | 2005 | C++, OpenGL, FreeType, wxWindows | Where are components having certain properties? |
| [S72] | EPOSee | 2005 | Not identified | What items have been changed at the same time? |
| [S73] | War Room Command Console | 2005 | Java, C++ | Who is currently working on what? |
| [S74] | DOT | 2005 | Java, yFiles library | What are optimal parameters to distribute the work on the processors? |
| [S75] | Mobile Object Visualization | 2005 | Java, Java3D, GraphViz | How does the architecture supports object mobility behavior over time? |
| [S76] | FAN | 2005 | Not identified | Which method in the source code implements certain behavior? |
| [S77] | PlanAni | 2003 | Tcl/Tk | How the successive values of the variable relate to each other and to other variables? |
| [S78] | Jive | 2003 | Not identified | What threads are in the program? |
| [S79] | EVolve | 2003 | Not identified | When and for how long particular events occur? |
| [S80] | Gamma / Gammatella | 2003 | Java, Swing, TreeMap Java Library | How often the statement is executed? |
| [S81] | GEVOL | 2003 | Not identified | How and by whom the parts of the program were created? |
| [S82] | Not identified | 2003 | Not identified | How much of the data is dominantly accessed by the local nodes? |
| [S83] | Flatland | 2003 | OpenGL | To analyze massively parallel supercomputer architectures |
| [S84] | Kscope | 2003 | Java3D | To provide an analysis of Java programs |
| [S85] | GENISOM | 2003 | Java3D | To aid programmers in the process of reverse engineering |
| [S86] | Revision Towers | 2002 | Not identified | How often, and how, changes are made? |

sualization should be displayed (labeled as not identified). Among the 44% that explicitly mentioned a medium the majority (*i.e.,* 87%) specified the standard PC display. However, there were other studies that indicated diverse media from a small window in a standard monitor to a wall-display, large multi-touch tables, tablets, 3D glasses and immersive virtual reality.

Table 2.10: Classification of papers based on the medium.

| Medium | Reference | # |
|---|---|---|
| Not Identified | S1-S3, S5, S7-S8, S12-S13, S15, S17-S18, S21-S22, S25-S26, S28-S29, S31, S34-S36, S38-S40, S46, S48-S49, S51-S56, S58, S60, S63-S64, S66, S69, S72, S74, S76, S78-S80, S82, S85-S86 | 48 |
| Standard screen | S4, S6, S9-S11, S14, S16, S19-S20, S23-S24, S27, S30 ,S41-S43, S45, S47, S50, S57, S59, S61-S62, S65, S67-S68, S70-S71, S75, S77, S81, S83-S84 | 33 |
| Wall-display | S27, S33, S57, S62, S73 | 5 |
| 3D glasses [S71], Immersive Virtual Reality [S32], Tablet [S4], Multi-Touch Table [S37], Multi-Monitor [S44] | | 1 |

## 2.4 Discussion

In this section we discuss our findings, and we provide recommendations to practitioners and researchers, respectively, for adopting visualizations, and for identifying domains that require more attention.

A majority of studies do not follow a specific structure for describing their proposed techniques. We believe that following a specific structure encourages researchers to reflect on important dimensions that should drive the design of a visualization tool [MMC02, SvG05]. Moreover, we believe that providing a clear description of a research problem, and formulating explicit research questions ease tool adoption by practitioners. For instance, instead of a fuzzy description like "*provides an analysis of Java programs*" *[84]* which does not reflect an exact goal, we suggest a reformulation to "*analyze class dependency for validation of experimental software visualization techniques*."

In section 2.3.1 we classified the papers into six high-level software development tasks (shown in Table 2.4). We note that a different visualization is proposed to tackle developer needs that are classified in the same task. Hence, we argued that such a classification does not provide an appropriate support for practitioners to find and adopt a suitable visualization for their specific needs. We observe that practitioners require a more fine-grained classification that links existing visualization techniques to their concrete needs.

We observe that researchers who focused on the questions that developers ask during software development have classified the type of questions using diverse criteria. We rely on that research to *(1)* identify the used categories to classify the design study papers, and *(2)* evaluate how important are those categories based on the number of different type of questions and their frequency.

We classify the studies into *(1) subject-oriented* [FM10], *(2) process-oriented* [SMDV06], and *(3) problem-oriented* [LM10]. We believe mapping such classifications of developer needs to the visualization techniques proposed by studies provides a better support for practitioners to adopt a visualization in their daily tasks and allows us to analyze how well a proposed visualization supports developers to answer questions that actually arise during development. According to our investigation, these classifications offer an appropriate granularity to accommodate the questions from other studies too. Hence, we classified the 86 included papers by identifying categories in each classification that contain similar types of questions to the needs extracted from the papers (shown in Tables 2.8 and 2.9). In studies for which we extracted a goal instead of a question, we inferred the category from other types of questions that would help users to achieve that goal. In the following we present the classification of studies based on the classifications: subject, process and problem-oriented.

*Subject-oriented.* Fritz *et al.* [FM10] proposed a classification in which questions can belong to one of the following categories: *people* (*e.g.,* who is working on what), *code* (*e.g.,* changes to the code), *progress* (*e.g.,* work item progress), *build* (*e.g.,* broken builds), *test* (*e.g.,* test case analysis), *web* (*e.g.,* web related concerns), and *other* questions. The result of the classification of the studies is shown in Table 2.11. We found that visualizations that we classified as dealing with *(i) code* particularly focused on subjects such as architecture, commits, critiques, features, memory management, threads, and working sets, *(ii) other* spanned subjects such as databases, event traces, failure reports, and visualization examples, *(iii) people* subjects addressed ownership, *(iv) test* subjects focused on GUI and compiler, and *(v) build* subjects were related to compiler optimizations and build configurations.

*Process-oriented.* Sillito *et al.* [SMDV06] introduced a classification that focuses on understanding the cognitive process of moving from questions to answers. Their classification of type of questions includes the following categories: *(i) finding* initial software entities that might lead developers to formulate a concrete question, *(ii) building* on those points

Table 2.11: Subject-oriented classification of studies.

| Category | References | # |
|---|---|---|
| Code | S2, S5-S7, S11-S13, S21, S23-S25, S27-S37, S40-S41, S43-S46, S50-S51, S54-S57, S60, S62, S67-S68, S71-S72, S76, S81, S84-S86 | 45 |
| Other | S1, S3-S4, S8-S10, S14-S20, S22, S26, S38-S39, S42, S47-S49, S52-S53, S58-S59, S61, S63-S66, S69-S70, S73-S75, S77-S80, S82-S83 | 41 |
| People | S2, S6, S24, S34-S35, S40, S51, S54-S55, S68, S81, S86 | 12 |
| Test | S28, S32, S56, S57 | 4 |
| Build | S25 | 1 |
| Progress | - | 0 |
| Web | - | 0 |

by identifying relationships between entities, *(iii) understanding* a group of entities and relationships, and *(iv) questioning* how various groups relate each other. In Table 2.12 we present the results of the classification of studies. We observe that as the cognitive process increments in complexity (finding $\rightarrow$ building $\rightarrow$ understanding $\rightarrow$ questioning) the number of proposed visualization decreases.

Table 2.12: Process-oriented classification of studies.

| Category | References | # |
|---|---|---|
| Finding | S1-S86 | 86 |
| Building | S1-S2, S4-S18, S20-S26, S28-S61, S63-S66, S68-S71, S74-S81, S83-S86 | 78 |
| Understanding | S2, S4-S9, S12-S18, S20-S23, S25-S26, S28-S33, S38-S39, S42, S44-S47, S49-S53, S55-S57, S59, S63-S65, S68-S71, S76-S78, S81, S85-S86 | 55 |
| Questioning | S2, S8, S13-S17, S20-S21, S23, S28-S29, S32-S33, S39, S51, S53, S55-S56, S59, S65, S69, S76-S77, S86 | 25 |

*Problem-oriented.* LaToza *et al.* [LM10] proposed a classification that comprises 21 problem domains which they used to categorize 94 types of

questions. Table 2.13 presents the obtained results of the classification of the 86 studies.

In summary, we observe that in the subject-oriented classification the majority of questions supported by visualizations relates to the code category. There is also a moderate number of visualizations that support questions that focus on people. Certainly, both represent the main subjects of software visualization. Indeed, Figure 2.8 shows that both code and people are balanced. The chart also shows that there are a few subjects such as Builds, Progress, and Web that even though represent developer needs have less attention from proposed visualization.

In the process-oriented classification, we notice that visualization provides a good fit to the mental process of developers (*i.e.,* the more complex the mental process, the more visualizations are available). Typically, visualizations provide developers with an overview that help them to *find* interesting patterns. Developers can reflect on those patterns and *build* hypotheses. Developers can test hypotheses by getting details-on-demand of elements that lead in a deeper *understanding* of the system artifact. Finally, developers are able to answer complex *questions* by combining their understanding on multiple findings.

We observe that the problem-oriented classification is more fine-grained, thus facilitating the analysis to understand the relationships between the needs of developers and the proposed visualization.

In the following section we revisit our research questions based on the described classifications.

***RQ.1)*** *What are the characteristics of visualization techniques that support developer needs?*

While few problem domains in the classification (like debugging and testing) seem to be a task by themselves, they also occur very often in the context of addressing different tasks. That is, a visualization proposed to support questions regarding performance during a reverse engineering task (*e.g.,* "where is most of the time being spent?" *[S10]*) may differ from the one proposed for performance questions that arise during a debugging session (*e.g.,* "how did we get to these events?" *[S53]*). Figure 2.7 shows the mapping between the problem domains and the types of visualization techniques. In it, problem domains are labeled. The ones in the same category are vertically aligned (left-to-right changes, element relationships, and

Table 2.13: Problem-oriented classification of the 86 design studies.

| | Problem domain | Reference | # |
|---|---|---|---|
| Changes | Building and branching | - | 0 |
| | Debugging | S3, S14, S22, S34, S42, S48, S51, S53, S58, S59, S66, S69, S70, S78, S80 | 15 |
| | History | S1, S2, S16, S24, S25, S35, S36, S40, S44, S54, S56, S60, S62, S67, S68, S72, S86 | 17 |
| | Implementing | S6, S7, S11, S18, S19, S22, S30, S44, S58, S59, S63, S68 | 12 |
| | Implications | S4, S6, S33, S41, S44, S47, S51, S71, S73, S76, S79 | 11 |
| | Policies | - | 0 |
| | Rationale | S11, S81 | 2 |
| | Refactoring | S6, S11, S45 | 3 |
| | Teammates | S2, S24, S35, S37 | 4 |
| | Testing | S22, S44, S51, S57, S60, S84 | 6 |
| Element relationships | Architecture | S5, S8, S11, S12, S16, S23, S30, S32, S45, S51, S55, S83, S85 | 13 |
| | Contracts | S27 | 1 |
| | Control flow | S10, S11, S20, S42, S59, S65 | 6 |
| | Data flow | S8, S20, S42, S59 | 4 |
| | Dependencies | S5, S8, S13, S15, S21, S27, S31, S42, S43, S46, S50, S75, S76 | 13 |
| | Type relationships | S27 | 1 |
| Elements | Concurrency | S9, S17, S26, S29, S38, S39, S42, S59 | 8 |
| | Intent and implication | S4, S51, S73, S76 | 4 |
| | Location | S5, S28, S44, S51, S65, S77 | 6 |
| | Method properties | S16, S44 | 2 |
| | Performance | S49, S52, S53, S58, S61, S64, S74, S82 | 8 |

elements). The colors of the tiles encode the type of visualization technique used by studies tackling that domain. Problem domains that did not match any studies are shown in black. The size of a tile is proportional to the number of studies classified in that domain. Looking at the distribution of visualization techniques across the types of problem domains (*i.e.,* changes, element relationships and elements) we do not perceive a preferred one. Instead, we observe that dense pixel and geometrically-transformed are the most frequent techniques used in the main problem domains such as history, debugging, performance. In contrast, iconic techniques are present in only a few domains, but when present they predominate over other techniques such as history, implications and testing. Iconic techniques enforce comparison of multivariate data by mapping their properties to the various dimensions of a glyph (including its position). Questions regarding the history domain frequently involve the time, which is commonly mapped to the position. We think that this is the reason why most visualizations proposed to tackle needs in the history domain include iconic techniques.

***RQ.2)*** *How well are developer needs supported by visualization?*

Since there is a large number of questions that developers need to answer during development, we analyze them at a higher level by using classifications proposed by research in the field. For each classification, we estimate the importance of categories for practitioners based on the number of type of questions (and frequencies) that they contain. The more questions a category contains, the more important that category is for developers. The metric is then normalized to enabling comparison.

Figures 2.8, 2.9 and 2.10 compare the importance of developer needs (red bar) versus the number of visualization techniques proposed to address those needs (blue bar). They show that *(1)* when analyzing the *subjects* associated with a need [FM10] (see Figure 2.8) questions related to *code* and *people* are of high relevance for developers. Although researchers invest an adequate attention to the *code* category, many proposed visualizations are classified in the *others* category. The inspection of those questions shows that they relate to runtime analysis. That *(2)* the analysis of the importance of questions by the *process* [SMDV06] (see Figure 2.9) in which they are involved shows that developer needs and the proposed visualizations are almost in sync. Most questions that developers pose, involve *building* on initial findings. However, visualizations have focused slightly more on


Figure 2.7: Mapping type of visualization used by studies to problem domains.

understanding than purely building. Finally, *(3)* we analyze the needs of developers by grouping questions at problem domain [LM10] (see Figure 2.10). We observe that practitioners are more concerned about *changes*, while existing visualizations distribute their attentions among all three categories. Some problem domains (*e.g.,* rationale, intent, implementation, and refactoring) are very important for developers but have little visualization support. In contrast, several less important problem domains (*e.g.,* architecture, concurrency and dependencies) received a good degree of

attention. We wonder *why some are not supported?* We conjecture that less well-supported domains tackle problems that require hidden semantics to be inferred from software artifacts, so proposing a visualization is difficult.

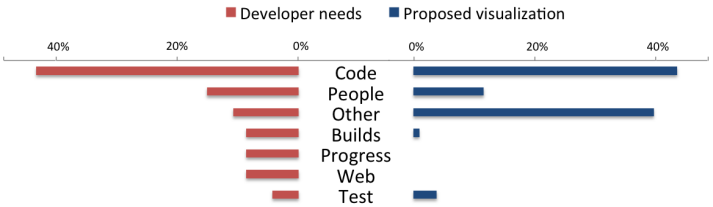



Figure 2.8: Subject-oriented analysis of importance of developer needs vs. their visualization support. The large number of visualizations in the *others* category exposes limitations of this classification.


Figure 2.9: Process-oriented analysis of importance of developer needs vs. their visualization support. The focus on each mental process seems well balanced.

### 2.4.1 Threats to Validity

The main threat to the validity of our study is bias in paper selection. We did not include papers from other venues. We mitigated this threat by selecting peer-reviewed papers from the most cited venues dedicated to software visualization. Moreover, we included design studies and excluded other types of papers. However, since most of papers do not specify their types, we may have missed some. We mitigated this threat by defining a cross-checking procedure and criteria for paper type classification. Finally, the data extraction process could be biased. We mitigated this by establishing

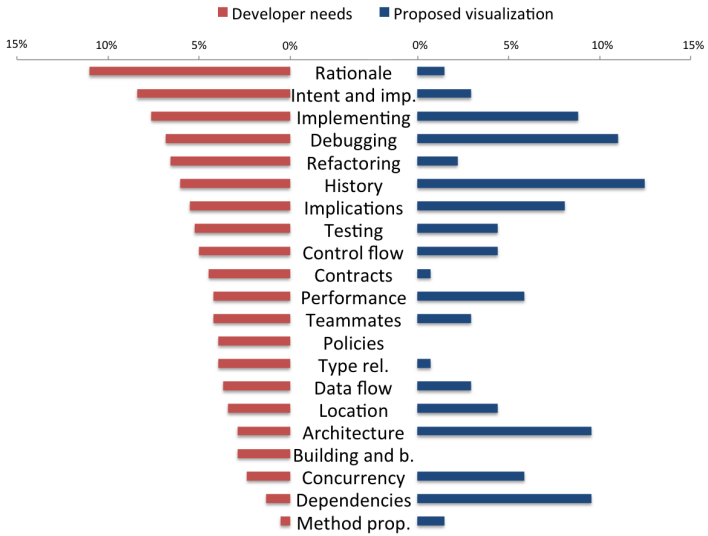



Figure 2.10: Problem-oriented analysis of importance of developer needs vs. their visualization support. Ideally, both charts would be symmetrical. However, we observe several unbalanced domains.

a protocol to extract the data of each paper equally; and by maintaining a spreadsheet to keep records, normalize terms, and identify anomalies.

## 2.5 Conclusion

We studied 86 publications in academia that describe how visualization techniques can help developers to carry out their tasks, and we investigated how well practitioner needs are supported by existing visualization techniques. On the one hand, we analyze research that describes complex questions that practitioners often ask during software development. On the other hand, we reviewed the literature looking for the needs that benefit from particular visualizations. We compared the degree of importance of needs grouped by subject, process and problem domains for practitioners to the visualization support available for them. Although the developer needs grouped by subject and process seem well supported by visualization, we

found a disconnect between the problem domains on which visualizations have focused and the domains that get the most attention from practitioners. The results of our literature study suggest that some problem domains such as rationale, refactoring, contracts and policies require more attention from the visualization community, while a considerable amount of work is devoted to dependencies, architecture and concurrency.

Our characterization provides an overview of the state-of-the-art in software visualization. We identified a collection of relevant proposed software visualization approaches, and classified them into various categories based on main characteristics. However, we did not include in the analysis whether those proposed visualization approaches have proved effective to support the targeted software engineering task. In the following chapter we elaborate on such aspect.

# 3

# Software Visualization Evaluation

## 3.1 Introduction

A developer can obtain the benefits of adopting a software visualization approach only if that approach has proven to be effective to support the target software engineering task. Therefore, we look into the evidence presented in the evaluations of proposed software visualizations.

Indeed, researchers adopt varying strategies to evaluate software visualization approaches, and therefore the quality of the evidence of their effectiveness varies. We believe that a characterization of the evaluation of software visualization approaches will *(i)* assist researchers in the field to improve the quality of evaluations, and *(ii)* increase the adoption of visualization among developers.

We consider previous research to be an important step to characterizing the evidence of the effectiveness of software visualization approaches. However, we reflect that previous research has failed to define what is an effective software visualization, and consequently comparing the effectiveness of visualization approaches is not possible. Moreover, we believe that some studies have used a loose definition of "case studies" and include many usage scenarios of visualization instead that present little evidence of

the effectiveness of an approach. In our investigation we perform a subtler analysis of the characteristics of evaluations to elucidate these concerns. Consequently, we formulated the following research questions:

*RQ1.*) What are the characteristics of evaluations that validate the effectiveness of software visualization approaches?

*RQ2.*) How appropriate are the evaluations that are conducted to validate the effectiveness of software visualization?

We believe that answering these questions will assist researchers in the software visualization field to improve the quality of evaluations by identifying evaluation strategies and methods and their common pitfalls. In particular, we reviewed 181 full papers of the 387 papers published in SOFTVIS/VISSOFT. We identified evaluation strategies such as surveys, case studies, and experiments, as well as characteristics such as tasks, participants, and systems used in evaluations. We found that 62% (*i.e.,* 113) of the proposed software visualization approaches either do not include any evaluation, or include a weak evaluation (*i.e.,* anecdotal evidence, usage scenarios). Almost all of them (*i.e.,* 110) introduce a new software visualization approach. The remaining three discuss an existing approach but without providing a stronger evaluation. We also found that 29% of the studies (*i.e.,* 53) conducted experiments in which 30% (*i.e.,* 16) corresponded to visualizations that target the novice developer audience, and included appropriate participants. The remaining 70% proposed visualizations for developers with various levels of experience. However, amongst them only 30% included experienced developers, and the remaining 70% (*i.e.,* 37) included in experiments only students and academics of a convenience sample who are vulnerable to selection bias and hence hinder generalization. We found that 7% (*i.e.,* 12) of the studies conducted a case study that involved (*i*) professional developers from industry, and (*ii*) real-world software systems. Finally, 3% (*i.e.,* 4) of studies conducted a survey. Even though we are not aware of a similar quantitative report of the state of the art in information visualization, a review of the practice of evaluation [IIC+13] found similar issues.

We believe that for software visualization approaches to be adopted by developers, visualizations not only must prove their effectiveness via evaluations, but evaluations should also include participants of the target audience, and be based on real-world software systems. Finally, we recommend researchers in the field to conduct surveys that can help them to identify what are the frequent and complex problems that affect developers.

A few studies have attempted to characterize the evaluation of software visualization approaches via a literature review. For instance, Schots and Werner [SW14] reviewed 36 papers published between 1993 and 2012 and proposed an extended taxonomy that includes evidence of the applicability of a software visualization as a dimension [SVW14]. They found that papers lacked a clear description of information related to the evidence on the use of visualization. Seriai *et al.* [SBCS14] analyzed 87 papers published between 2000 and 2012. They found that most visualizations are evaluated via case studies (*i.e.,* 78.16%), and only a few researchers conducted experiments (*i.e.,* 16.09%). They observed that even though the proportion of publications that include an evaluation is fairly constant over time, they lack rigor. Mattila *et al.* [MIK+16] included 83 papers published between 2010 and 2015 in their analysis. They also found that only a few researchers conducted experiments (*i.e.,* 13.25%), some performed case studies (*i.e.,* 22.89%), and the rest used other evaluation methods. In our investigation we cover a much larger body of literature (*i.e.,* 181 full papers) that spans up to 2017. We not only characterize the state-of-the-art in software visualization evaluation, but we also propose guidance to researchers in the field by detecting common pitfalls, and by elaborating on guidelines to conduct evaluation of software visualization approaches.

Other studies have opted to evaluate software visualization tools and have reported guidelines. For example, Storey *et al.* [SvG05] evaluated 12 software visualization tools, and proposed an evaluation framework based on intent, information, presentation, interaction, and effectiveness. Sensalire *et al.* [SOT08b, SOT09] evaluated 20 software visualization tools proposed for maintenance based via experiments, and elaborated various lessons learned. They identified a number of dimensions that are critical for organizing an evaluation, and then analyzing the results. Müller *et al.* [MKS+14] proposed a structured approach for conducting controlled experiments in envisioned 3D software visualization tools. Instead of concentrating on rather limited number of tools, we chose a meta analysis by analyzing the reports of the evaluation of proposed visualization tools. In this way we could analyze the state-of-the-art in the practice of software visualization evaluation, and consequently elaborate guidelines for defining what is an effective software visualization.

A few reviews of the software visualization literature that focus on various domains have tangentially analyzed the evaluation aspect. Lopez-Herrejon *et al.* [LHIE18] analyzed evaluation strategies used in visualizations proposed for software product line engineering, and they found

that most approaches used case studies. They also found that only a few performed experiments, and a few others did not explicitly describe an evaluation. Shahin *et al.* [SLB14] discussed the evaluation of visualization approaches proposed to support software architecture, and classified the evidence of the evaluation using a 5-step scale [ANAV10]. The analysis of the results showed that almost half of the evaluations represent toy examples or demonstrations. The other half correspond to industrial case studies, and a very few others described experiments and anecdotal evidence of tool adoption. Novais *et al.* [NTM+13] investigated the evaluations of approaches that proposed visualization to analyze software evolution. In most of the analyzed studies evaluation consisted in usage examples that were demonstrated by the authors of the study. In a few of them, the demonstration was carried out by external users. Evaluation strategies based on experiments were found to be extremely rare. In almost 20% of the studies they did not find an explicit evaluation. Since the main focus of these mentioned studies is not on evaluation (as opposed to ours), they only characterize the evaluation of the analyzed studies, and offer little advice for researchers who need to perform their own evaluations of software visualizations.

Similar efforts have been made in the information visualization field. Amar and Stasko [AS04] proposed a task-based framework for the evaluation of information visualizations. Forsell [For10] proposed a guide to scientific evaluation of information visualization that focuses on quantitative experimental research. The guide contains recommendations for *(a)* designing, *(b)* conducting, *(c)* analyzing results, and *(d)* reporting on experiments. Lam *et al.* [LBI+12] proposed seven scenarios for empirical studies in information visualization. Isenberg *et al.* [IIC+13] reviewed 581 papers to analyze the practice of evaluating visualization. Some of the pitfalls they found are that in some evaluations *(i)* participants do not belong to the target audience, *(ii)* goals are not explicit, *(iii)* the strategy and analysis method is not appropriate, and *(iv)* the level of rigor is low. Elmqvist and Yi [EY15] proposed patterns for visualization evaluation that present solutions to common problems encountered when evaluating a visualization system. We observed that advice given in the context of information visualization can also be applied to software visualization evaluation; however, we also observed that there are particularities in software visualization that require a tailored analysis, which is an objective of our investigation.

We consider previous research to be an important step to characterizing the evidence of the effectiveness of software visualization approaches. However, we reflect that previous research has failed to define what is an effective software visualization, and consequently a given conclusion that a particular visualization is effective cannot be compared with others. Moreover, we believe that some studies have used a loose definition of "case studies" and include many usage scenarios of visualization instead that present little evidence of the effectiveness of an approach. In our investigation we perform a subtler analysis of the characteristics of evaluations to elucidate these concerns. We think our study can help researchers in the field to identify common pitfalls when designing and conducting evaluations. Consequently, we formulated the following research questions:

*RQ.1*) What are the characteristics of evaluations that validate the effectiveness of software visualization approaches?

*RQ.2*) How appropriate are evaluations used to validate the effectiveness of software visualization?

We believe that answering these questions will assist researchers in the software visualization field to improve the quality of evaluations by identifying evaluation strategies and methods and their common pitfalls. In particular, we reviewed 181 full papers of the 387 papers published in SOFTVIS/VISSOFT. We identified evaluation strategies such as surveys, case studies, and experiments, as well as characteristics such as tasks, participants, and systems used in evaluations. We found that 62% (*i.e.,* 113) of the proposed software visualization approaches either do not include any evaluation, or include a weak evaluation (*i.e.,* anecdotal evidence, usage scenarios). Almost all of them (*i.e.,* 110) introduce a new software visualization approach. The remaining three discuss an existing approach but without providing a stronger evaluation. We also found that 29% of the studies (*i.e.,* 53) conducted experiments in which 30% (*i.e.,* 16) corresponded to visualizations that target the novice developer audience, and included appropriate participants. The remaining 70% proposed visualizations for developers with various levels of experience. However, amongst them only 30% included experienced developers, and the remaining 70% (*i.e.,* 37) included in experiments only students and academics of a convenience sample who are vulnerable to selection bias and hence hinder generalization. We found that 7% (*i.e.,* 12) of the studies conducted a case study that involved *(i)* professional developers from industry, and

*(ii)* real-world software systems. Finally, 3% (*i.e.,* 4) of studies conducted a survey. Even though we are not aware of a similar quantitative report of the state of affairs in information visualization, a review of the practice of evaluation [IIC+13] found similar issues.

We believe that for software visualization approaches to be adopted by developers, visualizations not only must prove their effectiveness via evaluations, but evaluations should also include participants of the target audience, and be based on real-world software systems. Finally, we recommend researchers in the field to conduct surveys that can help them to identify what are the frequent and complex problems that affect developers.

The remainder of the chapter is structured as follows: Section 3.2 describes the main concepts that are addressed in the characterization; Section 3.3 describes the methodology that we followed to collect and select relevant studies proposed in the software visualization field; Section 3.4 presents our results by classifying evaluations based on adopted strategies, methods and their characteristics; Section 3.5 discusses our research questions and threats to validity of our findings, and Section 3.6 concludes.

## 3.2 Background

The strategies that researchers adopt to evaluate the effectiveness of a software visualization approach can be classified into two main categories:

*(i)* *Theoretical* principles from information visualization that provide researchers support to justify a chosen visual encoding [Mun08]. For instance, the effectiveness of perceptual channels depends on the data type (*i.e.,* categorical, ordered, or quantitative) [Mac86].

*(ii)* *Empirical* evidence gathered from the evaluation of a technique, method or tool. Amongst them we find *a*) *exploratory* evaluations that involve high-level real-world tasks, for which identifying the aspects of the tool that boosted the effectiveness is complex; and *b*) *explanatory* evaluations in which high-level tasks are dissected into low-level (but less realistic) tasks that can be measured in isolation to identify the cause of an increase in the effectiveness of an approach [WRH+00].

Amongst the strategies used in empirical evaluations we find *(a) surveys* [WRH+12] that allow researchers to collect data from developers

who are the users of a system, and hence analyze the collected data to generalize conclusions; *(b) experiments* [SHH+05] that provide researchers with a high level of control to manipulate some variables while controlling others (*i.e.,* controlled experiments) with randomly assigned subjects (when it is not possible to ensure randomness the strategy is called "quasi-experiment"); and *(c) case studies* [RH09] that help researchers to investigate a phenomenon in its real-life context (*i.e.,* the case), hence giving researchers a lower level of control than an experiment but enabling a deeper analysis.

Several methods exist for collecting data in each evaluation strategy. The two most common methods [Fin03] are *(i) questionnaires* in which the researcher provides instructions to participants to answer a set of questions that can range from loosely structured (*e.g.,* exploratory survey) to closed and fully structured (*e.g.,* to collect data of the background of participants in an experiment), and *(ii) interviews* in which a researcher can ask a group of subjects a set of closed questions in a fixed order (*i.e.,* fully structured), a mix of open and closed questions (*i.e.,* semi-structured), and open-ended questions (*i.e.,* unstructured). Less frequent methods for collecting data are observational ones such as *(iii) think-aloud* in which researchers ask participants to verbalize their thoughts while performing the evaluation. Besides, recent experiments have collected data using *(iv) video recording* to capture the behavior of participants during the evaluation; *(v) sketch drawing* to evaluate recollection; and *(vi) eye tracking* to measure the browsing behavior of eye's movement.

Finally, there are several statistical tests that are usually used to analyze quantitative data collected from an experiment. For discrete or categorical data, tests such as *Chi-square* and *Cohen's kappa* are suitable. For questions that analyze the relationships of independent variables, *regression analysis* can be applied. For correlation analysis of dependent variables one has to first analyze if the parametric assumptions holds. That is, if the data is *(i)* collected from independent and unbiased samples, *(ii)* normally distributed (*Shapiro-Wilk test* is suggested and proven more powerful than *Kolmogorov-Smirnov* [RW+11]), and *(iii)* present equal variances (*e.g., Levene's test*, *Mauchly's test*). Parametric data can be analyzed with *Pearson's r*, while non-parametric with *Spearman's Rank Correlation*. For the analysis of differences of parametric data collected from two groups *Student's unpaired t-test*, *Paired t-test*, and *Hotelling's T-square* are appropriate. For the non-parametric case *Mann-Whitney U* and *Wilcoxon Rank sum test* are suitable. In the case of analysis that involves more

than two groups of parametric data *ANOVA* is a frequent choice, which is usually followed by a post-hoc test such as *Tukey HSD*. When data is non-parametric *Kruskal-Wallis test* and *Friedman test* are suitable as well.

## 3.3 Methodology

We build on the results presented in Chapter 2. However, we observe that amongst the 86 papers included in such analysis there are a number of them that do not correspond to full papers, and therefore, they are more likely to not describe an evaluation. Moreover, previously we only included design study papers, and excluded papers of other types, which would bias the analysis of evaluations.

We systematically review the evaluations of proposed software visualizations. We include not only design study papers, but also papers from other categories (*i.e.,* technique, system, evaluation). We also expand the collection of papers to include the ones published in VISSOFT 2017.

### 3.3.1 Inclusion and exclusion criteria

We reviewed the proceedings and programs of the venues to include full papers and exclude other types of papers that due to limited space are unlikely to contain enough detail. In particular, from the 387 papers we excluded 178 papers that corresponded to: (*i*) 61 poster. (*ii*) 52 new ideas and emerging results (NIER), (*iii*) 44 tool demo (TD), (*iv*) 8 keynote, (*v*) 8 position, and (*vi*) 5 challenge papers,

### 3.3.2 Quality assessment

We then assessed the quality of the remaining 209 papers. We classified the studies according to the categories proposed by Munzner [Mun08], in which a visualization paper can be classified into one of five categories:

*(a)* *Evaluations* describe how a visualization is used to deal with tasks in a problem domain. Evaluations are often conducted via user studies in laboratory settings in which participants solve a set of tasks while variables are measured.

*(b)* *Design studies* show how existing visualization techniques can be usefully combined to deal with a particular problem domain. Typically, design studies are evaluated through case studies and usage scenarios.

*(c)* *Systems* elaborate on the architectural design choices of a proposed visualization tool and the lessons learned from observing its use.

*(d)* *Techniques* focus on novel algorithms that improve the effectiveness of visualization.Techniques are often evaluated using benchmarks that measure performance.

*(e)* *Models* include *Commentary* papers in which an expert in the field advocate a position and argue to support it; *Formalism* papers present new models, definitions or terminology to describe techniques; and *Taxonomy* papers propose categories that help researchers to analyze the structure of a domain.

For each paper, we first read the abstract, second the conclusion, and finally, in the cases where we still were not sure of their main contribution, we read the rest of the paper. Although some papers might exhibit characteristics of more than one type, we classified them by focusing on their primary contribution.

We observe that model papers in which the main contribution is a commentary, a formalism or a taxonomy, usually do not describe explicit evaluations. Consequently, we excluded twenty-eight papers that we classified in those categories: *(i)* six commentary, *(ii)* seven taxonomy, and *(iii)* fifteen formalism papers.

Figure 3.1 provides an overview of the selection process. Figure 3.2 summarized the 387 collected papers and highlights the 181 included in the study. Figure 3.3 shows the outcome of our classification.

A frequent critique of visualization papers is a lack of evaluation. Indeed, papers in which the main contribution is an evaluation are unusual (*i.e.,* 10%). The chart also shows that the two main paper types in visualization are design study and technique.

The collection of 181 full papers includes studies from six to eleven pages in length. Initially, we were reluctant to include six-page papers, but we observed that in two editions of the conferences all full papers were of that length. Consequently, we analyzed the distribution of research strategies used to evaluate software visualization approaches by paper length. We did not find any particular trend, and so decided to include them.

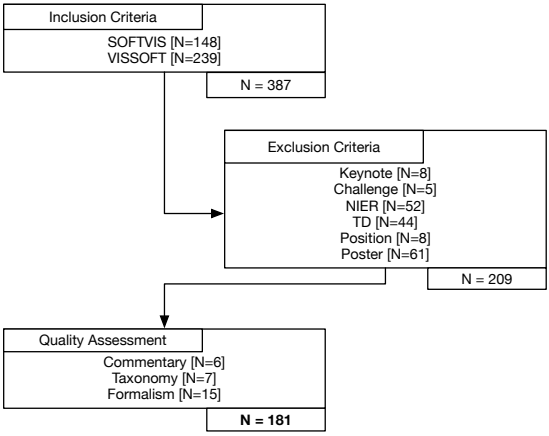



Figure 3.1: Stages of the search process, and the number of selected studies in each stage.

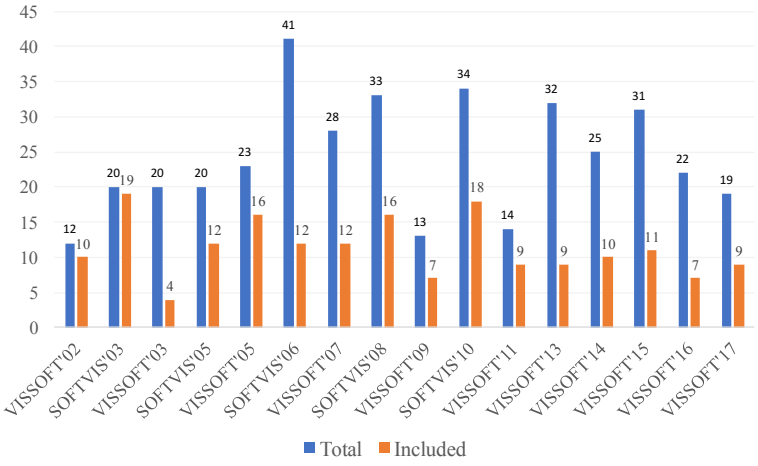



Figure 3.2: The 181 included papers from the collection of 387 papers published in SOFTVIS/VISSOFT venues.

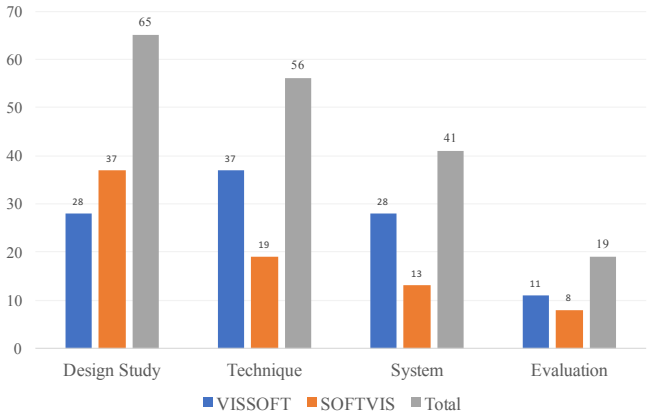



Figure 3.3: Classification of the 181 SOFTVIS/VISSOFT full papers by type.

### 3.3.3 Data extraction

To accelerate the process of finding and extracting the data from the studies, we collected keywords that authors commonly use to describe evaluations iteratively. That is, we started the process by searching for the following keywords in each paper: “evaluation”, “survey” “experiment”, “case study”, and “user study”. When we did not find these keywords, we manually inspected the paper and looked for other new representative keywords to expand our set. During the manual inspection when we did not find an explicit evaluation we labeled the papers accordingly. In the end, we collected the following set of keywords:

*{evaluation, survey, [case|user] stud[y|ies], [application | usage | analysis] example[s], use case[s], application scenario[s], [controlled | user] experiment, demonstration, user scenario[s], example of use, usage scenario[s], example scenario[s], demonstrative result[s]}*

We investigated whether evaluations that involve users are conducted with end users from the expected target audience (*i.e.,* representative sample) to ensure the generality of results. Therefore, in studies that used this type of evaluation, we extracted *who* conducted the evaluation, and *what*

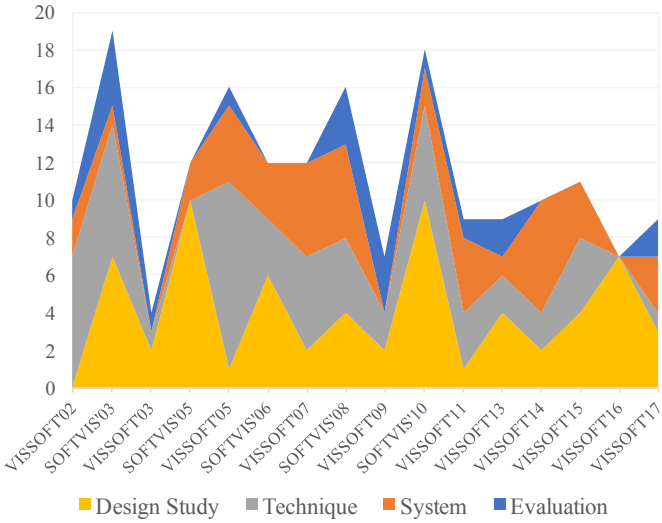



Figure 3.4: Evolution of SOFTVIS/VISSOFT papers by type (from the bottom upwards): Design Study, Technique, System, Evaluation.

subject systems were involved. We extracted these data by scanning the evaluation section of papers. In particular, we extracted *(i)* data collection methods (*e.g.,* think-aloud, interview, questionnaire); *(ii)* number of participants and their background, *(iii)* tasks, *(iv)* subject system, *(v)* dependent variables, and *(vi)* statistical tests.

### 3.3.4 Selected studies

We included in our study the 181 papers listed in Tables 3.1,3.2, and 3.3.

We present the evolution of the types of the included papers in Figure 3.4. We can observe that design study papers constitute the largest group, and evaluation papers are comparatively rare.

## 3.4 Results

We report the characteristics of the extracted data and the categories used to classify them for quantitative analysis. Figure 3.5 shows the distribution

Table 3.1: The papers included in the study [S1-S70].

| Id and Reference | Venue | Evaluation |
|---|---|---|
| [S1] Aesthetics of class diagrams, **Eichelberger, H.** | V'02 | Theorical |
| [S2] Specifying algorithm visualizations in terms of dat..., **Francik, J.** | V'02 | Usage Scenario |
| [S3] View definitions for language-independent multipl..., **Sajaniemi, J.** | V'02 | Usage Scenario |
| [S4] The CONCEPT project - applying source code analysis to..., **Rilling, J. *et al.*** | V'02 | - |
| [S5] UML collaboration diagram syntax: an empir..., **Purchase, H.C. *et al.*** | V'02 | Experiment |
| [S6] Runtime visualisation of object oriented soft..., **Smith, M.P. *et al.*** | V'02 | Usage Scenario |
| [S7] Reification of program points for visual execution , **Diehl, S. *et al.*** | V'02 | - |
| [S8] Metrics-based 3D visualization of large obj..., **Lewerentz, C. *et al.*** | V'02 | Usage Scenario |
| [S9] Analogical representations of programs, **Ploix, D.** | V'02 | Usage Scenario |
| [S10] Revision Towers, **Taylor, C.M.B. *et al.*** | V'02 | Usage Scenario |
| [S11] Self-Organizing Maps Applied in Visualising ..., **Brittle, J. *et al.*** | V'03 | Experiment |
| [S12] KScope: A Modularized Tool for 3D Visualizati..., **Davis, T.A. *et al.*** | V'03 | Theorical |
| [S13] Visualization to Support Version Control Software..., **Wu, X. *et al.*** | V'03 | Experiment |
| [S14] Techniques for Reducing the Complexity o..., **Hamou-Lhadj, A. *et al.*** | V'03 | Usage Scenario |
| [S15] A topology-shape-metrics approach for the automa..., **Eiglsperger, M. *et al.*** | S'03 | - |
| [S16] A new approach for visualizing UML class diagrams, **Gutwenger, C. *et al.*** | S'03 | - |
| [S17] Visualizing model mappings in UML, **Hausmann, J.H. *et al.*** | S'03 | - |
| [S18] Visualizing software for telecommunication services..., **Gansner, E.R. *et al.*** | S'03 | - |
| [S19] Graph visualization for the analysis of the structure an..., **Zhou, C. *et al.*** | S'03 | - |
| [S20] Interactive locality optimization on NUMA architectures, **Mu, T. *et al.*** | S'03 | - |
| [S21] End-user software visualizations for fault ..., **Ruthruff, J. *et al.*** | S'03 | Experiment |
| [S22] Interactive visual debugging with UML, **Jacobs, T. *et al.*** | S'03 | Usage Scenario |
| [S23] Designing effective program visualization too..., **Tudoreanu, M.E.** | S'03 | Experiment |
| [S24] Dancing hamsters and marble statue..., **Huebscher-Younger, T. *et al.*** | S'03 | Experiment |
| [S25] Algorithm visualization in CS education: com..., **Grissom, S. *et al.*** | S'03 | Experiment |
| [S26] A system for graph-based visualization of t..., **Collberg, C. *et al.*** | S'03 | Usage Scenario |
| [S27] Visualization of program-execution data for dep..., **Orso, A. *et al.*** | S'03 | Usage Scenario |
| [S28] Visualizing Java in action, **Reiss, S.P.** | S'03 | Usage Scenario |
| [S29] Plugging-in visualization: experiences integrating a ..., **Lintern, R. *et al.*** | S'03 | - |
| [S30] EVolve: an open extensible software visualizatio..., **Wang, Q. *et al.*** | S'03 | Usage Scenario |
| [S31] 3D representations for software visualization..., **Marcus, A. *et al.*** | S'03 | Usage Scenario |
| [S32] Growing squares: animated visualization of ..., **Elmqvist, N. *et al.*** | S'03 | Experiment |
| [S33] Program animation based on the roles of va..., **Sajaniemi, J. *et al.*** | S'03 | Experiment |
| [S34] Visualizing Feature Interaction in 3-D, **Greevy, O. *et al.*** | V'05 | Usage Scenario |
| [S35] Identifying Structural Features of Java Prog..., **Smith, M.P. *et al.*** | V'05 | Usage Scenario |
| [S36] Support for Static Concept Location with sv3D, **Xie, X. *et al.*** | V'05 | Usage Scenario |
| [S37] Interactive Exploration of Semantic Clusters, **Lungu, M. *et al.*** | V'05 | Usage Scenario |
| [S38] Exploring Relations within Software Systems ..., **Balzer, M. *et al.*** | V'05 | Usage Scenario |
| [S39] The Dominance Tree in Visualizing Software Dep..., **Falke, R. *et al.*** | V'05 | Usage Scenario |
| [S40] User Perspectives on a Visual Aid to Program Com..., **Cox, A. *et al.*** | V'05 | Experiment |
| [S41] Interactive Visual Mechanisms for Exploring So..., **Telea, A. *et al.*** | V'05 | Usage Scenario |
| [S42] Fractal Figures: Visualizing Development Ef..., **D'Ambros, M. *et al.*** | V'05 | Usage Scenario |
| [S43] White Coats: Web-Visualization of Evolving S..., **Mesnage, C. *et al.*** | V'05 | Usage Scenario |
| [S44] Multi-level Method Understanding Using Microprints , **Ducasse, S. *et al.*** | V'05 | - |
| [S45] Visual Realism for the Visualization of Softwa..., **Holten, D. *et al.*** | V'05 | Usage Scenario |
| [S46] Visual Exploration of Combined Architectural and Met..., **Termeer, M. *et al.*** | V'05 | - |
| [S47] Evaluating UML Class Diagram Layout base..., **Andriyevska, O. *et al.*** | V'05 | Experiment |
| [S48] Interactive Exploration of UML Sequence Diagra..., **Sharp, R. *et al.*** | V'05 | Usage Scenario |
| [S49] SAB - The Software Architecture Browser, **Erben, N. *et al.*** | V'05 | - |
| [S50] Towards understanding programs through wear-b..., **DeLine, R. *et al.*** | S'05 | Experiment |
| [S51] Online-configuration of software visualizations with Vi..., **Panas, T. *et al.*** | S'05 | - |
| [S52] Visualization of mobile object environments..., **Frishman, Y. *et al.*** | S'05 | Case Study |
| [S53] Visualizing structural properties of irregular par..., **Blochinger, W. *et al.*** | S'05 | - |
| [S54] Jove: java as it happens, **Reiss, S.P. *et al.*** | S'05 | - |
| [S55] Methodology and architecture of JIVE, **Gestwicki, P. *et al.*** | S'05 | Anecdotal |
| [S56] Visual specification and analysis of use cas..., **Kholkar, D. *et al.*** | S'05 | Case Study |
| [S57] Visualizing multiple evolution metrics, **Pinzger, M. *et al.*** | S'05 | Usage Scenario |
| [S58] The war room command console: shared visual..., **O'Reilly, C. *et al.*** | S'05 | Case Study |
| [S59] CVSscan: visualization of code evolution, **Voinea, L. *et al.*** | S'05 | Case Study |
| [S60] Visual data mining in software archives, **Burch, M. *et al.*** | S'05 | Usage Scenario |
| [S61] Algorithm visualization using concept keyboa..., **Baloian, N. *et al.*** | S'05 | Experiment |
| [S62] Mondrian: an agile information visualization f..., **Meyer, M. *et al.*** | S'06 | Usage Scenario |
| [S63] Multiscale and multivariate visualizations of ..., **Voinea, L. *et al.*** | S'06 | Usage Scenario |
| [S64] Visualization of areas of interest in softwar..., **Byelas, H. *et al.*** | S'06 | Case Study |
| [S65] Visual exploration of function call graphs for feature..., **Bohnet, J. *et al.*** | S'06 | - |
| [S66] Using social agents to visualize software..., **Alspaugh, T.A. *et al.*** | S'06 | Experiment |
| [S67] Transparency, holophrasting, and automatic layout appl..., **Gauvin, S. *et al.*** | S'06 | - |
| [S68] A data-driven graphical toolkit for softwa..., **Demetrescu, C. *et al.*** | S'06 | Usage Scenario |
| [S69] Visualizing live software systems in 3D, **Greevy, O. *et al.*** | S'06 | Usage Scenario |
| [S70] Execution patterns for visualizing web servic..., **de Pauw, W. *et al.*** | S'06 | Anecdotal |

Table 3.2: The papers included in the study [S71-S146].

| Id and Reference | Venue | Evaluation |
|---|---|---|
| [S71] Experimental evaluation of animated-verifying o..., **Jain, J. *et al.*** | S'06 | Experiment |
| [S72] Narrative algorithm visualization, **Blumenkrants, M. *et al.*** | S'06 | Experiment |
| [S73] The Clack graphical router: visualizing net..., **Wendlandt, D. *et al.*** | S'06 | Anecdotal |
| [S74] A Visualization for Software Project Awaren..., **Ripley, R.M. *et al.*** | V'07 | Usage Scenario |
| [S75] YARN: Animating Software Evolution, **Hindle, A. *et al.*** | V'07 | Usage Scenario |
| [S76] DiffArchViz: A Tool to Visualize Correspondence ..., **Sawant, A.P.** | V'07 | Usage Scenario |
| [S77] A Bug's Life" Visualizing a Bug Database""A..., **D'Ambros, M. *et al.*** | V'07 | Usage Scenario |
| [S78] Task-specific source code dependency investig..., **Holmes, R. *et al.*** | V'07 | Experiment |
| [S79] Visualizing Software Systems as Cities, **Wettel, R. *et al.*** | V'07 | - |
| [S80] Onion Graphs for Focus+Context Views of UML Cl..., **Kagdi, H. *et al.*** | V'07 | Usage Scenario |
| [S81] CocoViz: Towards Cognitive Software Visuali..., **Boccuzzo, S. *et al.*** | V'07 | Usage Scenario |
| [S82] Distributable Features View: Visualizing the..., **Cosma, D.C. *et al.*** | V'07 | Usage Scenario |
| [S83] Trace Visualization Using Hierarchical Edge B..., **Holten, D. *et al.*** | V'07 | Usage Scenario |
| [S84] Visualization of Dynamic Program Aspects, **Deelen, P. *et al.*** | V'07 | Usage Scenario |
| [S85] Visualizing Dynamic Memory Allocations, **Moreta, S. *et al.*** | V'07 | Usage Scenario |
| [S86] Applying visualisation techniques in software..., **Nestor, D. *et al.*** | S'08 | Usage Scenario |
| [S87] Stacked-widget visualization of scheduling-..., **Bernardin, T. *et al.*** | S'08 | Usage Scenario |
| [S88] Visually localizing design problems with dish..., **Wettel, R. *et al.*** | S'08 | Usage Scenario |
| [S89] Visualizing inter-dependencies between scenarios, **Harel, D. *et al.*** | S'08 | - |
| [S90] Software visualization for end-user pr..., **Subrahmaniyan, N. *et al.*** | S'08 | Case Study |
| [S91] Streamsight: a visualization tool for large-s..., **de Pauw, W. *et al.*** | S'08 | Anecdotal |
| [S92] Improving an interactive visualization of transition ..., **Ploeger, B. *et al.*** | S'08 | - |
| [S93] Automatic layout of UML use case diagrams, **Eichelberger, H.** | S'08 | - |
| [S94] Gef3D: a framework for two-, two-and-a-h..., **von Pilgrim, J. *et al.*** | S'08 | Usage Scenario |
| [S95] A catalogue of lightweight visualizations to ..., **Parnin, C. *et al.*** | S'08 | Usage Scenario |
| [S96] An interactive reverse engineering environment..., **Telea, A. *et al.*** | S'08 | Experiment |
| [S97] Representing unit test data for large scale ..., **Cottam, J.A. *et al.*** | S'08 | Anecdotal |
| [S98] HDPV: interactive, faithful, in-vivo run..., **Sundararaman, J. *et al.*** | S'08 | Usage Scenario |
| [S99] Analyzing the reliability of communication be..., **Zeckzer, D. *et al.*** | S'08 | Usage Scenario |
| [S100] Visualization of exception handling constructs..., **Shah, H. *et al.*** | S'08 | Experiment |
| [S101] Assessing the benefits of synchronization-adorn..., **Xie, S. *et al.*** | S'08 | Experiment |
| [S102] Extraction and visualization of call dependen..., **Telea, A. *et al.*** | V'09 | Usage Scenario |
| [S103] Visualizing the Java heap to detect memory proble..., **Reiss, S.P.** | V'09 | Anecdotal |
| [S104] Case study: Visual analytics in software prod..., **Telea, A. *et al.*** | V'09 | Usage Scenario |
| [S105] Visualizing massively pruned execution trace..., **Bohnet, J. *et al.*** | V'09 | Case Study |
| [S106] Evaluation of software visualization tool..., **Sensalire, M. *et al.*** | V'09 | Experiment |
| [S107] The effect of layout on the comprehension of..., **Sharif, B. *et al.*** | V'09 | Experiment |
| [S108] Beyond pretty pictures: Examining the benef..., **Yunrim Park *et al.*** | V'09 | Experiment |
| [S109] Representing development history in s..., **Steinbrueckner, F. *et al.*** | S'10 | Usage Scenario |
| [S110] Visual comparison of software architectures, **Beck, F. *et al.*** | S'10 | Usage Scenario |
| [S111] An automatic layout algorithm for BPEL processes, **Albrecht, B. *et al.*** | S'10 | - |
| [S112] Off-screen visualization techniques for clas..., **Frisch, M. *et al.*** | S'10 | Experiment |
| [S113] Jype - a program visualization and programm..., **Helminen, J. *et al.*** | S'10 | Survey |
| [S114] Zinsight: a visual and analytic environment..., **de Pauw, W. *et al.*** | S'10 | Case Study |
| [S115] Understanding complex multithreaded softwa..., **Truemper, J. *et al.*** | S'10 | Case Study |
| [S116] Visualizing windows system traces, **Wu, Y. *et al.*** | S'10 | Usage Scenario |
| [S117] Embedding spatial software visualization in th..., **Kuhn, A. *et al.*** | S'10 | Experiment |
| [S118] Towards anomaly comprehension: using structural..., **Lin, S. *et al.*** | S'10 | Experiment |
| [S119] Dependence cluster visualization, **Islam, S.S. *et al.*** | S'10 | Usage Scenario |
| [S120] Exploring the inventor's paradox: applying jig..., **Ruan, H. *et al.*** | S'10 | Usage Scenario |
| [S121] Trevis: a context tree visualization & anal..., **Adamoli, A. *et al.*** | S'10 | Usage Scenario |
| [S122] Heapviz: interactive heap visualizati..., **Aftandilian, E.E. *et al.*** | S'10 | Usage Scenario |
| [S123] AllocRay: memory allocation visualizati..., **Robertson, G.G. *et al.*** | S'10 | Experiment |
| [S124] Software evolution storylines, **Ogawa, M. *et al.*** | S'10 | - |
| [S125] User evaluation of polymetric views using a ..., **Anslow, C. *et al.*** | S'10 | Experiment |
| [S126] An interactive ambient visualization fo..., **Murphy-Hill, E. *et al.*** | S'10 | Experiment |
| [S127] Follow that sketch: Lifecycles of diagrams an..., **Walny, J. *et al.*** | V'11 | Experiment |
| [S128] Visual support for porting large code base..., **Broeksema, B. *et al.*** | V'11 | Usage Scenario |
| [S129] A visual analysis and design tool for planning..., **Beck, M. *et al.*** | V'11 | Case Study |
| [S130] Visually exploring multi-dimensional code coup..., **Beck, F. *et al.*** | V'11 | Usage Scenario |
| [S131] Constellation visualization: Augmenting progra..., **Deng, F. *et al.*** | V'11 | Experiment |
| [S132] 3D Hierarchical Edge bundles to visualize relations ..., **Caserta, P. *et al.*** | V'11 | - |
| [S133] Abstract visualization of runtime m..., **Choudhury, A.N.M.I. *et al.*** | V'11 | Usage Scenario |
| [S134] Telling stories about GNOME with Complicity, **Neu, S. *et al.*** | V'11 | Usage Scenario |
| [S135] E-Quality: A graph based object oriented so..., **Erdemir, U. *et al.*** | V'11 | Experiment |
| [S136] Automatic categorization and visualization o..., **Reiss, S.P. *et al.*** | V'13 | Usage Scenario |
| [S137] Using HTML5 visualizations in software faul..., **Gouveia, C. *et al.*** | V'13 | Experiment |
| [S138] Visualizing jobs with shared resources in di..., **de Pauw, W. *et al.*** | V'13 | Usage Scenario |
| [S139] SYNCTRACE: Visual thread-interplay analysis, **Karran, B. *et al.*** | V'13 | Usage Scenario |
| [S140] Finding structures in multi-type code c..., **Abuthawabeh, A. *et al.*** | V'13 | Experiment |

Table 3.3: The papers included in the study [S141-S181].

| Id and Reference | Venue | Evaluation |
|---|---|---|
| [S141] SourceVis: Collaborative software visualizat..., **Anslow, C. *et al.*** | V'13 | Experiment |
| [S142] Visualizing software dynamicities with heat..., **Benomar, O. *et al.*** | V'13 | Usage Scenario |
| [S143] Performance evolution blueprint: Underst..., **Sandoval, J.P. *et al.*** | V'13 | Usage Scenario |
| [S144] An empirical study assessing the effect of s..., **Sharif, B. *et al.*** | V'13 | Experiment |
| [S145] Visualizing Developer Interactions, **Minelli, R. *et al.*** | V'14 | Usage Scenario |
| [S146] AniMatrix: A Matrix-Based Visualization of ..., **Rufiange, S. *et al.*** | V'14 | Usage Scenario |
| [S147] Visualizing the Evolution of Systems and The..., **Kula, R.G. *et al.*** | V'14 | Usage Scenario |
| [S148] ChronoTwigger: A Visual Analytics Tool for Unde..., **Ens, B. *et al.*** | V'14 | Experiment |
| [S149] Lightweight Structured Visualization of Asse..., **Toprak, S. *et al.*** | V'14 | Experiment |
| [S150] How Developers Visualize Compiler Messages: A..., **Barik, T. *et al.*** | V'14 | Experiment |
| [S151] Feature Relations Graphs: A Visualisation ..., **Martinez, J. *et al.*** | V'14 | Case Study |
| [S152] Search Space Pruning Constraints Visualizati..., **Haugen, B. *et al.*** | V'14 | Usage Scenario |
| [S153] Integrating Anomaly Diagnosis Techniques int..., **Kulesz, D. *et al.*** | V'14 | Experiment |
| [S154] Combining Tiled and Textual Views of Code, **Homer, M. *et al.*** | V'14 | Experiment |
| [S155] Visualizing Work Processes in Software Engine..., **Burch, M. *et al.*** | V'15 | Usage Scenario |
| [S156] Blended, Not Stirred: Multi-concern Visua..., **Dal Sasso, T. *et al.*** | V'15 | Usage Scenario |
| [S157] CodeSurveyor: Mapping Large-Scale Software to..., **Hawes, N. *et al.*** | V'15 | Experiment |
| [S158] Revealing Runtime Features and Constituent..., **Palepu, V.K. *et al.*** | V'15 | Usage Scenario |
| [S159] A Visual Support for Decomposing Complex Featu..., **Urli, S. *et al.*** | V'15 | Usage Scenario |
| [S160] Visualising Software as a Particle System, **Scarle, S. *et al.*** | V'15 | Usage Scenario |
| [S161] Interactive Tag Cloud Visualization of Sof..., **Greene, G.J. *et al.*** | V'15 | Usage Scenario |
| [S162] Hierarchical Software Landscape Visualizati..., **Fittkau, F. *et al.*** | V'15 | Experiment |
| [S163] Vestige: A Visualization Framework for Eng..., **Schneider, T. *et al.*** | V'15 | Usage Scenario |
| [S164] Visual Analytics of Software Structure and Met..., **Khan, T. *et al.*** | V'15 | Experiment |
| [S165] Stable Voronoi-Based Visualizations for Sof..., **Van Hees, R. *et al.*** | V'15 | Usage Scenario |
| [S166] Visualizing the Evolution of Working Sets, **Minelli, R. *et al.*** | V'16 | Experiment |
| [S167] Walls, Pillars and Beams: A 3D Decompositio..., **Tymchuk, Y. *et al.*** | V'16 | Case Study |
| [S168] CuboidMatrix: Exploring Dynamic Structura..., **Schneider, T. *et al.*** | V'16 | Experiment |
| [S169] A Tool for Visualizing Patterns of Spread..., **Middleton, J. *et al.*** | V'16 | Experiment |
| [S170] Jsvee & Kelmu: Creating and Tailoring Program Ani..., **Sirkiae, T.** | V'16 | Usage Scenario |
| [S171] Visualizing Project Evolution through Abstr..., **Feist, M.D. *et al.*** | V'16 | Usage Scenario |
| [S172] Merge-Tree: Visualizing the Integration of Com..., **Wilde, E. *et al.*** | V'16 | Usage Scenario |
| [S173] A Scalable Visualization for Dynamic Data in ..., **Burch, M. *et al.*** | V'17 | Experiment |
| [S174] An Empirical Study on the Readability of R..., **Hollmann, N. *et al.*** | V'17 | Experiment |
| [S175] Concept-Driven Generation of Intuitive Explana..., **Reza, M. *et al.*** | V'17 | Usage Scenario |
| [S176] Visual Exploration of Memory Traces and Call ..., **Gralka, P. *et al.*** | V'17 | Usage Scenario |
| [S177] Code Park: A New 3D Code Visualization Tool..., **Khaloo, P. *et al.*** | V'17 | Experiment |
| [S178] Using High-Rising Cities to Visualize Perform..., **Ogami, K. *et al.*** | V'17 | Usage Scenario |
| [S179] iTraceVis: Visualizing Eye Movement Data With..., **Clark, B. *et al.*** | V'17 | Experiment |
| [S180] On the Impact of the Medium in the Effective..., **Merino, L. *et al.*** | V'17 | Experiment |
| [S181] Method Execution Reports: Generating Text and ..., **Beck, F. *et al.*** | V'17 | Experiment |

of the studies categorized by paper type [Mun08] and research strategy used to evaluate visualizations. Table 3.4 presents our classification of the evaluation strategy adopted by papers into one of three main categories: *(i) theoretical*, *(ii) no explicit evaluation*, and *(iii) empirical*. For evaluations that used an empirical strategy, we classified them into one of five categories: *(i) anecdotal evidence*, *(ii) usage scenarios*, *(iii) survey*, *(iv) case study*, and *(v) experiment*.

We report on characteristics of experiments such as data collection methods, type of analysis, visual tasks, dependent variables, statistical tests, and scope. The complete classification of the 181 included studies is displayed in Tables 3.5, 3.6, 3.7, 3.8, 3.9, and 3.10.

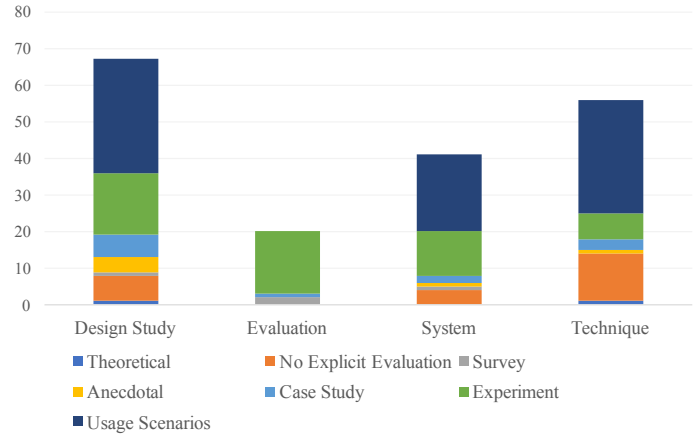



Figure 3.5: The distribution of the 181 included papers categorized by paper types and research strategy used to evaluate software visualization approaches.

### 3.4.1 Data Collection Methods

In Table 3.5 we list the various methods that researchers used to collect data from experiments. The most frequent were questionnaires, which are normally used to collect data of the background of participants at the beginning of experiments and final observations at the end. Questionnaires are found across all types of evaluation strategies (*i.e.,* survey, experiment, case study). Interviews are fairly frequent and found mostly in case studies. We also found traditional observational methods (*e.g.,* think-aloud), but also fairly new methods (*e.g.,* eye tracking).

### 3.4.2 Evaluation Strategies

In twenty-four (*i.e.,* 13%) studies we did not find an explicit evaluation that presents evidence for supporting the claim of effectiveness of software visualization approaches. These studies indicate that the evaluation of the proposed visualization is planned as future work. In the remaining studies, we found that several strategies were used to evaluate software visualization approaches. We observed that only two studies (*i.e.,* 1%) used *theoretical* references to support the claim of the effectiveness of software

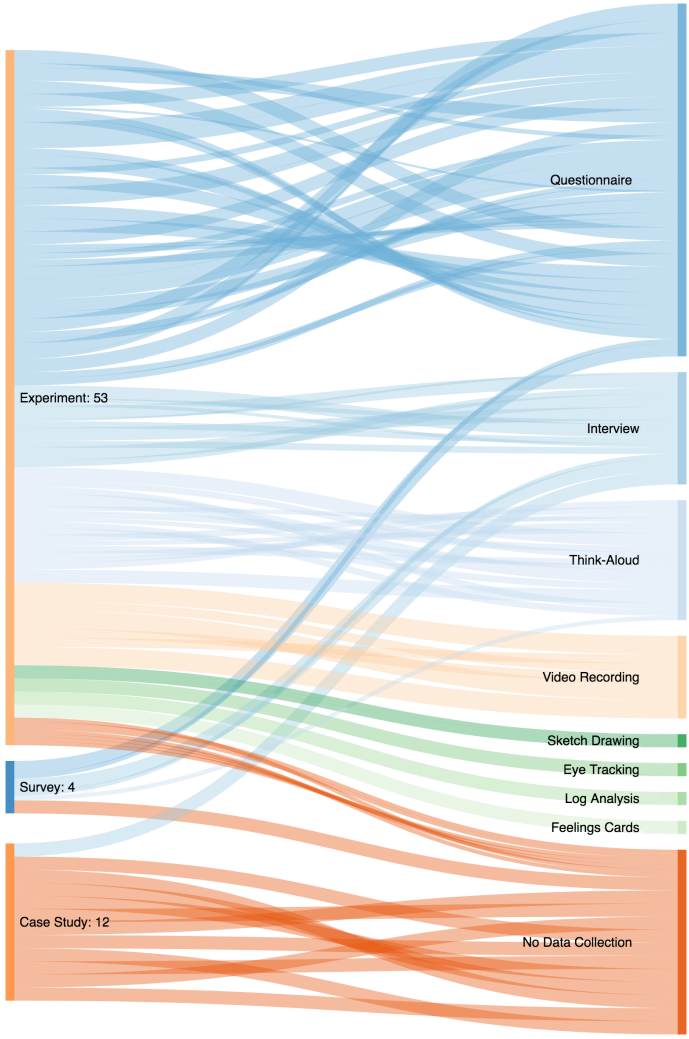



Figure 3.6: Sankey diagram showing the data collection methods (right) employed in evaluation strategies (left) adopted in empirical evaluations.

Table 3.4: Research strategies used to evaluate software visualization approaches.

| Category | Strategy | Reference | # |
|---|---|---|---|
| Theoretical | | S1, S12 | 2 |
| No Explicit Evaluation | | S4, S7, S15, S16, S17, S18, S19, S20, S29, S44, S46, S49, S51, S53, S54, S65, S67, S79, S89, S92, S93, S111, S124, S132 | 24 |
| Empirical | Survey | S13, S71, S100, S113 | 4 |
| | Anecdotal Evidence | S55, S70, S73, S91, S97, S103 | 6 |
| | Case Study | S52, S56, S58, S59, S64, S90, S105, S114, S115, S129, S151, S167 | 12 |
| | Experiment | S5, S11, S13, S21, S23, S24, S25, S32, S33, S40, S47, S50, S61, S66, S71, S72, S78, S96, S100, S101, S106, S107, S108, S112, S117, S118, S123, S125, S126, S127, S131, S135, S137, S140, S141, S144, S148, S149, S150, S153, S154, S157, S162, S164, S166, S168, S169, S173, S174, S177, S179, S180, S181 | 53 |
| | Example | S57, S60, S62, S63, S68, S69, S74, S75, S76, S77, S80, S81, S82, S83, S84, S85, S86, S87, S88, S94, S95, S98, S99, S102, S104, S109, S110, S116, S119, S120, S121, S122, S128, S130, S133, S134, S136, S138, S139, S142, S143, S145, S146, S147, S152, S155, S156, S158, S159, S160, S161, S163, S165, S170, S171, S172, S175, S176, S178 | 83 |

visualizations. One technique paper *[S1]* that proposes aesthetic criteria for class diagrams, considered their proposed criteria effective since it was derived from the UML specification, and one design study paper *[S12]*

Table 3.5: Data collection methods used to evaluate software visualization approaches.

| Method | Reference | # |
|---|---|---|
| Questionnaire | S11, S13, S25, S32, S40, S47, S50, S61, S66, S72, S90, S100, S106, S107, S108, S112, S125, S126, S127, S135, S137, S140, S141, S144, S149, S150, S153, S154, S157, S162, S164, S168, S173, S177, S179, S180, S181 | 37 |
| Think-Aloud | S40, S50, S100, S112, S117, S118, S123, S125, S126, S135, S141, S148, S150, S169, S173, S179, S180 | 17 |
| Interview | S33, S71, S78, S90, S100, S106, S123, S127, S153, S174, S177, S180 | 12 |
| Video Recording | S33, S50, S117, S125, S127, S140, S141, S144, S180 | 9 |
| Sketch Drawing | S117, S127, S180 | 3 |
| Others | Eye Tracking (S144), Log Analysis (S166), Feelings Cards (S180) | 3 |

evaluated the visualization based on previously proposed criteria for visualizing software in virtual reality [YM98]. Both studies planned as future work to conduct an experimental evaluation. The remaining 155 studies (*i.e.,* 86%) adopted an *empirical* strategy to evaluate software visualization approaches. Amongst them, we found that multiple strategies were used. We investigated the evidence of the effectiveness of visualization approaches provided by those strategies.

Figure 3.6 shows the relation between the data collection methods used in evaluation strategies. We observe that most case studies do not describe the methods used to collect data; however, we presume they are observational ones, such as one *[S90]* that reported to have conducted interviews. The few surveys in the analysis collected data using interviews and questionnaires. One survey *[S113]* did not describe the method to collect data. Experiments use multiple methods to collect data. They mainly use questionnaires, interviews, and the think-aloud protocol. Recent experiments

have used video recording, and other methods such as sketch drawing, eye tracking, log analysis, and emotion cards.

### Anecdotal Evidence

We found six studies (*i.e.,* 3%) that support the claim of effectiveness of visualizations on anecdotal evidence of tool adoption. Two papers *[S55,S73]* proposed a visualization to support the student audience and reported that tools were successfully used in software engineering courses. The remaining four studies *[S70,S91,S97,S103]* that focused on the developer audience reported that visualizations were used intensively and obtained positive feedback.

### Usage Scenarios

Eighty-three studies (*i.e.,* 46%) evaluated software visualizations via usage scenarios. In this type of evaluation, authors posed envisioned scenarios and elaborated on how the visualization was expected to be used. Usually, they selected an open-source software system as the subject of the visualization. The most popular systems that we found were written in *(i)* Java, such as *ArgoUML* (4×), *Ant* (4×), *JHotDraw* (3×), *Java SDK* (2×), and *Weka* (2×); *(ii)* C++, such as *Mozilla* (7×), *VTK* (2×), and *GNOME* (2×); and, *(iii)* Smalltalk *Pharo* (4×). We found that several names were used among the studies to describe this strategy. We observed that sixty-seven studies (*i.e.,* 37%) labeled evaluations as case studies, while twenty-six (*i.e.,* 14%) presented them as use cases. In the rest of the cases, authors used titles such as: "application examples", "usage examples", "application scenarios", "analysis example", "example of use", "usage scenarios", "application scenarios", and "usage example".

### Survey

Only four studies (*i.e.,* 2%) performed a survey, which is consistent with the findings of related work [MIK+16, SBCS14]. Three of them *[S13,S71,S100]* surveyed developers to identify complex problems and collect requirements to design a proposed visualization approach: one focused on supporting development teams who use version control systems *[S13]*, another asked former students of a course what they considered the most difficult subject in the lecture *[S71]*, and another was concerned with understanding exception-handling constructs *[S100]*. In one study

*[S113]* students who used a visualization approach were surveyed to collect anecdotal evidence of its usefulness. Two surveys *[S71,S113]* were conducted for visualization approaches that target the student audience in a software engineering course, while the remaining two *[S13,S100]* target the developer audience.

We found that surveys are used to identify frequent and complex problems that affect developers; such problems are then interpreted as requirements for a new visualization approach. We conjecture whether the low number of surveys has an effect on the disconnect between the proposed software visualization approaches and the needs of developers that we found in Chapter 2.

### Case Study

We classified twelve papers (*i.e.,* 7%) in the case study category. Usually, case studies are conducted to evaluate visualization approaches that target professional developers working on real-world projects in an industrial setting. The *case* of the study describes the context of the project in which difficulties arise, and shows how a visualization approach provides developers support for tackling them. We observed that in three studies *[S56,S90,S114]* some or all authors of the study come from industry, while in the rest there seems to be a strong relation of authors with industrial companies. In all of them, the evaluation involved professional developers.

### Experiment

Fifty-three studies (*i.e.,* 29%) evaluated software visualization via experiments. Although the level of detail varies, we identified a number of characteristics such as *(i) data collection methods*; *(ii) type of analysis*; *(iii) participants*; *(iv) tasks*; *(v) dependent variables*; and *(vi) statistical tests*. In the following we describe the results of the extracted data.

*(i) Participants.* We observed a high variance in the participants in experiments (shown in Figure 3.7). The highest number of participants is found in a study *[S25]* that included 157 students. The minimum number corresponds to a study *[S100]* that involved three participants (graduate students with experience in industry). The median was 13 participants. A similar analysis of participants in the evaluation of information visualization approaches [IIC+13] shows similar results. Most evaluations of information visualization approaches

involve 1–5 participants (excluding evaluations that do not report on the number of participants). The second most popular group includes 11–20 participants, and the group that includes 6–10 is the third most popular. Overall the median is 9 participants. Although many evaluations in software visualization included a number of participants in that ranges, the most popular ones are 6–10 and 11–20, followed by 21–30. One reason that might explain the difference could be that in our analysis we only included full papers that might present more thorough evaluations including a higher number of participants.

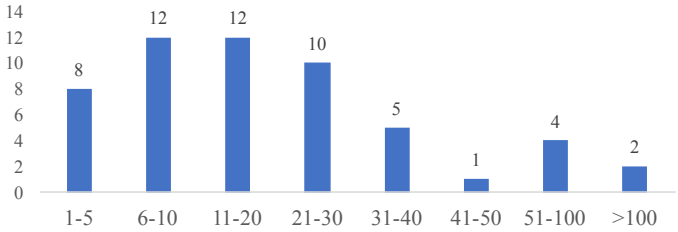



Figure 3.7: Histogram of the number of participants reported in evaluation.

We noticed that experiments to evaluate software visualization approaches for teaching software engineering (*e.g.,* algorithms and data structures) include a high number of participants since they usually involve a whole course and sometimes several of them. This type of experiment typically evaluates the effect of introducing visualization tools as a means for helping students to learn the subject of the course. All of them found that visualizations do help students. However, they do not provide insights into whether the particular visualization technique tested in the experiment is the most suitable one. All experiments include participants selected from a convenience sample. Normally, they are students and academics at various levels with little experience working in industry.

*(ii) Type of Analysis.* Table 3.6 presents our classification of the type of analysis adopted in experiments. We categorized the type of analysis into one of two categories: *quantitative* and *qualitative*. We found thirteen studies that adopted a quantitative analysis, while twenty-two used a qualitative one. In eighteen studies there was both a quantitative and qualitative analysis. Common examples of quantitative

analyses in experiments include the measure of quantitative variables such as time and correctness

Table 3.6: Type of analysis adopted in experiments.

| Type of Analysis | References | # |
|---|---|---|
| Quantitative | S21, S23, S24, S25, S71, S78, S101, S107, S137, S150, S154, S164, S174 | 13 |
| Qualitative | S11, S13, S33, S61, S66, S96, S100, S106, S112, S117, S123, S127, S135, S140, S141, S148, S149, S153, S157, S166, S169, S181 | 22 |
| Quantitative / Qualitative | S5, S32, S40, S47, S50, S72, S108, S118, S125, S126, S131, S144, S162, S168, S173, S177, S179, S180 | 18 |

*(iii) Task.* In table 3.10 the column *Task* summarizes exemplary tasks that we extracted from the design of each experiment. In almost half of the experiments (*i.e.,* 26) we found explicit tasks that we identify with a check mark ✓. The remaining tasks that we list correspond to rationales that we inferred from analyzing the goals of experiments.

Typically, experiments were described as being formative or exploratory, and adopted a qualitative analysis of results (*i.e.,* 75%). Several experiments also used a quantitative analysis to report evidence that supports the effectiveness of software visualization approaches. Although reporting on early results of preliminary evaluations has contributed important knowledge to the software visualization field, we believe that for software visualization approaches to become an actionable choice for developers, they have to present sound evidence of their effectiveness via surveys, controlled experiments, and case studies.

*(iv) Dependent Variables.* Table 3.8 lists the dependent variables that were measured in experiments. We adopted the classification proposed by Lam *et al.* [LBI+12] and classified the dependent variables based on two of the proposed scenarios for evaluation of the understanding of visualizations: *user performance* and *user experience*. We found

35 (*i.e.,* 66%) studies that evaluated user performance, 8 (*i.e.,* 15%) evaluated user experience, and 10 (*i.e.,* 19%) that evaluated variables of both. To evaluate performance most experiments defined as dependent variables *correctness* and *time*, some others specified that the experiment aimed at evaluating effectiveness without presenting details, and a few described multiple variables such as recollection, visual effort, scalability, and efficiency. To evaluate user experience researchers asked participants their perception of various variables such as usability, engagement, understandability, and emotions.

*(v) Statistical Tests.* Table 3.9 summarizes the statistical tests used in experiments for the quantitative analysis of data. We observed that the choice of the test is governed primarily by the number of dependent variables, their treatment and the type of the collected data (*i.e.,* categorical, ordinal, interval). For instance, a questionnaire that uses a 5-step Likert scale to ask participants how suitable they find particular characteristics of a software visualization approach for a certain task would be ordinal. In that case, there would be one dependent variable, with five levels of ordinal data, for which the Kruskal-Wallis test would be a suitable match. Also, ANOVA is a common choice to test hypotheses. However, we observed that in some cases researchers found that parametric assumptions do not hold. Although there are alternative tests for non-parametric data, we observe that for data that do not follow a normal distribution, they can perform an *Aligned Rank Transform* [WFGH11] *[S177].*

*(vi) Task.* In table 3.10 the column *Task* summarizes exemplary tasks that we extracted from the design of each experiment. In almost half of the experiments (*i.e.,* 26) we found explicit tasks that we identify with a check mark ✓. The remaining tasks that we list correspond to rationales that we inferred from analyzing the goals of experiments.

We observed that in several studies participants were asked to use a visualization to *lookup* some aspects of the system. Although in some cases a database query might be a more effective tool than a visualization, we observed that these tasks are often used as a stepping stone towards complex tasks, in which developers certainly benefit from visualizing the context. For instance, participants used a visualization to answer questions where they had to:

*a*) count elements such as *"how many packages are in the Java API?" [S125]*, *"what is the number of packages?" [S164]*, *"determine the total number of packages this system has" [S180]*, *"how many methods does the largest class have (in terms of LOC)?" [S144]*, and

*b*) find outliers such as *"find the process with the longest duration." [S32]*, *"who are the top three most active code contributors?" [S108]*, *"what are the two largest classes?" [S141]*, *"name three applications that have a high fan-in" [S162]*, *"find the three classes with the highest NOA" [S180].*

We also observe that most studies build on these answers and ask participants to complete tasks that require them to *explore*. We believe that visualizations inherently excel in such tasks in contrast to text-based approaches. For instance, participants used visualizations to answer questions that involve:

*(a)* Feature location such as *"which method contains the logic to increase the speed?" [S50]*, *"locate the feature that implements the logic: users are reminded that their accounts will be deleted if they do not log in after a certain number of months" [S117]*,

*(b)* Change impact analysis such as *"which classes of the package* `dependency` *will be directly affected by this change?" [S108]*, *"analyze the impact of adding items to a playlist" [S78]*,

*(c)* Analyze the rationale of an artifact such as *"find the purpose of the given application" [S117]*, *"what is the purpose of the application" [S162]*, and

*(d)* Pattern detection such as *"can you identify some interactions that are identical, along time, between groups of classes?" [S168]*, *"find the most symmetric subtree in the tree" [S169]*, *"locate the best candidate for the god class smell" [S180].*

Moreover, we classify these tasks according to the taxonomy proposed by Munzner [Mun14]. In it, she proposed that the task that motivates a visualization be classified using the following dimensions:

*(a) Analyze.* The goal of a visualization can be to *consume*, that is, to *discover* new knowledge, *present* already discovered knowledge,

and *enjoy* it; or it can be to create new material, which could be to *annotate* elements in the visualization, *record* visualization elements, and *derive* data elements from the existing ones.

*(b) Search.* All analyses require users to search. However, the type of search can differ depending on whether the target of the search and the location of that target are known. When both the target and its location are known, it is called *lookup*. When the target is known but not its location, it is called *locate*. When the target is unknown but its location is known, it is called *browse*. Finally, when both target and its location are unknown, it is called *explore*.

*(c) Query*. Once the searched targets are found, users query them. In tasks that involve a single target, the type of query is referred to as to *identify*. In tasks that involve two targets, it is referred to as to *compare*. Finally, in tasks that involve more than two targets, it is referred as to *summarize*.

Table 3.7: Classification of tasks used in experiments according to Munzner [Mun14]

| **Query** \ **Search** | Identify | Compare | Summarize |
|---|---|---|---|
| Lookup | — | S5, S125 | S108 |
| Locate | S123, S131, S137, S153, S177, S180 | S168 | S21, S71, S100, S112, S126, S149, S179 |
| Explore | S11, S173 | S72 | S13, S23, S24, S25, S32, S33, S40, S50, S61, S78, S96, S106, S117, S118, S127, S135, S140, S144, S148, S150, S154, S157, S162, S166, S169, S174, S181 |
| Browse | S66, S101 | S47 | S107, S141, S164 |

We classify all tasks collected from the studies into the *discovery* category. The results of the classification in the remaining two dimensions is presented in Table 3.7. We observed that most of the tasks were designed to *explore* and *summarize*, that is, participants have to *summarize* many targets that they neither know, nor for which they know the location in the visualization. Almost half of the twenty-seven tasks in this category were explicitly described in the studies, while for the other half we only found a rationale. Tasks in this category tackle:

*(a)* Comprehension *[S23]*, *[S24]*, *[S25]*, *[S32]*, *[S33]*, *[S40]*, *[S61]*, *[S96]*, *[S106]*, *[S148]*, *[S154]*, *[S174]*;

*(b)* Change impact analysis *[S50]*, *[S78]*, *[S118]*;

*(c)* Debugging *[S144]*, *[S150]*, *[S181]*;

*(d)* Code Structure *[S140]*, *[S157]*;

*(e)* Project Management *[S166]*, *[S169]*;

*(f)* Rationale *[S13]*, *[S117]*, *[S127]*, *[S162]*; and

*(g)* Refactoring *[S135]*.

We found seven other studies with tasks in which participants were asked to *summarize* targets but in which the targets were known, and therefore we classified them in the *locate* category. Studies in this category involve tasks that deal with:

*(a)* Comprehension *[126]*;

*(b)* Debugging *[S21]*, *[S71]*;

*(c)* Dependencies *[100]*, *[149]*;

*(d)* Code structure *[112]*; and

*(e)* Project Management *[S179]*.

Only five studies involved tasks that asked participants to compare two targets. All of these tasks related to comprehension. Finally, the tasks of ten studies involved *identify*ing a single target. These tasks deal with:

*(a)* Comprehension *[S11]*, *[S101]*, *[S173]*, *[S180]*;

*(b)* Change impact analysis *[S177]*; and

*(c)* Debugging *[S66]*, *[S123]*, *[S131]*, *[S137]*, *[S153]*.

Table 3.8: A summary of the dependent variables found in experiments.

| **Dependent Variable** | | **References** | **#** |
|---|---|---|---|
| User Performance | Not Explicit | S96, S108 | 2 |
| | Time | S5, S11, S32, S40, S71, S107, S125, S137, S144, S162, S164, S173, S174, S177, S180 | 15 |
| | Correctness | S5, S11, S13, S21, S24, S25, S32, S33, S40, S47, S71, S72, S78, S101, S106, S107, S108, S118, S123, S125, S126, S137, S144, S150, S162, S164, S168, S173, S179, S180 | 29 |
| | Effectiveness | S13, S21, S50, S66, S72, S78, S100, S101, S112, S127, S131, S141, S148, S157, S162, S164, S166 | 17 |
| | Completion | S50,S164 | 2 |
| | Recollection | S150,S180 | 2 |
| | Others | Visual Effort (S144), Scalability (S32), Efficiency (S32) | 3 |
| User Experience | Not Explicit | S96, S126, S49 | 3 |
| | Usability | S11, S13, S32, S40, S61, S117, S137, S140, S49, S153, S164, S169, S177, S181 | 14 |
| | Engagement | S154, S177 | 2 |
| | Understandability | S118, S181 | 2 |
| | Feeling | Enjoyment (S32), Intuitive (S137), Satisfaction (S164), Confidence (S107, S126) | 5 |
| | Others | Acceptability (S164), Learnability (S164), Difficulty (S180) | 3 |

Table 3.9: Statistical tests used to analyze data from experiments.

| Id. | Test | Reference | # |
|---|---|---|---|
| T1 | ANOVA | S25, S32, S40, S107, S144, S164, S174, S177, S180 | 9 |
| T2 | Pearson | S25, S40, S50, S107, S108, S150 | 6 |
| T3 | Cohen | S107, S150 | 2 |
| T4 | Wilcoxon | S101, S107, S126, S150, S164 | 5 |
| T5 | Student T | S5, S72, S101, S137, S162 | 5 |
| T6 | Shapiro-Wilk | S107, S162, S177, S180 | 4 |
| T7 | Kruskal-Wallis | S25, S108, S180 | 3 |
| T8 | Mann-Whitney | S25, S107, S168 | 3 |
| T9 | Descriptive Statistics and Charts | S24, S78, S118, S125, S131, S141, S154, S173, S179 | 9 |
| T10 | Levene | S162, S180 | 2 |
| T11-T18 | | Tukey (S180), Mauchly (S174), Greenhouse-Geisser (S174), Friedman (S21), Hotelling (S71), Kolmogorov-Smirnov (S72), Spearman (S25), Regression Analysis (S24) | 8 |

## 3.5 Discussion

We now revisit our research questions. Firstly, we discuss the main characteristics that we found amongst the analyzed evaluations. Secondly, we discuss whether the conducted evaluations are appropriate considering their scope. Finally, we discuss the threats to the validity of our investigation.

***RQ1.)*** *What are the characteristics of evaluations that validate the effectiveness of software visualization approaches?*

**Beyond traditional data collection methods**. The methods used to collect data during the evaluation have to facilitate the subsequent analysis. Consequently, in a formative experiment researchers interview participants to freely explore aspects of complex phenomena. In a case study researchers can interview developers in their work environment, which can help researchers to formulate hypotheses that can be tested

in experiments. Questionnaires can be used in surveys for exploration, reaching a higher number of participants who can provide researchers feedback of past experiences. We observed that several studies record sessions with participants. Afterwards, these records are used to dissect a user's performance (*e.g.,* correctness of answers and their completion time) and experience (*e.g.,* level of engagement of participants with a tool). We observed that few non-traditional methods are used: (*i*) eye tracking to capture how participants see the elements in visualizations; (*ii*) log analysis to investigate how participants navigate visualizations; and (*iii*) emotion cards to help participants to report their feelings in a measurable fashion. Finally, we believe that the capabilities of recent devices used to display visualizations ( *e.g.,* mobile phones, tablets, head-mounted displays) can complement the standard computer screen, and provide researchers with useful data for investigating both user performance and user experience.

**Thorough reports of *anecdotal evidence* and *usage scenarios***. Tool adoption can be considered the strongest evidence of the usability of an application [ANAV10]. However, we observe a lack of rigor amongst studies that reported anecdotal evidence. Normally, these studies report that tools were used, but often they do not specify the context, for instance, whether the tools are freely adopted or enforced as a requirement in a software engineering teaching course. Moreover, they describe subjective feedback from users using expressions such as "the tool was used with much success" *[S55]*, "feedback was positive" *[S97]* We propose that also reporting objective evidence, for instance number of downloads, would help them in making a stronger case to support the effectiveness of visualizations.

We also observed that one third of studies employed usage scenarios to demonstrate the effectiveness of the software visualization approaches. Typically they describe how the approach can answer questions about a software system. Normally, usage scenarios are carried out by the researchers themselves. Although researchers in the software visualization field are frequently both experts in software visualization and also experienced software developers, we believe they are affected by construction bias to perform the evaluation. Usage scenarios can help researchers to illustrate the applicability of a visualization approach. In fact, use cases that drive usage scenarios can reveal insights into the applicability of an visualization approach in an early stage [HHPS07]. Nonetheless, we believe they must involve external developers of the target audience who can produce a less biased evaluation, though related

work [HRW00] found that software engineering students can be used instead of professional software developers under certain conditions. We found multiple subject systems in usage scenarios, of which the most popular are open source. We reflect that open source software systems provide researchers an important resource for evaluating their proposed visualization approaches. They allow researchers to replicate evaluations in systems of various characteristics (*e.g.,* size, complexity, architecture, language, domain). They also ease the reproducibility of studies. However, we think that defining a set of software systems to be used in benchmarks would facilitate comparison across software visualization evaluation [MM03, MFB+17].

**The value of visualizations beyond *time* and *correctness***. We believe that it is necessary to identify the requirements of developers and evaluate whether the functionality offered by a visualization tool is appropriate to the problem. Indeed, past research has found a large gap between the desired aspects and the features of current software visualization tools [BK01]. A later study [SOT08b] analyzed desirable features of software visualization tools for corrective maintenance. A subsequent study [KM10] analyzed the requirements of visualization tools for reverse engineering. We observed, however, little adoption of the proposed requirements. Usability is amongst them the most adopted one. Scalability was adopted only in one study *[S32]*. Others such as interoperability, customizability, adoptability, integration, and query support were not found amongst the variables measured in experiments (see Table 3.8). We observed that even though none of the studies proposed that users of software visualizations should find answers quickly (*i.e.,* time) and accurately (*i.e.,* correctness), there are many evaluations that only considered these two variables.

We observed that evaluations in most studies aimed at proving the effectiveness of software visualization approaches. However, some studies do not specify *how* the effectiveness of the visualization is defined. Since something *effective* has "the power of acting upon the thing designated",[1] we reflect that effective visualization should fulfill its designated requirements. Then we ask *what are the requirements of software visualization?* We extract requirements from the dependent variables analyzed in experiments. We observed that the two main categories are user performance and user experience. Indeed, practitioners who adopt a visualization ap-

[1] "effective, adj. and n." OED Online. Oxford University Press, June 2017. Accessed October 27, 2017.

proach expect to find not only *correct* answers to software concerns, they expect that the visualization approach is also *efficient* (*i.e.,* uses a minimal amount of resources), and helps them to find answers in a short amount of *time* [VW06]. However, they also aim at obtaining a good experience in terms of *(i) engagement* when the target audience is composed of students of a software engineering course; *(ii) recollection* when the audience involves developers understanding legacy code [B+56]; and *(iii)* positive *emotions* in general.

We believe that effective software visualization approaches must combine various complementary variables, which depend on the objective of the visualization. That is, variables used to explicitly define effectiveness relate to the domain problem and the tasks required by a particular target audience. We think that a deeper understanding of the mapping between users' desired variables to usage scenarios of visualization can bring insights for defining *quality metrics* [BTK11] in the software visualization field.

**The *case* in case studies**. We classified twelve papers into the case study category. In these papers, we identified a *case* that is neither hypothetical nor a toy example, but a concrete context that involves a real world system in which developers adopted a visualization approach to support answering complex questions. In only one paper *[S90]* did we find a thorough evaluation that describes the use of various research methods to collect data such as questionnaires and interviews. In contrast, in others we found less detail and no explicit description of the methods employed to collect data. In particular, in three papers *[S52,S114,S151]* a reference was given to a paper that contains more details. We observed that in studies in which authors come from industry *[S56,S90,S114]* there are many details provided as part of the evaluation. In all of them, (*i*) users who evaluated the proposed visualization approach were senior developers from industry, and (*ii*) the evaluation adopted a qualitative analysis. Case studies are often accused of lack of rigor since biased views of participants can influence the direction of the findings and conclusions [Yin13]. Moreover, since they focus on a small number of subjects, they provide little basis for generalization.

In summary, we reflect on the need for conducting more case studies that can deliver insights into the benefits of software visualization approaches, and highlight the compulsion of identifying a concrete real-world *case*.

**The *scope* of experiments in software visualization**. Table 3.10 summarizes our extension to the framework proposed by Wohlin *et al.* [WRH+12] to include key characteristics of software visualizations. We believe that the extended framework can serve as a starting point for researchers who are planning to evaluate a software visualization approach. Each row in the table can be read as follows:

> "Analyze *[Object of study]* executing in a *[Environment]* to support the *[Task]* using a *[Technique]* displayed on a *[Medium]* for the purpose of *[Purpose]* with respect to *[Quality Focus]* from the point of view of *[Perspective]* in the context of *[Context]*."

We used the framework to describe the scope of a recent experiment of 3D visualization in immersive augmented reality [MBN18].

***RQ2.)*** *How appropriate are the evaluations that are conducted to validate the effectiveness of software visualization?*

**Explicit goal of evaluations**. We observed that studies often do not explicitly specify the goal of the evaluation. They formulate sentences such as "To evaluate our visualization, we conducted interviews ..." *[S100]*. We investigate what aspects of the visualization are evaluated. We reflect that a clear and explicit formulation of the goal of the evaluation would help developers to assess if the evaluation provides them enough evidence that support the claimed benefits of a proposed visualization approach. Although in most studies we infer that the goal is to evaluate the effectiveness of a visualization, in only a few studies is there a definition of effectiveness. For instance, one study *[S131]* defines effectiveness of a visualization in terms of the number of statements that need to be read before identifying the location of an error; however, we believe this definition suits better the definition of *efficiency*. Indeed, practitioners will benefit from effective and efficient software visualization. Nonetheless, we believe the game-changing attribute of a visualization resides in the user experience, for which multiple variables should be included in evaluations (*e.g.,* usability, engagement, emotions).

Table 3.10: The evaluation scope of experiments in software visualization (explicit tasks are check marked ✓).

| Ref. | Object of Study | Task | Env. | Technique | Med. | Purpose (To gain insights on ...) | Quality Foc. | Pers. | Context | Statistical Test |
|---|---|---|---|---|---|---|---|---|---|---|
| S5 | *UML diagram notation* | Identify if an UML diagram correspond to a specification | – | UML | SCS | Whether a specification matches a diagram | Performance | All | 35 CS students | T5 |
| S11 | Genisom | Search for information held within the self-organizing map. | – | City | SCS | How users extract information from a visual | Per./Exp. | All | 114 CS students | – |
| S13 | Xia | Why a particular file changed | – | Node-link | SCS | Initial requirements | Per./Exp. | All | 5 CS students | – |
| S21 | *Spreadsheets* | Localization of faulty cells | – | Aug. source code | SCS | Faulty cells in spreadsheets | Per./Exp. | Nov. | 87 CS students | T14 |
| S23 | *Reduc. Cognitive Effort* | Tasks related to distributed computations | – | Node-link; Iconic | SCS | Cognitive economy | Performance | Nov. | 20 CS st. (5 female) | – |
| S24 | *DancingHamst.;Marble.* | Tasks related to algorithm analysis | – | Anim. Node-link | SCS | The impact of visualization in learning | Performance | Nov. | 12 CS st.; 43 CS st. | T18 |
| S25 | *Algorithm visualization* | Tasks related to the sorting algorithms | – | Aug. source code | SCS | The impact of visualization in learning | Performance | Nov. | 157 CS students | T1,T7,T8,T17 |
| S32 | *GrowingSquares* | Is process $x$ causally related time to process $y$? ✓ | – | Node-link; Hasse | SCS | The impact of a technique | Performance | All | 12 part. (4 female) | T1 |
| S33 | PlanAni | Tasks related to sorting algorithms | Various | Aug. source code | SCS | Supporting teaching programming in CS | Performance | Nov. | 91 CS students | – |
| S40 | *Variable dependency* | Complete an unfinished function | – | UML | SCS | Intra-procedural variable dependencies | Per./Exp. | All | 38 CS st. (3 female) | T1,T2 |
| S47 | *UML class diagram* | Match the role of a particular class | – | UML | SCS | Stereotype-based architectural UML layout | Correctness | All | 20 CS students | – |
| S50 | *Wear-based filtering* | Change the program to obtain an expected behavior ✓ | – | UML | SCS | The impact of using wear-based filtering | Performance | All | 7 male developers | T2 |
| S61 | *Algorithm visualization* | Tasks related to algorithm analysis | – | Node-link | SCS | The impact of using concept keyboards | Experience | Nov. | 17 CS st.; 18 CS st. | – |
| S66 | *Social agents* | What faults did you find, and when did you find each one? | – | Iconic | SCS | The impact of the tool | Effectiveness | Nov. | 22 CS students | – |
| S71 | jGrasp | Find and correct all the non-syntactical errors | Various | Aug. source code | SCS | Supporting teaching programming in CS | Performance | Nov. | - | T15 |
| S72 | *Algorithm visualization* | How different are Prim and Dijkstra algorithms? ✓ | – | Node-link | SCS | The impact of the narrative visualization approach | Performance | Nov. | 34 CS students | T5,T16 |
| S78 | Gilligan | Analyze the impact of changing a program. ✓ | – | Aug. source code | SCS | The impact of a tool | Performance | All | 6 participants | T9 |
| S96 | SolidFX | Tasks related to reverse-engineering open-source code | Windows | HEB; Pixel | SCS | Architecture, metrics and dependencies | Per./Exp. | All | 8 part. (ind. & acad.) | – |
| S100 | Enhance | Find dependencies between structural elements | – | Node-link | SCS | How devs. understand exception-handling constructs | Performance | Nov. | 3 CS students | – |
| S101 | saUML | Select the candidate that best describes the depicted behavior | – | UML | SCS | Synchronization-adorned sequence diagrams | Performance | Nov. | 24 CS students | T4,T5 |
| S106 | Various | Tasks related to program comprehension and maintenance | – | Various | SCS | The impact of a tool | Performance | All | 90 part. (ind. & acad. | – |
| S107 | *UML Class diagram* | Identify classes to be changed to add a requirement | – | UML | SCS | The impact of the layout | Per./Exp. | All | 45 CS students | T1-T4,T6 |
| S108 | Version Tree vs Augur | Which classes will be directly affected by this change? ✓ | – | Node-link | SCS | Benefits of vis. for open source newcomers | Performance | Nov. | 27 CS students (9 females) | T2,T7 |
| S112 | *UML Class diagram* | Count abstract classes to see if proxies are distinguished | – | UML | SCS | Initial requirements | Performance | All | 8 CS stud. & staff (2 female) | – |
| S117 | CodeMap | Find the purpose of the given application ✓ | – | Island | SCS | Visualizations embedded in the IDE | Experience | All | 7 part. (ind. & acad.) | – |
| S118 | ProfVis | How the program can be modified to improve performance ✓ | Java | Node-link | SCS | Execution traces of Java programs | Per./Exp. | All | 4 participants | T9 |
| S123 | AllocRay | Find the location of a memory leak | – | Pixel | SCS | Allocation patterns and memory problems | Performance | All | 4 developers | – |
| S125 | SystemHotspotsView | How much bigger is “Component” than “Window” class? ✓ | – | Polymetric views | SCS | Visualization rendered on a wall display | Performance | All | 11 part. (3 fem. ind. & acad.) | T9 |
| S126 | StenchBlossom | Identify code smells | Eclipse | Aug. source code | SCS | Supporting software quality based on code smells | Per./Exp. | All | 12 part. (ind. & acad. | T4 |
| S127 | *Software dev. lifecycle* | Analyze the context, and roles of involved people in projects | – | Node-link | SCS | How devs. draw diagrams of soft. lifecycle | Performance | All | 8 par. (CS stud. & resear.) | – |
| S131 | *Constellation* | Identify the location in the code of a fault | – | Node-link | SCS | Software understanding and pattern recognition | Performance | All | 30 CS students | T9 |
| S135 | E-Quality | Select the most significant refactoring candidates of a program | – | Node-link; Iconic | SCS | Visualization of design flaws and refact. opportunities | Experience | All | 16 developers | – |
| S137 | Gzoltar | Identify the location in the code of a fault | Java;Ecli. | Icicle; Treemap | SCS | Fault localization for debugging Java progs. | Per./Exp., ... | All | 40 CS students | T5 |
| S140 | PNLV; IMMV | What interesting visual structures do you find in the vis.? ✓ | – | Node-link | SCS | Visualization for understanding an unknown system | Experience | All | 8 part. (ind. & acad.) | – |
| S141 | SourceVis | How many interfaces does this class depend on? ✓ | – | Polymetric views | MMT | Vis. on multitouch tab. for co-located collab. in unders. | Performance | All | 6 par. (CS stud. &resear.) | T9 |
| S144 | SeeIT3D | Identify why the program produce a poor print quality ✓ | Eclipse | City | SCS | Visualization for architecture of Java systems | Performance | All | 97 CS students | T1,T18 |
| S148 | ChronoTwigger | Investigate the software while thinking out loud | – | Node-link | I3D | Visualization of the development process and testing | Performance | All | 3 dev. (1 female) | – |
| S149 | regVIS | Track of the Overall Control Flow ✓ | Windows | Visual language | SCS | Supporting assembler control-flow of regular expr. | Experience | All | 10 par. (CS stud. & resear.) | – |
| S150 | *Compiler Messages* | Identify the cause of an error through highlighted elements ✓ | – | Aug. source code | SCS | Helping devs. on comprehending error messages | Performance | Nov. | 28 CS students | T2-T4 |
| S153 | SIFEI | Find a failure and specify a test scenario for it ✓ | Excel | Visual language | SCS | Spreadsheets formulas | Per./Exp. | All | 9 part. (ind. & acad.) | – |
| S154 | TiledGrace | Describe the behavior of a program ✓ | Web | Visual language | SCS | Supporting programming in the Grace language | Experience | Nov. | 33 CS students | T9 |
| S157 | CodeSurveyor | Rank the code maps that best represent the codebase ✓ | – | Island | SCS | Code maps for learning and navigating systems | Performance | All | 5 dev. (1 female) | – |
| S162 | ExplorViz | What is the purpose of the WWWPRINT application? ✓ | Web | City | I3D | Architecture based on metric analysis | Performance | All | 25 CS students | T5,T6,T10 |
| S164 | VIMETRIK | What is the number of compilation units in “Tomcat”? ✓ | – | City; Matrix | SCS | Metrics and visualization of a software system | Per./Exp. | All | 21 CS students | T1,T4,T9 |
| S166 | *Working Sets* | Analyze the developer activity on entities of the working sets | – | Node-link | SCS | Visualization of the evolution of working sets | Performance | All | 14 developers | – |
| S168 | CuboidMatrix | Identify identical interactions, along time, between classes? ✓ | – | Space-time cube | SCS | The impact of the tool in software comprehension | Performance | All | 8 par. (CS stud. & resear.) | T8 |
| S169 | Perquimans | Which sheets contained the most dangerous formula practice ✓ | – | Node-link | SCS | Vis. tool’s support of exploration, and quantification | Experience | All | 4 CS students | – |
| S173 | *Indented Hierarchy* | Find the most symmetric subtree in the tree. ✓ | – | Pixel | SCS | A technique compared with node-link diagrams | Performance | All | 18 vis. exp. (3 female) | T9 |
| S174 | *Regex textual vs graph.* | Is $ABC$ in the language defined by a regular expression? ✓ | – | Aug. source code | SCS | A graphical notation on the readability of a regex | Performance | Nov. | 22 par. (CS stud. and staff) | T1,T12,T13 |
| S177 | Code Park | Identify where in the code add the logic to support a feature ✓ | – | City | I3D | The impact of the tool on usability and engagement | Per./Exp. | Nov. | 28 CS st. (6 female) | T1,T6 |
| S179 | iTraceVis | Where did the dev. not look at in the same method? ✓ | Eclipse | Heatmap | SCS | Eye movement data of code reading | Performance | All | 10 CS students | T9 |
| S180 | CityVR | Locate the best candidate for the god class smell ✓ | Pharo; U. | City | SCS | Architecture based on metrics in OOP | Performance | All | 21 participants | T1,T6,T7,T10,T11 |
| S181 | MethodExecutionRep.x | Tasks related to execution report for profiling and debugging ✓ | Java | Charts | SCS | Supporting summarization of methods execution | Per./Exp. | All | 11 part. (ind. & acad.) | – |

**Experiments' tasks must be in-line with evaluations' goal**. Software visualizations are proposed to support developers in tasks dealing with multiple development concerns. A problem thus arises for developers willing to adopt a visualization but who need to match a suitable visualization approach to their particular task at hand [MGN+16b]. We investigate how suitable a visualization approach is for the tasks used in evaluations. We reflect that proving a software visualization approach to be effective for tasks for which there exist other more appropriate tools (but not included in the evaluation) can lead to misleading conclusions. Since many evaluations included in our analysis do not state an explicit goal, and some of the remaining ones refer to rather generic terms (*e.g.,* effectiveness, usability) without providing a definition, understanding whether the tasks used in experiments are in-line with the goals of evaluations is still uncertain.

**Beyond *usage scenarios***. Related work concluded that describing a case study is the most common strategy used to evaluate software visualization approaches. Indeed, we found many papers that contain a section entitled *case study*; however, we observe that most of them correspond to usage scenarios used to demonstrate how the proposed visualization approach is expected to be useful. In all of them, the authors (who usually are also developers) select a subject system and show how visualizations support a number of use cases. For example, one study *[S158]* describes the presence of *independent judges*, but without providing much detail about them. In the past, such a self-evaluation, known as an *assertion* [ZW98], has been used in many studies, and is not considered an accepted research method for evaluation [WRH+00]. Instead, we prefer to refer to them as *usage scenarios* (as they are called in many studies). This name has also been adopted in the information visualization community [IIC+13], and therefore its adoption in software visualization will ease comparison across the two communities. Nonetheless, usage scenarios do not represent solid evidence of the benefits of proposed software visualization, and should be used only as a starting point to adjust requirements, and improve an approach.

**Surveys to collect software visualization requirements**. We observed that *surveys* are adequate to identifying requirements for software visualizations. Through a survey, the problems that arise in the development *tasks* carried out by a target *audience* that involve a particular *data* set can

be collected as assessed as potential candidates for visualization. Then, researchers can propose an approach that defines the use of a visualization *technique* displayed in a *medium*. We observed that a main threat in software visualization is the disconnect between the development concerns that are the focus of visualization, and the most complex and frequent problems that arise during real-life development.

**Report on thorough experiments**. Although formative evaluations can be useful at an early stage, evidence of the user performance and user experience of a software visualization approach should be collected via thorough experiments (when variables included in the evaluation can be controlled). Experiments should include participants of a random sample of the target audience and real-world software systems. Experiments should aim at reproducibility, for which open source software projects are suitable. Moreover, open source projects boost replicability of evaluations across systems of various characteristics. The tasks used in experiments should be realistic, and as already discussed, consistent with the goal of the evaluation, otherwise conclusions can be misleading. Finally, we observed that standardizing evaluations via benchmarks would promote their comparison.

In summary, we observed that the main obstacles that prevent researchers from doing more appropriate evaluations are *(i)* the lack of a ready-to-use evaluation infrastructure, *e.g.,* visualization tools to compare with; *(ii)* the lack of benchmarks that ease comparison across tools, *e.g.,* quality metrics; *(iii)* the tradeoff between the effort of conducting comprehensive evaluations and little added value to paper acceptance; and *(iv)* the difficulties to involve industrial partners willing to share resources, *e.g.,* include participants of the target audience.

### 3.5.1 Threats to Validity

*Construct validity.* Our research questions may not provide complete coverage of software visualization evaluation. We mitigated this threat by including questions that focus on the two main aspects that we found in related work: *(1)* characterization of the state-of-the-art, and *(2)* appropriateness of adopted evaluations.

*Internal validity.* We included papers from only two venues, and may have missed papers published in other venues that require more thorough evaluations. We mitigated this threat by identifying relevant software visu-

alization papers that ensure an unbiased paper selection process. Therefore, we selected papers from the most frequently cited venue dedicated to software visualization: SOFTVIS/VISSOFT. We argue that even if we would have included papers from other venues the trend of the results would be similar. Indeed, related work did not find important differences when comparing software visualization evaluation in papers published in SOFTVIS/VISSOFT to papers published in other venues [MIK+16, SBCS14]. Moreover, our results are in line with the conclusions of related work that have included papers from multiple venues [LHIE18, SLB14, NTM+13]. We also mitigated the paper selection bias by selecting peer-reviewed full papers. We assessed the quality of these papers by excluding model papers (*i.e.,* commentary, formalism, taxonomy) that are less likely to include an evaluation. However, since software visualization papers do not specify their types, we may have missed some. We mitigated this threat by defining a cross-checking procedure and criteria for paper type classification.

*External validity.* We selected software visualization papers published between 2002 to 2017 in SOFTVIS/VISSOFT. The excluded papers from other venues or published before 2002 may affect the generalizability of our results.

*Conclusion validity.* Bias in the data collection procedure could obstruct reproducibility of our study. We mitigated this threat by establishing a protocol to extract the data of each paper equally, and by maintaining a spreadsheet to keep records, normalize terms, and identify anomalies.

## 3.6 Conclusion

We reviewed 181 full papers of the 387 that were published to date in the SOFTVIS/VISSOFT conferences. We extracted evaluation strategies, data collection methods and other various aspects of evaluations. We found that 62% (*i.e.,* 113) of the proposed software visualization approaches do not include a strong evaluation. We identified several pitfalls that must be avoided in the future of software visualization: *(i)* evaluations with fuzzy goals (or without explicit goals), for which the results are hard to interpret; *(ii)* evaluations that pursue effectiveness without defining it, or that limit the assessment to time, correctness (user performance) and usability (user experience) while disregarding many other variables that can contribute to effectiveness (*e.g.,* recollection, engagement, emotions); *(iii)* experiment tasks that are inconsistent with the stated goal of the evaluation; *(iv)* lack

of surveys to collect requirements that explain the disconnect between the problem domains on which software visualization have focused and the domains that get the most attention from practitioners; and *(v)* lack of rigor when designing, conducting, and reporting on evaluation.

We call researchers in the field to collect evidence of the effectiveness of software visualization approaches by means of (*1*) case studies (when there is a case that must be studied *in situ*), and (*2*) experiments (when variables can be controlled) including participants of a random sample of the target audience and real-world open source software systems that promote reproducibility and replicability.

We believe that our study will help *(a)* researchers to reflect on the design of appropriate evaluations for software visualization, and *(b)* developers to be aware of the evidence that supports the claims of benefit of the proposed software visualization approaches.

We have presented a characterization of the evaluations conducted in software visualization approaches. We have identified common pitfalls, and presented guidelines to help researchers in the field who need to evaluate their visualization approaches. We observed that the quality of evidence of the effectiveness of proposed software visualization approaches varies. Amongst the many proposed visualizations there are some that have presented strong evidence of their effectiveness, which also sometimes have been maintained by a large community since long time, and shown maturity; while others might be promising visualizations that support unexplored software concerns. We ask whether a developer willing to adopt software visualizations can identify a suitable approach. In the following chapter we address this concern, and elaborate on our attempts to fill the gap between proposed visualizations and their practical applications.

# 4 Actionable Software Visualization

## 4.1 Introduction

Little research has been carried out to fill the gap between existing software visualization techniques and their practical applications. For example, one study [HDSP09] proposed an approach for generating visualizations specifically for maintenance tasks. Another study [SS11] proposed an approach to derive interactive visualizations from descriptions of code analysis tasks. That approach, however, required developers to use a domain-specific language to describe the task. A study [GTS10] reported on how information visualization novices construct visualizations. In it, the authors analyzed the usage of basic visualization techniques such as charts and scatter plots. Although these techniques provide limited support for the analysis of development concerns, they acknowledge the need for tools that suggest a potential visualization.

In this chapter we elaborate on our efforts to fill the gap between existing software visualization techniques and their practical applications. First, we discuss a *meta-visualization* approach that relates frequent questions that arise during developments to actionable visualization examples that they easily be tailored and apply to their particular context. Secondly, we

introduce a *curated catalog* of available software visualization tools that have been proposed, which we characterize by their target task, execution environment, employed visualization techniques, and evidence of effectiveness. Finally, we elaborate on early results of developing an *ontology* of software visualizations that encapsulate their main characteristics.

## 4.2 MetaVis

Software visualization can play an effective role to answer a number of questions that arise during software development. For instance, before *"refactoring a legacy software system"*, developers should know *"what are the dependencies of this code?"* Obviously, a visualization on which developers can identify entities and trace dependencies would help them to prioritize the tasks that might require more effort.

Though existing visualizations are often characterized by the types of questions that they are well-suited to answer, as we show in Chapter 2, each work introduces a new tool or technique [MGN16a]. That is, developers may need to explore a long list of existing visualizations to adopt the one that fits their needs. Consider the case of the Roassal visualization engine [ABC+13] available for Smalltalk. Although it provides 363 examples that developers can adapt, the examples belong to 36 different visualization categories that are categorized based on the addressed technique or feature rather than on development concerns.

We conjecture that the difficulties that developers experience in searching for a suitable visualization obstruct their adoption. We believe that providing visualization support within IDEs and categorizing existing techniques in a way that maps to the certain needs for development tasks is very helpful for developers. We have performed a small experiment that supports our hypothesis. We instrumented the Roassal example browser to monitor the behavior of users who have installed Roassal recently, and thus have demonstrated their interest in adopting visualizations. Over the period of one month we collected the usage behavior of 58 anonymous users. They showed a trend that confirms our intuition. The top 10 users who browsed the highest number of examples had to traverse at least 5 categories on average (with a maximum of 13 categories traversed by a user who tried 60 examples) before they found an example of interest.

We propose *MetaVis* [MGN+16b], a tool for exploring visualization examples suitable to answer frequent development questions. MetaVis

uses a tag-iconic cloud-based visualization to connect frequently recurring and meaningful words, called *tags*, retrieved from the collected questions to icons that represent visualization examples. The tool allows users to discover and adapt appropriate visualization examples with the help of tags that are relevant to their needs. We present initial results of integrating MetaVis into the Pharo programming environment [DZHC17]. Amongst 173 questions that developers frequently ask during software development, collected from related work, we assigned 76 of them to 49 suitable visualization examples selected from 363 examples in the Roassal engine. To ease the reproducibility of our research MetaVis and our data sets are publicly available[1].

Figure 4.1 shows the *MetaVis* visualization, which is based on three main components: (1) a set of developer's questions, (2) a set of visualization examples, and (3) the relationships between the two sets. We now explain these components and elaborate on how the visualization supports users for their comprehension.

### 4.2.1 Developer's Questions

Developers often should answer several questions to perform a development task. Indeed, a complex task, such as *"refactoring a legacy software system"*, is broken down into some specific questions like *"what are the dependencies between these two packages?"*, *"who is the owner or expert for this code?"*, etc. Various researchers have mined, analyzed and thoroughly classified such questions. LaToza and Myers [LM10] surveyed 179 seasoned developers who answered *"what hard-to-answer questions about code have you recently asked?"*, and identified 91 types of such questions. Sillito *et al.* [SMDV06] collected 44 types of questions from two observational studies: in one study they interviewed 9 computer science graduate students, and in another, 16 industrial programmers. Fritz and Murphy [FM10] also interviewed 11 developers with varying expertise in industry, and gathered 46 types of questions.

We could identify 173 distinct questions from the aforementioned studies. We studied these questions to identify those for which visualization represents a suitable means to reveal an answer. Each participant studied each question independently. In our experience, questions that aim at analyzing relationships among entities, comparing metrics and classifying

[1] `http://scg.unibe.ch/research/meta-vis`

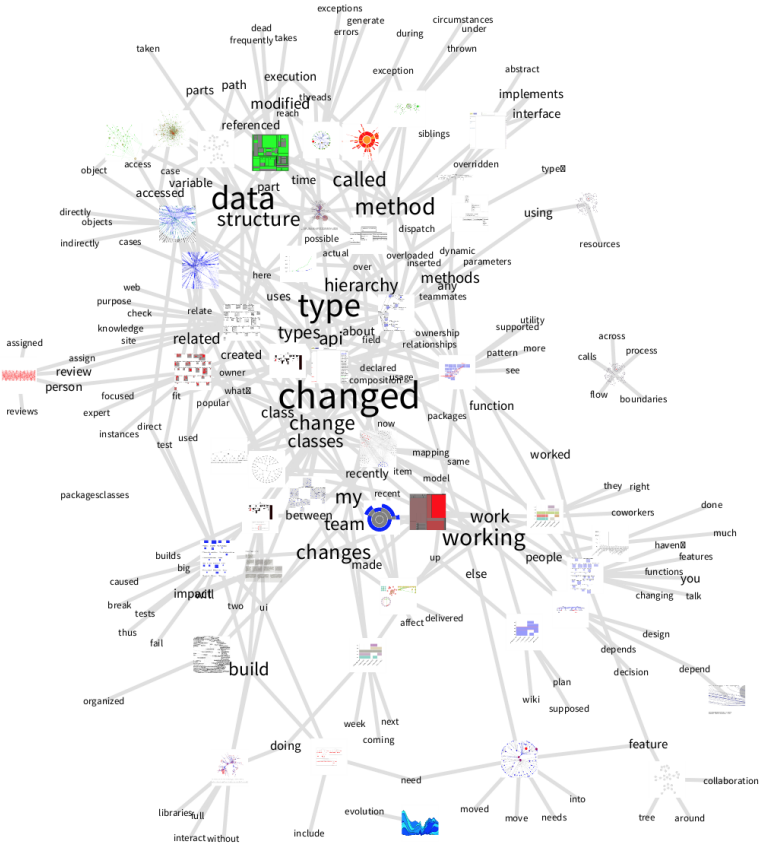



Figure 4.1: MetaVis visualization depicts *tags*, collected from frequent questions that arise during development, linked to *icons* of suitable visualization examples.

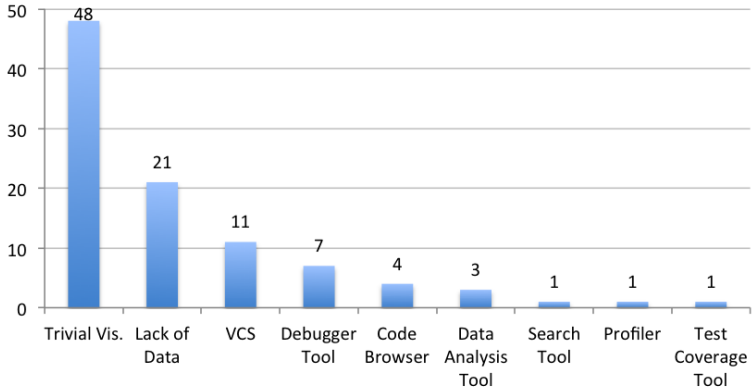



Figure 4.2: Classification of the 97 excluded questions.

entities using certain criteria can benefit from visualization. At the end, we compared our results and discussed any conflict. We agreed that out of 173 questions visualization significantly helps to answer 76 of them (44%) like *"how big is this code?"*, *"where is this method called or type referenced?"*, and *"what classes have been changed most?"* just to name a few. We excluded the 97 remaining questions (shown in Figure 4.2) for multiple reasons. We mainly excluded questions (1) already supported by tools part of the standard development environment (*e.g., VCS*, *Debugger tool*, *Code browser*), and (2) on which visualization is trivial and gathering the data represents most of the answer, labeled as *Trivial Visualization*, or on which the input data is not available (*e.g.,* assumptions, intent, policies), labeled as *Lack of Data*. We thus excluded (1) questions such as *"what are the arguments to this function?"* for which the *debugger* is appropriate, *"who made a particular change?"*, which can be queried in the *versioning control System*, or *"is this code tested?"* for which a *test coverage* tool will provide a more comprehensive analysis, and (2) questions such as *"what parameter values does each situation pass to this method?"*, *"how many recursive calls happen during this operation?"*, and *"why was it done this way?"* Similar questions from other studies could be incorporated into our approach by expanding the set of related tags that represent a given development concern.

### 4.2.2 Visualization Examples

We take the specific case of the examples that are shipped with the Roassal visualization engine. Roassal is a general-purpose visualization engine, which means that it is not limited to visualization of software concerns. It provides 363 examples that show novice users how various APIs can be used to obtain a certain visualization. The examples are organized into 36 categories (*e.g., Color Palettes*, *Interaction*, *Tree map*). Users browse a category and see small screenshots of its visualization examples. Users can select an example, inspect its implementation and shape it to their needs.

We analyzed the 363 examples one by one. Although examples are not designed specifically for visualization of software development concerns, we found 49 that provide a useful starting point on which users can build visualizations to answer some of the questions identified in 4.2.1.

Identifying which of dozens of questions relate to the actual need of a developer is a hard task. Consequently, MetaVis automatically split questions into frequently occurring and meaningful words (*e.g.,* verbs, nouns), called *tags*, that we manually relate to suitable visualization examples. In the following we elaborate on the visualization that we designed for their exploration.

### 4.2.3 TIC: Tag-Iconic Cloud-Based Visualization

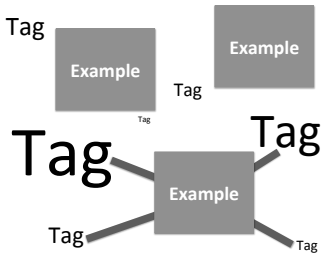



Figure 4.3: TIC wireframe composed of (1) tags from questions, (2) visualization examples, and (3) on-demand edges that connect tags and examples.

The TIC visualization follows Shneidermann's visualization mantra [Shn96]: first users explore an *overview* of the cloud of development concerns to identify tags of interest, then they *zoom* into details of surrounding visualization examples, and finally they obtain *details-on-demand* by selecting an example that they can modify to fit their needs. Figure 4.3 shows the basic components of the TIC visualization: (1) tags that encode in their size how frequently they arise in the set of questions, (2) icons that represent

visualization examples, and (3) on-demand edges that connect tags to their suitable examples. We use a force-directed algorithm [FR91] to lay out the bigraph of tags and icons. As a consequence, related elements are clustered together, thus revealing types of visualization techniques that are suitable to tackle the development concerns represented by the tags in the neighborhood. Edges are transparent to avoid cluttering. They are revealed on demand when users hover over a tag or an icon.

We chose the tag cloud technique to ease the comprehension of our visualization. Its popularity makes it self-explanatory. However, we reflected that in a tag cloud typically the positions of tags do not encode data. We decided then to group tags by development concerns. We expect that this will encourage users to discover suitable visualizations proposed for other needs within the concern.

The TIC visualization can also be used to tackle problems in other domains. We consequently classify it using the five dimensions proposed by Maletic *et al.* [MMC02] to ease its reuse. The *task* tackled by our visualization is the exploration of appropriate visualization examples to answer development questions. The *audience* of this visualization are software developers who want to adopt visualization techniques for software analysis. The *target* data consists of a set of questions, a set of visualization examples, and a relation between questions and suitable examples for answering them. The *representation* is a tag-iconic cloud-based visualization that can be classified as *iconic-based* according to Keim's taxonomy [KK96], and the *medium* used to display the visualization is a high-resolution monitor with at least 2560 x 1440 pixels.

### 4.2.4 Implementation

We realized a prototype tool implementation of MetaVis in Pharo. The tool is based on the Roassal visualization engine and builds upon the GTInspector tool [CGNS15], which provides users with navigation and basic interactions (*e.g.,* zoom-in/out, pop-up, view center), and GTSpotter [SCG+15], which is used to search less frequent tags that can be difficult to find visually. *MetaVis* supports the following workflow: (1) users explore the cloud and select a visualization of their interest, (2) they inspect the associated code example and adapt it for their needs, and finally (3) they are able to put it into *action* and view the outcome visualization.

### 4.2.5 Analysis Example

In this section we present some sample questions from the literature, and show how MetaVis helps us to identify suitable visualizations to answer these questions.

***A.*** *Who is the owner or expert for this code? [LM10]*

We observe that *owner* and *expert* are not frequent tags in our data set, hence their corresponding tags are difficult to find at first sight and require us to search for them. When we search for *owner*, two results *owner* and *ownership* are returned. Once we select the first tag, the visualization centers and highlights it. We then follow three steps shown in Figure 4.4 (top): *(1)* we select one of the visualization examples that is linked to the selected tag (left pane); *(2)* the code example of the selected visualization appears in the center pane. We modify the source code towards the analysis of code authorship. In particular, we add line 4 to collect all distinct authors of the set of classes, add lines 5-6 to create an object that returns a different color for each author, and modify line 7 to assign those colors to methods based on their author; *(3)* we obtain a visualization (right pane) that shows classes with their methods colored according to their authors.

***B.*** *Where is this method called or type referenced? [SMDV06]*

We identify two potential tags in this question: *method* and *called*. In Figure 4.4 (bottom) we show the sequence of steps performed. The visualization pane (left) shows the tags that we spot at first glance since they are quite common. We select one depicting a node-base diagram of the linked visualization examples and inspect its source code. Although the example already includes the main elements required in the analysis (classes, dependent classes, relationships), the number of edges depicted obstruct the analysis of dependencies of a particular class. We add interaction to the class nodes to highlight their dependencies when we hover over one of the classes.

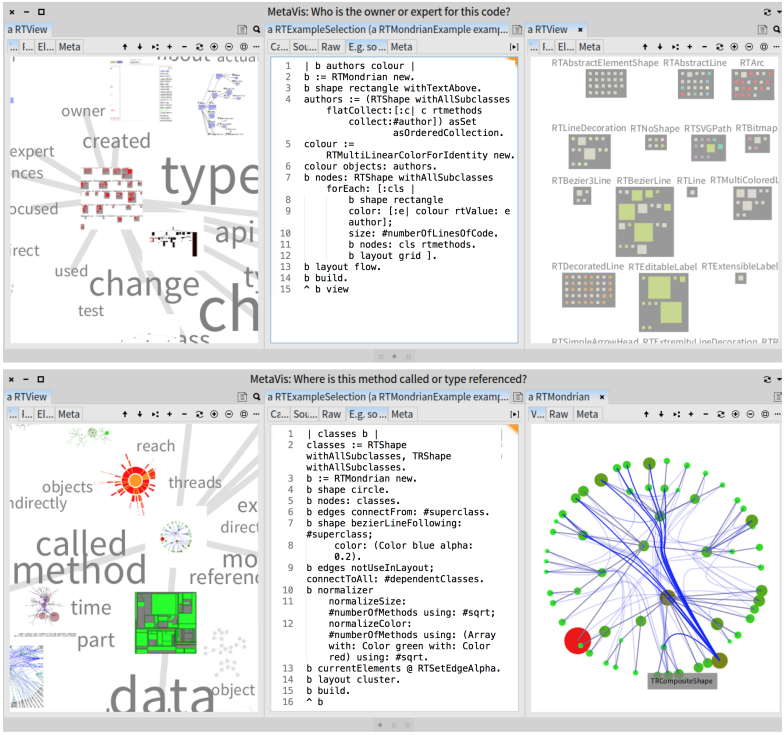



Figure 4.4: Two examples of the usage of MetaVis. On the top, we use it to answer *"who is the owner or expert of this code?"* The left pane shows the exploratory visualization that links a visualization to tags retrieved from questions. In the example, we look for *owner*, select a visualization example and start modifying its source code (center pane) to identify the authors of the various methods of classes. The resulting visualization is shown in the right pane. At the bottom, we aimed at answering *"where is this method called or type referenced?"* For this example we just needed to add interaction to nodes to highlight the outgoing edges representing dependencies.

### 4.2.6 Discussion

During the analysis of questions that were good candidates for visualization, we identified three key groups of questions:

*(1) Relating.* Some questions sought to analyze *relationships* among software artifacts such as types, methods, objects, exceptions, and libraries. For example *"what depends on this code?"*, *"how are these types related?"* We found that suitable visualizations for this group are based on node-link diagrams, parallel coordinates [ID91], and Sunburst [SCGM00].

*(2) Weighting.* Certain questions tried to *weigh* entities for comparison. Examples are *"how big is this code?"*, *"which part of this code takes the most time?"* The visualizations that we found suitable for them were mostly based on simple charts, TreeMap [JS91], and Polymetric Views [LD03].

*(3) Identifying.* Other questions aim to *identify* entities such as software artifacts, or people involved in development tasks. Examples are *"who is using that API?"*, *"who implements this interface?"* We recognize multiple visualization techniques suitable to tackle such questions, therefore we do not identify a particular preferred technique.

We observe that detecting what visualization techniques are frequently proposed to answer a particular group of questions (*e.g.,* relate, weigh, identify) suggests a future work direction on automating the process of visualization.

**Limitations.**

A general limitation of *MetaVis* is bias in the choice and size of the set of development questions, in the set of visualization examples, and in the relationships between them. We mitigated these limitations by building the set of questions from relevant research in the field, collecting examples from a visualization engine developed by a highly active community, and discussing the relationships (manually assigned). Regarding the *TIC* visualization technique, the size of the tags across multiple development concerns makes less frequent ones difficult to find visually. We observe that this issue can be mitigated by providing users with independent clouds for each development concern. Also the choice of words used to formulate the selected questions can affect the discoverability of development concerns; normalizing words and unifying synonyms could alleviate that issue.

### 4.2.7 Summary

Although large numbers of visualization techniques have been proposed, and much research has investigated their effective use, little support is available for developers seeking a suitable visualization for their task at hand.

We have studied related work and have collected questions that programmers frequently ask during software development. We manually mapped these questions to suitable visualization examples. We designed a tag-iconic cloud-based visualization that relates frequent tags retrieved from questions and links them to appropriate visualization examples. Developers explore the cloud, identify important tags for their particular needs, and find suitable examples that they can customize.

However, we observed that there are some more complex tasks for which lightweight visualizations are not suitable, and that might require dedicated software visualization tools. We argue that the lack of organization to characterize proposed software visualization tools obstructs their adoption. In the following section we introduce a curated catalog of proposed software visualization tools that we checked are available.

## 4.3 Software Visualization Tools

We built on the data sets from the proposed software visualization approaches (presented in Chapters 2 and 3). We reviewed the 387 software visualization papers published in VISSOFT/SOFTVIS conferences. In included in our catalog only software visualization tools that: *(i)* are identified with a name, and *(ii)* are publicly available in the web.

We scanned each paper to identify a name of the proposed software visualization approach. Then, we looked for an URL where the tool might be available. In most cases (where we do not find an URL in the paper), we searched the web using the name of the tool. When we did not find a positive result, we added "visualization" to the search keywords. When we found an available tool, we checked the last time when to tool was updated. Sometimes, we had to download the tool to check that date amongst the files. In the end, we found 70 software visualization tools that fulfill the criteria, and therefore, that we included in our catalog.

To characterized the 70 software visualization tools, we first identified whether the proposed tool focuses on *structure*, *behavior*, or *evolution* of software systems [Die07]. Then, for the tools in each category, we

identified the *development concern* dealt by the visualization. Instead of describing high-level tasks (*e.g.,* reverse-engineering), we formulated descriptions with the main keywords of the concerns (*e.g.,* "reports that summarize methods execution"), which we think can help developers to relate their particular context to the one envisioned by a proposed visualization tool. We also classified the tools based on their *execution environment* (*e.g.,* Eclipse plug-in), the employed visualization *technique* (*e.g.,* city metaphor), and the *medium* used to display them (*e.g.,* standard computer screen). Finally, we reused the data presented in Chapter 3 to highlight the maturity of tools that have proven effective to support the target task through evaluations.

Table 4.1 presents our curated catalog of 70 actionable software visualization tools classified by the software's *date* of last update, *environment* required to execute, employed visualization *technique*, *medium* (*e.g.,* standard computer screens SCS, immersive virtual reality I3D), and evidence of visualizations effectiveness through *evaluations*. Each tool's name is linked to a URL that contains instructions for downloading and installation. We reused Diehl's classification [Die07] of software visualization approaches. In it, approaches are classified into one of three *aspects*: behavior, evolution, structure.

### Behavior

Several visualization tools are proposed to support teaching various subjects in computer science. *ToonTalk* proposes a visual language (similar to Scratch [MBK+04]) to be used on the web, which targets the children's audience. We are not aware of evaluations. However, the tool has been maintained over the last twelve years, which shows evidence of maturity. Similarly, *Tiled Grace* offers a visual representation alternative to the textual mode when programming in the Grace language. Another mature tool is *Clack*, which helps students of network courses to understand the behavior of routers. *GraphWorks* focuses on supporting students of graph theory, though it has not been maintained in the last few years.

Some other tools are available to deal with understanding the execution of programs for testing. The Eclipse plug-in *Jive* (shown in Figure 4.5) stands out since it has been maintained for the last eleven years, which is congruent with the anecdotal adoption evidence of its effectiveness. Even though all of these tools are available, almost none of them have been maintained lately. Amongst them, *ProfVis* is the only one that has

Table 4.1: A curated catalog of 70 actionable software visualization tools.

| Asp. | Tool's Name | Date | Software Concern | Env. | Technique | Med. | Evaluation |
|---|---|---|---|---|---|---|---|
| Behavior | Clack | 2018 | Concepts for teaching networks in CS | Java | Node-link | SCS | Anecdotal |
| | ToonTalk | 2018 | Concepts for teaching children to program | Web | Visual language | SCS | N/A |
| | Jive | 2016 | Execution traces of Java programs | Eclipse | Node-L.; Aug.src. | SCS | Anecdotal |
| | LTSView | 2017 | Transition systems | Various | 3D Node-link | SCS | N/A |
| | jGrasp | 2015 | Concepts for teaching programming in CS | Various | Aug. source code | SCS | Exp.; Survey |
| | PlanAni | 2011 | Concepts for teaching programming in CS | Various | Aug. source code | SCS | Experiment |
| | Beat | 2014 | Execution traces of Java concurrent prog. | Eclipse | Aug. source code | SCS | N/A |
| | Jive | 2007 | Execution traces of Java programs | Java | Charts | SCS | Usage Scen. |
| | Gzoltar | 2017 | Fault localization for debugging Java progs. | Java;Ecli. | Icicle; Treemap | SCS | Experiment |
| | SIFEI | 2017 | Spreadsheets formulas for testing | Excel | Visual language | SCS | Experiment |
| | GraphWorks | 2013 | Concepts for teaching graph theory in CS | Java | Anim. Node-link | SCS | N/A |
| | SwarmDebugging | 2017 | Reuse knowledge of debugging sessions | Eclipse | Node-link | SCS | Usage Scen. |
| | Jove | 2007 | Execution traces of Java programs | Java | Charts | SCS | N/A |
| | GEM | 2011 | Dynamic verification of MPI programs | Eclipse | Aug. source code | SCS | N/A |
| | ProfVis | 2011 | Execution traces of Java programs | Java | Node-link | SCS | Experiment |
| | VeldVisualizer | 2007 | Execution traces of Java programs | Java | Pixel | SCS | N/A |
| | Cerebro | 2016 | Execution traces for feature identification | Web | Node-link | SCS | Usage Scen. |
| | TiledGrace | 2015 | Programming in the Grace language | Web | Visual language | SCS | Experiment |
| | MethodExecutionReports | 2017 | Summarization of methods execution | Java | Charts | SCS | Experiment |
| | xViZiT | 2015 | Spreadsheets formulas for testing | Java | Aug. source code | SCS | Usage Scen. |
| | Synchrovis | 2013 | Execution traces of Java concurrent prog. | Java | City | SCS | Usage Scen. |
| | Dyvise | 2009 | Java heap to detect memory problems | Java | Icicle | SCS | Anecdotal |
| | TraceVis | 2007 | Execution traces based on call graphs | Java | Node-link | SCS | Usage Scen. |
| | Evolve | 2003 | Execution traces of Java programs | Java | Pixel | SCS | Usage Scen. |
| | Jsvee; Kelmu | 2016 | Concepts for teaching programming in CS | Web | Aug. source code | SCS | Usage Scen. |
| | regVIS | 2014 | Assembler control-flow of regular expr. | Windows | Visual language | SCS | Experiment |
| | ALVIS | 2006 | Concepts for teaching programming in CS | Windows | Visual language | SCS | N/A |
| Evolution | SHriMP | 2015 | Hierarchical structures in OOP | Eclipse | Node-link | SCS | N/A |
| | AGG | 2013 | Hierarchical structures in OOP | Java | Node-link | SCS | N/A |
| | CVSgrab | 2009 | Interactions during debugging | Windows | Pixel | SCS | N/A |
| | VisualCodeNavigator | 2007 | Source code changes | Windows | Aug. src.; Pixel | SCS | Usage Scen. |
| | CVSscan | 2007 | Source code changes | Windows | Pixel | SCS | Case Study |
| | MetricView | 2006 | Hierarchical structures and metrics in OOP | Windows | 3D UML | SCS | N/A |
| | DEVis | 2013 | Technical documents | Eclipse | Spiral | SCS | Theoretical |
| | ObjectEvolutionBlueprint | 2016 | Object mutations | Pharo | Charts | SCS | Experiment |
| | SoftwareEvolutionStorylines | 2010 | Developers interactions in projects | Processing | StoryLines; Charts | SCS | N/A |
| | FlaskDashboard | 2017 | Flask Python web services performance | Python | Charts; Heatmap | SCS | Usage Scen. |
| | TypeV | 2016 | Abstract syntax trees of a system's project | Web | Charts | SCS | Usage Scen. |
| | ClonEvol | 2013 | Software quality based on code clones | Windows | HEB | SCS | Usage Scen. |
| Structure | Softwarenaut | 2017 | Architecture and dependency analysis | VisualWorks | Node-L.; Treemap | SCS | N/A |
| | CodeCity | 2015 | Software quality based on code smells | Pharo | City | SCS | Usage Scen. |
| | SolidFX | 2013 | Architecture, metric and dependencies | Windows | HEB; Pixel | SCS | Experiment |
| | StenchBlossom | 2014 | Software quality based on code smells | Eclipse | Aug. source code | SCS | Experiment |
| | VisMOOS | 2010 | Software architecture of Java systems | Eclipse | Node-link | SCS | N/A |
| | CodeMetropolis | 2017 | Software quality based on metric analysis | Java | City | SCS | N/A |
| | Explen | 2017 | Slice-based techs. for large metamodels | Eclipse | UML | SCS | N/A |
| | PhysVis | 2018 | Software quality based on metric analysis | VisualStudio | 3D Node-link | I3D | Usage Scen. |
| | Rigi | 2009 | Architecture and dependency analysis | Various | Node-link | SCS | N/A |
| | SpartanRefactoring | 2018 | Automatic code refactoring for readability | Eclipse | Aug. source code | SCS | N/A |
| | Barrio | 2009 | Architecture and dependency analysis | Eclipse | Node-link | SCS | Usage Scen. |
| | Visuocode | 2014 | Navigation and composition of systems | Mac | Aug. source code | SCS | N/A |
| | ExplorViz | 2016 | Architecture based on metric analysis | Web | City | S/I | Experiment |
| | iTraceVis | 2017 | Eye movement data of code reading | Eclipse | Heatmap | SCS | Experiment |
| | SeeIT3D | 2013 | Software architecture of Java systems | Eclipse | City | SCS | Experiment |
| | Kayrebt | 2015 | Control and data flow of the Linux kernel | Linux | Node-link | SCS | Usage Scen. |
| | MetaVis | 2016 | Annotated visualization example objects | Pharo | Node-L.; Tag cloud | SCS | Usage Scen. |
| | Explora | 2015 | Software quality based on metric analysis | Pharo | Polymetric views | SCS | Usage Scen. |
| | OrionPlanning | 2015 | Arch. modularization and consistency | Pharo | Node-link | SCS | Usage Scen. |
| | VariabilityBlueprint | 2015 | Decomposition of models in FOP | Pharo | Polymetric views | SCS | Usage Scen. |
| | AspectMaps | 2013 | Architecture of aspect-oriented programs | Pharo | Iconic; Pixel | SCS | Experiment |
| | CityVR | 2017 | Architecture based on metrics in OOP | Pharo; U. | City | I3D | Experiment |
| | SolidSDD | 2014 | Software quality based on code clones | Windows | HEB | SCS | Usage Scen. |
| B.-E.-S. | Mondrian | 2018 | Execution traces of feature dependencies | Pharo | Polymetric views | SCS | N/A |
| | CHIVE | 2015 | Feature location (reconnaissance) | Eclipse | 3D Node-link | SCS | N/A |
| | Vizz3D | 2013 | Software architecture and quality | Java | 3D Node-link | SCS | N/A |
| | CodeBubbles | 2018 | Debugging within CodeBubbles | Ecli.; VS | Visual language | SCS | N/A |
| | GEF3D | 2010 | Execution traces of Java programs | Eclipse | 3D UML | SCS | Usage Scen. |
| | Graph | 2015 | Code dependencies | Pharo | Node-link | SCS | Usage Scen. |
| | Getaviz | 2018 | Developing and evaluating software vis. | Web | City+ | S/I | N/A |
| | SpiderSense | 2015 | Execution traces of Java programs | Web | Pixel; Treemap | SCS | Usage Scen. |

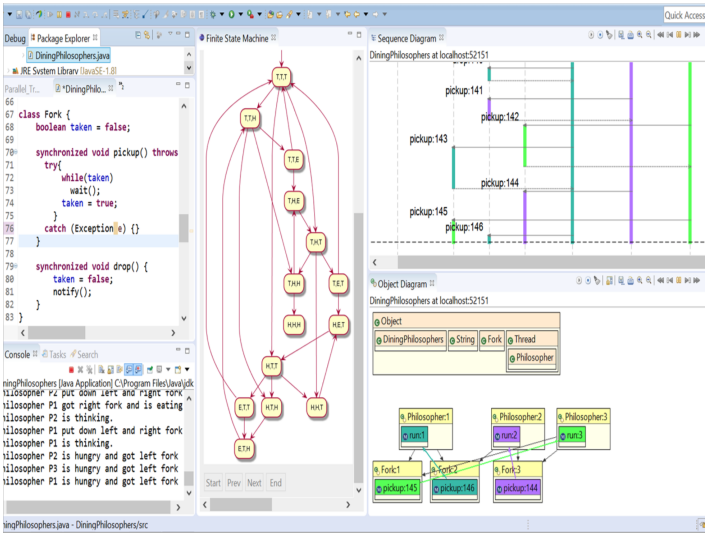



Figure 4.5: The *Jive* visualization tool to support the analysis of behavior of concurrent Java applications.

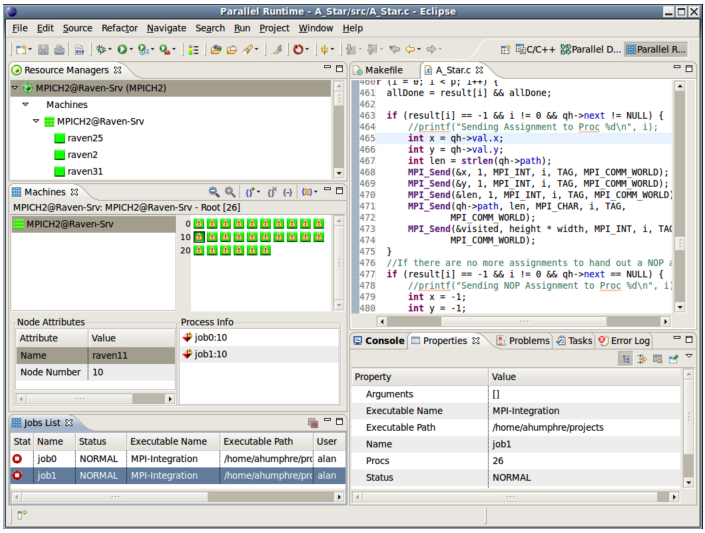



Figure 4.6: The *GEM* graphical explorer of MPI programs.

proven effective in an experiment. A few others, *Jove*, and *Veld Visualizer* have been presented only through usage scenarios. Other tools that, to our knowledge, have not been evaluated are *Jive*, *TraceVis*, and *Evolve*. Two tools, *Beat* and *Synchrovis*, target the analysis of the behavior of concurrent Java programs, while the tool *Cerebro* can be used to identify software features from the runtime data.

Three visualization tools support debugging tasks based on the visualization of program behavior. Introduced a few years ago, *Gzoltar* has shown evidence of effectiveness through an experiment. *SwarmDebugging* is an Eclipse plug-in that aims to reuse the knowledge of previous debugging sessions to recommend locations in the code to define breakpoints. Similarly, *Dyvise* supports the detection of memory problems through the visualization of the Java heap. GEM (shown in Figure 4.6) is a graphical explorer of MPI programs.

Other visualization tools deal with various particular concerns. *LTSView* is the oldest one, which is still being maintained. It supports the visualization of transition systems that model the behavior of a software system. *SIFEI* and *xViZiT* focus on the visualization of spreadsheets, while *regVis* deals with the visualization of assembler control-flow based on regular expressions. *Method Execution Reports* embeds word-size graphics in reports of method executions.

All the twenty-seven listed tools that focus on the behavior of software systems are displayed on the standard computer screen.

### Evolution

A few tools support the visualization of the evolution of hierarchical structures in object-oriented programs such as *AGG*. *SHriMP* is the oldest one, and has been maintained for twelve years. *MetricView* presents an UML class diagram in 3D that is augmented with software metrics. Others deal with various concerns. *CVSgrab* supports the visualization of the evolution of interactions of developers during debugging, while *Visual Code Navigator* and *CVSscan* (shown in Figure 4.7) focus on source code changes. *DEVis* is used to visualize the evolution of technical documents. The *Object Evolution Blueprint* deals with the evolution of object mutations. *Flask dashboard* supports the visualization of the performance of web services implemented using the Flask framework for Python. *TypeV* allows to analyze the evolution of a system through the visualization of abstract syntax

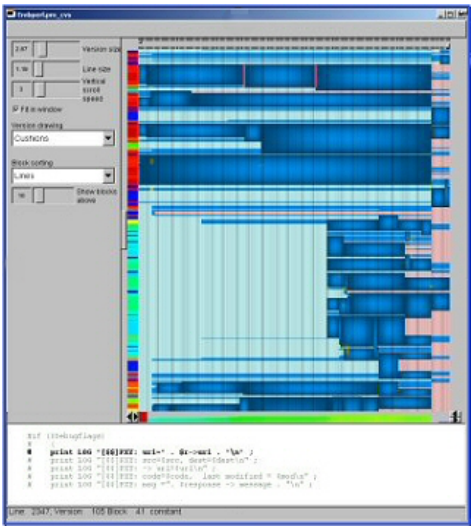


Figure 4.7: The *CVSscan* visualization tool to support the analysis of evolution for software maintenance.

trees. *ClonEvol* (shown in Figure 4.8) visualizes the evolution of code clones to improve the quality of systems.

All the twelve listed tools that focus on the evolution of software systems are displayed on the standard computer screen.

### Structure

Various other visualization tools focus on particular concerns. *MetaVis* can be used to visualize annotated software visualization example objects. *OrionPlanning* includes visualization for modularization and consistency of software projects. *Explen* supports the visualization of large metamodels. *iTraceVis* have shown evidence of been effective to investigate how developers read code through the visualization of their eye gazes. *Spartan Refactoring* allows automatic code refactoring in the editor. *Visuocode* supports the navigation and composition of software systems.

Some visualization tools are available for supporting architecture tasks such as *SeeIT3D* and *VisMOOS*. *SolidFX*, *Softwarenaut* (shown in Figure 4.9), *Rigi*, and *Barrio* are suitable for the analysis of structures and

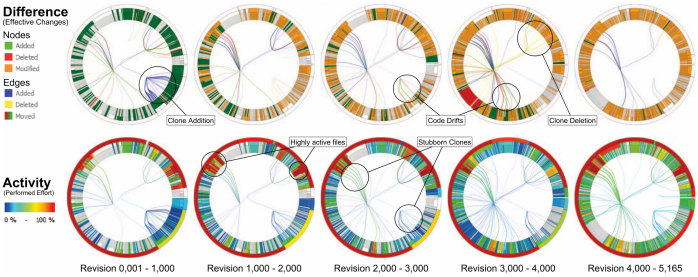



Figure 4.8: The *ClonEvol* visualization tool that help developers to analyze the evolution of code clones.

dependencies in object-oriented software systems, *AspectMaps* supports aspect-oriented programs, and *Variability blueprint* does so for feature-oriented programs.

Two tools support the visualization of the structure of software systems for the analysis of code smells. *CodeCity* (shown in Figure 4.10) visualizes software metrics based on the city metaphor, and *StenchBlossom* that augments the Eclipse source code editor with ambient visualizations.

Twenty of the listed tools that focus on the structure of software systems are displayed using the standard computer screen. Only three used immersive virtual reality: *PhysVis* in which users visualize software metrics visualized as a physical particle system, *ExplorViz* in which developers obtain an overview of the architecture of a system represented as a city, and *CityVR* which adds interactions and visualization of software metrics and smells.

### Behavior/Evolution/Structure

Eight software visualization approaches correspond to frameworks that can be used to visualize multiple aspects of software systems. Four of them correspond to active projects introduced several years ago. *Mondrian* is an engine for rapid lightweight visualization, which is currently supported in the Roassal engine [ABC+13]. *CodeBubbles* is an environment that encapsulates code snippets into bubbles that can be reused through composition. Originally available only for Eclipse, now it is also available for Visual Studio. *Vizz3D* is a framework for online configuration of 3D information

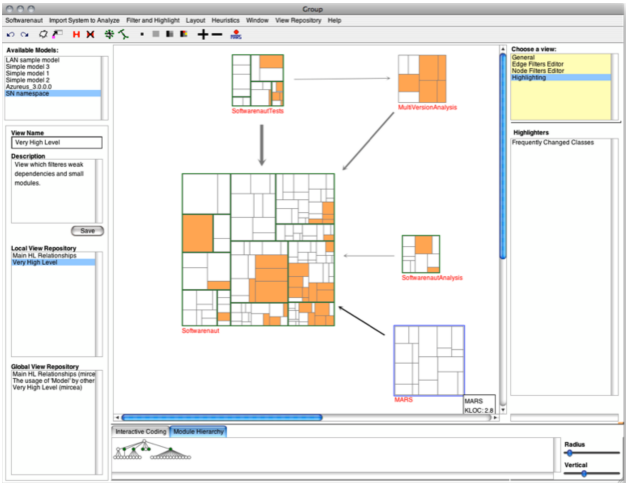



Figure 4.9: The *Softwarenaut* tool for visualization of hierarchical structures to support architecture tasks.

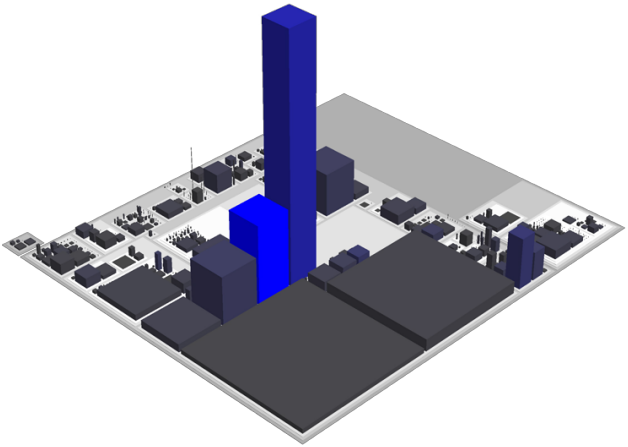


Figure 4.10: The *CodeCity* tool that visualizes the structure of software systems to support the analysis of code smells.

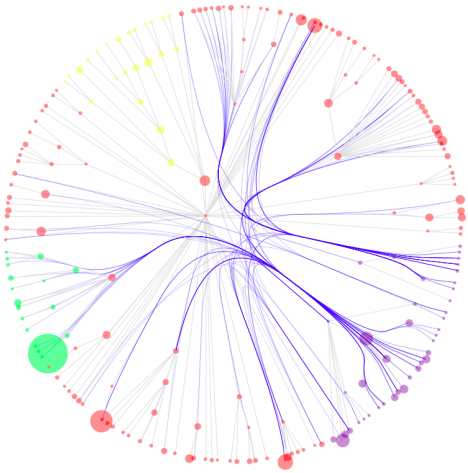


Figure 4.11: The *Graph* domain-specific language for agile prototyping of visualization of graph structures.

visualizations, while *CHIVE* is a framework for developing, in particular, 3D software visualizations.

*GEF3D* is a framework for developing 2D/2.5D/3D graphical editors. *Graph* is a domain-specific language for visualizing software dependencies as a graph (shown in Figure 4.11). *Getaviz* and *SpiderSense* enable the design, implementation, and evaluation of software visualizations.

One framework (*Getaviz*) is supports visualizations displayed on immersive virtual reality, while the seven other frameworks are limited to the standard computer screen.

### 4.3.1 Summary

In this section we have described a curated catalog of 70 actionable software visualization tools and frameworks. We have characterized the approaches, and linked the tools in a way that they can be downloaded and used.

The characterization presented in Table 4.1 contains only part of the content of our data set. We already described various other characteristics of software visualizations (Chapters 2 and 3) that can help developers willing to adopt visualization to find a suitable approach. Certainly, our data set

does not fit in a table. We observe that a much richer model is needed to make sense of the multiple characteristics of software visualizations. We believe that an *ontology* corresponds to such model.

## 4.4 Software Visualization Ontology

Ontologies are formal and explicit descriptions of concepts in a domain [Gru93]. Ontologies can help to *(i)* share common understanding of the structure of information among people or software agents, *(ii)* reuse domain knowledge, enforce domain assumptions, *(iii)* separate domain knowledge from the operational knowledge, and *(iv)* analyze domain knowledge. Through a software visualization ontology we aim to encapsulate the main characteristics of proposed software visualizations such as tasks, techniques, and media to enable both textual and visual search methods that support developers. We believe that an ontology can help developers to find suitable visualizations for their particular problems, and also it can support researchers to reflect on the software visualization domain. In this section, we elaborate on early results of designing and implementing an ontology.

### 4.4.1 Background

An ontology is a formalization of a model to describe what is essential in a *domain*. That is, the ontology describes the *concepts* in the domain, which can define various *properties* and *restrictions*. Hence, an ontology that is populated with a set of individual *instances* of the concepts is usually referred as a knowledge base. However, defining what in the domain is modeled as a concept or an instance is subjective. We opted to follow the widely used guide proposed by Noy and McGuiness [NM+01]. We now elaborate on how we addressed their suggested steps to create our *software visualization ontology*.

***Step 1.*** *Determine the domain and scope of the ontology.*

- *What is the domain that the ontology will cover?* Software visualizations
- *For what we are going to use the ontology?* To allow 1) developers find suitable visualizations for their particular concerns, and 2) researchers reflect on the software visualization domain.

- *For what types of questions the information in the ontology should provide answers?* Questions that identify particular software visualizations that fulfill the restrictions imposed by the context of the developers needs.
- *Who will use and maintain the ontology?* Software developers willing to adopt visualizations, and who have used a visualization from the ontology and want to add new supported questions to it. Also, researchers who want to add new data to the ontology for a new or an existing indexed visualization approach.

***Step 2.*** *Consider reusing existing ontologies.* To the best of our knowledge this is the first ontology of software visualizations.

***Step 3.*** *Enumerate important terms in the ontology.* We include the characteristics of software visualization and their evaluations, as well as the classifications presented in Chapters 2 and 3.

***Step 4.*** *Define the concepts and the concept hierarchy.* We opt for a bottom-up development process in which we start from instances of proposed software visualizations. For each, we identify the various concepts involved its context (*e.g.,* tasks, media, environments, frameworks, questions, evaluation strategies). We define a hierarchy of concepts following an "is-a" relation. When defining the concepts we avoid to create cycles, and validate that siblings concepts (that are at the same level in the hierarchy) correspond to the same level of generality.

***Step 5.*** *Define the properties of concepts.* We characterize the concepts based on their properties. For instance, for the concept *medium* we define the *dimensionality* (*e.g.,* 2D/3D) property. Then, when we define particular software visualizations as instances in the ontology, we can specify a medium and its dimensionality. Thus, researchers can use the ontology to investigate, for instance, the correlation between evaluation strategies and visualizations that use visualization techniques of a higher dimensionality displayed on a medium of a lower dimensionality.

***Step 6.*** *Define the restrictions of the properties.* We only use restrictions to define disjoint concepts.

***Step 7.*** *Create instances.* We create instances in the ontology for each proposed software visualization in our data set. Thus, visualization tools are the materialization of a combination of property values of concepts.

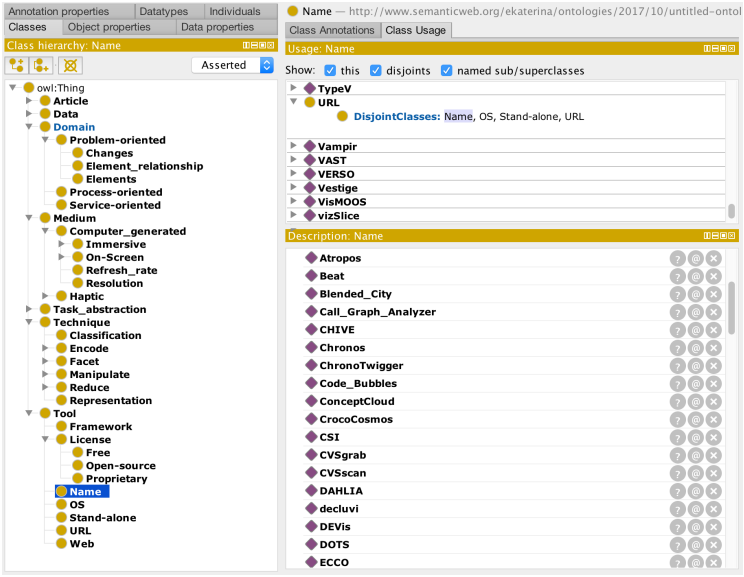



Figure 4.12: The *classes* view in Protégé showing the hierarchy of concepts. We selected the name of the tools, which are listed in the right pane.

### 4.4.2 Protégé

We implement our ontology using *Protégé* [Mus15], a popular, free, and open-source framework for the design and use of ontologies. In it, we define the concepts (in the tool called *classes*), properties, restrictions, and instances. Figure 4.12 shows the *classes* view in Protégé with a detail of the hierarchy of concepts. We selected the name of the tools' concept, which are listed in the right pane.

Figure 4.13 shows an overview of our implementation of the concepts hierarchy using the *OntoGraf* visualization plug-in included in Protégé.

We have developed an initial ontology, which we made publicly available[2]. We present some metrics of the ontology in Table 4.2. Although we consider that many more individuals and relationships must be added to the ontology to increase its usability, we observe that our current implementation is not small. A survey of ontology metrics [SRGBSA12] reported

[2] `http://scg.unibe.ch/research/vison`

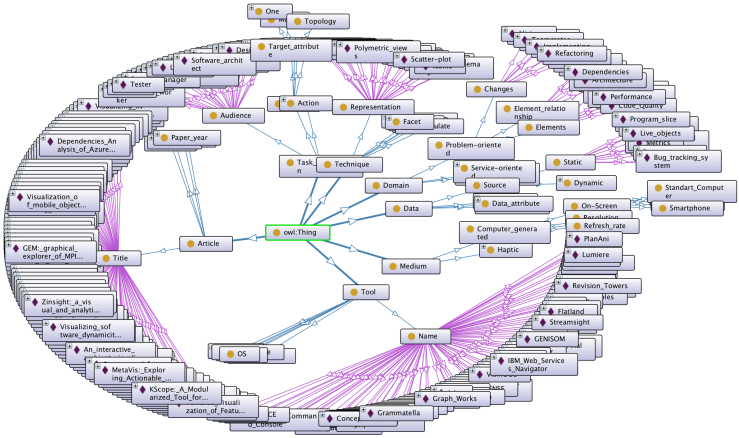



Figure 4.13: An overview of the concepts hierarchy of the software visualization ontology using the *OntoGraf* visualization plug-in.

Table 4.2: Metrics of the Software Visualization Ontology

| | | |
|---|---|---|
| Metrics | Axiom | 3290 |
| | Logical axiom count | 2428 |
| | Declaration axiom count | 862 |
| | Class count | 150 |
| | Individual property count | 20 |
| Class axioms | SubClassOf | 143 |
| | DisjointClasses | 32 |
| Object property axioms | SubObjectPropertyOf | 1 |
| | ObjectPropertyDomain | 2 |
| | ObjectPropertyRange | 3 |
| Individual axioms | ClassAssertion | 696 |
| | ObjectPropertyAssertion | 1547 |
| | NegativeObjectPropertyAssertion | 4 |

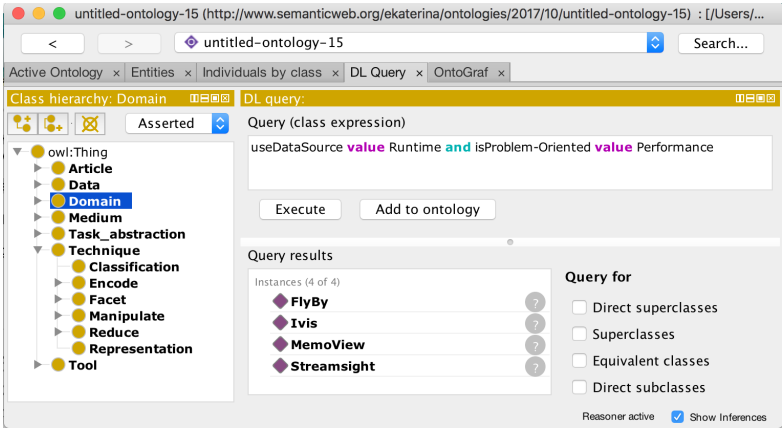



Figure 4.14: **Scenario 1.** Finding suitable visualization tools that support the analysis of *performance* issues at *runtime*.

that ontologies in average contain: *(i)* 36.11 classes (standard deviation of 78.53), and *(ii)* 28.13 instances (standard deviation of 97.59).

### 4.4.3 Usage Scenarios

We now demonstrate the ontology trough two usage scenarios.

***Scenario 1.*** *Find suitable visualization tools that support the analysis of performance issues at runtime.*

To two concepts are defined in the specification of this need: *(1)* the source of the data is the *runtime*, and *(2)* the problem dealt is the *performance* of the software system. We translate this specification to the syntax specified by the ontology web language (OWL). Figure 4.14 shows the resulting query, and the suitable tools returned.

***Scenario 2.*** *Find visualization tools under a* free *license that support the analysis of source code.*

Similarly, the specification of this need defines two concepts: *(1)* the license of the tool has to be *free*, and *(2)* the source of the data must be the *data source* of the software system. We translate this specification to the syntax specified by the ontology web language (OWL). Figure 4.15 shows

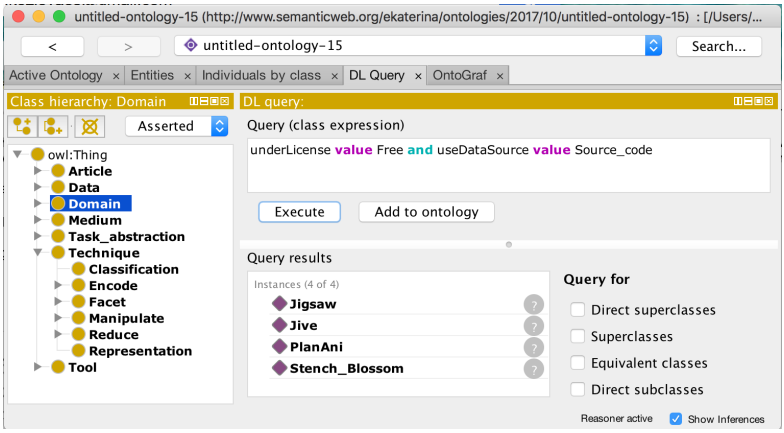



Figure 4.15: **Scenario 2.** Finding suitable free visualization tools that support the visualization of source code.

the translated specification of the need in the OWL syntax, and the suitable tools returned.

### 4.4.4 Summary

We motivated the need of a richer model to encapsulate the various characteristics of software visualizations. We argued that ontologies represent a suitable means for modeling for software visualizations. We elaborated on our experience when designing an ontology for software visualizations. We discussed our implementation of the ontology in the Protégé tool. Then, we demonstrated how the ontology can be used through usage scenarios. We made the ontology publicly available[2]. We expect the ontology will help developers to find suitable software visualizations, and researchers to reflect on the field. Users of the ontology will be able to contribute, for instance, by adding new proposed visualizations, or adding the results of evaluations of existing visualizations. In the future we plan to combine the previously described meta-visualization approach to our software visualization ontology, which we discuss further, amongst other future work, in the following chapter.

## 4.5 Conclusion

Although many software visualization approaches have been proposed to deal with various software concerns, usually developers are not aware of tools that they can put into action. In this chapter, we have presented our attempts to fill the gap between existing software visualizations and their practical applications: *(1)* we introduced a meta-visualization approach of live visualization example objects that are annotated with the type of development questions that they can help to investigate. In the visualization, developers can identify suitable visualization examples by detecting the surrounding keywords in the tag-iconic cloud-based visualization; *(2)* we presented a curated catalog of 70 actionable software visualization tools that we linked to their repositories. We classified the tools into various categories (*e.g.,* task, data, environment) to help developers who look for suitable visualizations; *(3)* we summarized early results in developing a software visualization ontology.

The analysis of our software visualization catalog shows that the city metaphor is a common visualization technique. Although some studies have shown that software cities are effective to support comprehension tasks, we observe that their evaluations have focused on traditional aspects of user performance (*i.e.,* completion time, correctness). We ask how the effectiveness of city visualizations can be increased. Moreover, we argue that to assess the power of such visualizations to support communication and discovery, the evaluations should not only include other variables of user performance, but also they should cover the assessment of user experience. In the following chapter, we introduce a prototype visualization tool, and explore such aspects via a formative experiment.

# 5

# Gameful Software Visualization

## 5.1 Introduction

We have observed that a medium such as immersive virtual reality has almost never been used to display software visualizations. We have argued that some visualization techniques when displayed in a different medium might increase their effectiveness. We have also discussed the need of expanding the traditional time and correctness variables (profusely used in software engineering tools) to others such as recollection, and engagement to have an appropriate evaluation of software visualization tools. We believe such variables might shed light on the value of visualization for communication. To decode the message within a visualization (to collect insights of the system) developers need to be willing to spend time on it. We observe that *gamification* have been used to study such concerns.

Indeed, the gamification of software engineering tasks (*i.e.,* applying computer game elements and design techniques) improves developer engagement [DT13]. Most approaches in software engineering, however, have struggled when putting the concept into action and applied only simple gamification mechanisms such as points and badges [PGBP15].

The three main concepts that promote engagement in computer games are *curiosity*, *challenge* and *fantasy* [Mal80]. We observe that developers commonly associate the first two concepts with software visualizations [MFB+17]. The third concept, defined as "an illusory appearance"[1], is also inherent to visualizations. Therefore, we believe software visualizations can be more effective than previous approaches to gamify software engineering tasks. However, we observe that not all software visualization tools promote developer engagement equally.

The medium, technique and interaction are architectural choices in developing software visualizations that can play a role in enhancing user engagement. Consequently, we formulated the following research question:

*RQ) How can architectural design choices in developing software visualization tools enhance developer engagement?*

We argue, to maximize engagement we need gameful software visualization (*i.e.,* visualization that provides developers with an interface analogous to computer games). We examine our research question by focusing on software comprehension tasks. We designed *CityVR* —an interactive visualization tool that implements the city metaphor technique displayed in immersive virtual reality (I3D) to boost developer engagement in software comprehension tasks (shown in Figure 5.1). We investigated the effectiveness of CityVR via a formative experiment in which developers visualized *ArgoUML*, a UML diagramming framework. We measured engagement in terms of experienced feelings, interaction, and time perception. We report how our design choices relate to developer engagement. We found that developers *(i)* felt *curious*, *immersed*, *in control*, *excited*, and *challenged*, *(ii)* spent considerable interaction time navigating and selecting elements, and *(iii)* perceived that time passed faster than in reality, and therefore were willing to spend more time using the tool to solve software engineering tasks.

Only a few software visualization tools have used virtual reality for software comprehension tasks. FileVis [YM98] implements a glyph-based visualization, and Software World [KM99] uses the city metaphor technique. As opposed to CityVR, both tools use a standard computer screen as the medium to display the visualization (probably due to the limited tech-

[1] "fantasy | phantasy, n." OED Online. Oxford University Press, March 2017. Web. 6 April 2017.

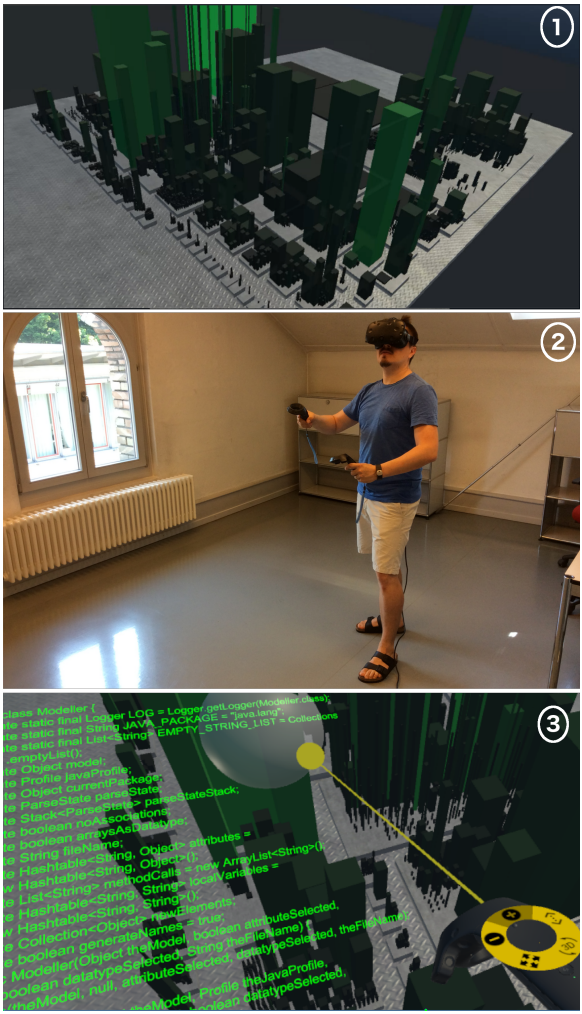



Figure 5.1: *CityVR* gameful software visualization for comprehension. ① the city metaphor *technique*; ② a developer in *immersive virtual reality* (I3D) medium; ③ developers *interact* with elements using a controller and a bubble.

nology available at the time). Two other studies have proposed software visualization using other media. *Imsovision* [MLMD01] allows developers to visualize software using the CAVE medium. Recently, Fittkau *et al.* [FKH15b] evaluated the visualization of software cities using the Oculus Rift device. We observe that neither do existing studies elaborate on the methodological principles that support architectural choices in developing visualizations, nor do they discuss the impact of the design decisions in developer engagement, but they limit their analysis to performance.

Little research has proposed gamification of software related tasks based on visualization tools. Two software visualization tools have been proposed for teaching software engineering: *(1)* CodeSmellExplorer [Raa12] helps students to recognize code smells by interacting with a 2D graph network visualization displayed on a tabletop. In the tool students are challenged to connect physical cards (each listing a given code smell) to refactorings; *(2)* Sort Attack [YP15] implements a 2D visualization based on a standard computer screen, in which a number of game techniques such as lives, levels and time help students to learn algorithms. CodeMetropolis [BGBG16] supports developers in comprehension of test suites through an enriched software city visualization, implemented using the Minecraft game engine, displayed on a standard computer screen.

Instead, CityVR is designed to boost developer engagement during visualization for software comprehension by considering the impact of architectural design choices such as the selected technique, interaction and medium. As opposed to CodeCity [WLR11], which offers interactions based on a computer screen setup, in CityVR developers interact with visualizations by *(i)* moving across the available physical space, and *(ii)* selecting classes using controllers with in their hands.

## 5.2 CityVR Overview

CityVR is an interactive software visualization tool that is displayed in an I3D medium. We selected I3D as the medium for our visualization since it promotes engagement [BC04]. CityVR allows developers to obtain an overview of the software system while they are immersed in it. We use our taxonomy introduced in Chapter 2 to characterize our visualization tool. We first introduce the three dimensions that relate to the problem domain (*i.e.,* audience, task and data) and then the two that relate to our solution:

representation and medium. We split the representation dimension into technique and interaction.

CityVR targets the software maintainer *audience* who has to perform software comprehension *tasks* in order to correct and evolve software systems. The *data* available are the source code of software systems. We discuss the architectural design choices that we made to boost developer engagement in comprehension tasks.

### 5.2.1 Design

*Medium*. In Chapter 2 we characterized software visualizations and found that most visualizations are designed to be displayed on the standard computer screen. We observe that the complexity of software comprehension tasks requires developers concentration, which developers usually pursue by using large screens. Developers also typically boost their concentration by isolating themselves from ambient noise with headphones. We observe that I3D offers developers a more complete immersion, which could help to increase the effectiveness of visualizations. In consequence, we selected I3D as the medium to display our visualization.

*Technique*. We selected the city metaphor technique as it has not only proven to be effective to support developers in software comprehension tasks [WLR11], but is also present in popular games such as SimCity, and Grand Theft Auto. Figure 5.1 ① shows a software city. Each building in the city represents a class in the system, and the districts represent software packages. The technique can encode three metrics: one in the square base of buildings (*i.e.,* width and depth), one in the height, and one in the color of buildings.

A software city has the advantage that it provides orientation to developers. Developers can quickly understand the metaphor, and refer to the classes in the software system by their location in the city (*e.g.,* in the north). Also, once defined a configuration to map the set of metrics to the visual properties of buildings, a particular topology in the city is unveiled. Some classes are going to be represented by skyscrapers, while others will be represented by flat buildings. The density of the buildings eases comparison, and promotes the identification of patterns. In that way, developers identify visual patterns that they relate to metric values, and obtain insight of the quality of the software (*i.e.,* alert on well used design patterns, and bad code smells). For instance, a *god class* that contains many lines of code and many methods can stand out when represented as a

massive colored building. Also, a long *facade class* that contains only a few lines of code but it contains many methods can be easily identified when represented as thin and tall building. Certainly, the choice of how mapping the metrics to the properties of buildings has an effect in the usability of the visualization, that we address through pilot experiments.

*Interaction*. We observe that developers spend long hours sitting in a chair or standing in front of a computer screen. The lack of movement during programming sessions has a negative impact on their daily experience. We conjecture that an environment that encourage developers to freely navigate the system (*e.g.,* walk, crouch, jump) without having to stop software comprehension tasks can improve their engagement [AMN+10]. In CityVR developers interact with the visualization through navigation and selection. We scaled the visualization of the software system to fit the physical space available in the room where the tool is used. In CityVR developers can select classes using two controllers with their hands. Figure 5.1 ③ shows a developer who finds a class of interest, and uses one of the controllers to create a *bubble* (pointed by the yellow beam). Then, he drags the bubble with the other controller and drop it in a building to inspect the source code of the represented class. In that way, developers can analyze metric values and inspect source code to get a better understanding of a particular artifact. The source code and metric values are displayed in a panel attached to one of the controllers, so developers can move it with their hand. Developers can also scroll through the code using the buttons on one of the controllers (+ and - buttons).

### 5.2.2 Workflow

To obtain in a short time a working application that we could use to test our hypotheses, we opted for exporting the models of the city visualizations from an existing implementation for the Moose platform [NDG05]. Moose is a data analysis platform, based on the Pharo [DZHC17] programming environment. Moose offers multiple features the analysis of a model of a software system. The model of a system contains essential information of it such as classes, and dependencies. Thus, the model can be used to build interactive visualizations. We opted for using system's models in the Moose default format: MSE. MSE is similar to other data formats such as XML, JSON. The main difference to other formats is that MSE use parentheses to define elements, which makes it suitable for manipulating

large data sets. There are several exporters for creating MSE models from system's source code.

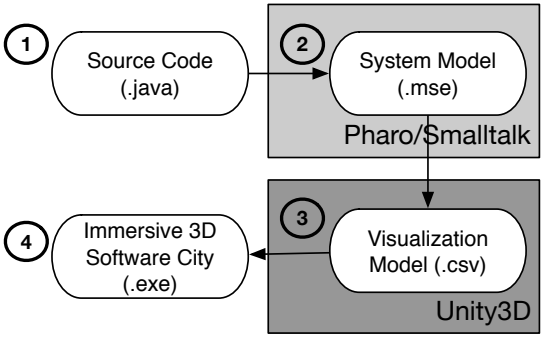



Figure 5.2: The four step workflow implemented in CityVR. From system's source code to an immersive 3D city visualization.

Figure 5.2 shows the four step workflow that developers have to follow when using CityVR: *(1) Source code*: a developer creates a model of the system by passing source code files (*i.e.,* Java and C/C++) as input to an MSE exporter (*e.g.,* VerveinJ, inFAMIX, jdt2famix). The output model (.mse file) contains the main characteristics of the system such as inheritance, dependency, and metrics that can be used for analysis. *(2) System model*: using CodeCity[2] for the Moose 5[3] platform, developers configure a city visualization by defining the mapping between a set of system properties to the dimensions available in the visualization technique. Once developers are happy with the visualization displayed on-screen, they can export the model of the visualization (.csv file). *(3) Visualization model*: the model contains the selected properties of the system to be visualized and the layout of the buildings of the city. Developers use the visualization model (*i.e.,* .csv file) as input to CityVR in Unity3D 5.5[4]. They can adjust parameters such as the size of the room and compile the application (.exe file). Finally, *(4) Immersive 3D Software City*: developers can use the visualized system to solve their tasks. We used an HTC Vive VR headset with a 2160 x 1200 combined resolution, 90 Hz refresh rate and 110° field of view. We selected the HTC Vive since it includes the highest number

[2] `http://smalltalkhub.com/#!/~RichardWettel/CodeCity`
[3] `http://www.moosetechnology.org/`
[4] `https://unity3d.com/`

of sensors (among similar devices). Since our interest is to analyze user engagement, we consider these sensors to be useful.

More implementation details are available on the CityVR web site[5].

## 5.3 Formative Experiment

We investigated the effectiveness of CityVR based on a formative experiment. We configured CityVR to visualize the ArgoUML v.0.34[6] system (as shown in Figure 5.1). In the software city three metrics are encoded in the properties of buildings, namely the *number of lines of code* (NLOC), the *number of methods* (NOM), and the *number of attributes* (NOA), which are mapped to their color (using a linear transformation), height, and width/depth, respectively. We invited six participants to explore CityVR, and we subsequently conducted semi-structured interviews. They were not paid and freely opted to participate in the study. All of them were experienced developers (*i.e.,* 6.5 ± 1.5 years), and all have an academic background in computer science (*i.e.,* one bachelor, four PhD, and one post-doc). We selected participants with some experience using software visualizations (their self-reported experience ranged between 2 and 5 in a 5-step Likert scale). Participants neither had prior experience using I3D, nor did they have knowledge of the implementation details of ArgoUML. After explaining the encoding used in the visualization and the interactions available, we asked participants to complete two comprehension tasks. The tasks are as follows:

*(T.1) How well is ArgoUML designed (*e.g., *patterns/smells)?*

*(T.2) What is the semantics of each package?*

Participants found several code smells such as a bright and massive *god class*, several thin and long *facade* classes, and a few large and flat *data* classes. The code inspection also revealed some of the semantics of packages hidden in the source code. For example, a *configuration* package that contains three data classes with parameters required by various components of the system. Participants also identified a package that contains the implementation of the graphical interface of the system.

[5] `http://scg.unibe.ch/research/cityvr`
[6] `https://sourceforge.net/projects/argouml/`

These rather difficult tasks were not designed to evaluate the effectiveness of the software city technique but to *stress* navigation and interaction, thus allowing an evaluation of the engagement of participants. We measured engagement in terms of *(i)* interaction (*i.e.,* movement), *(ii)* feelings, and *(iii)* time perception.

**Interaction** We observed that the more participants engaged, the more they interacted. We analyze participants' engagement by measuring their movement across the physical space. We instrumented our tool to record the position of participants during the visualization of the system. The results of each participant are shown in a separate chart in Figure 5.3. The marks in the chart represent the position of participants. We observe that participants feel oriented using the visualization and adopt various strategies to navigate the city *(i) Center view*. Some participants *[P2]* and *[P6]* opted to explore the city starting from its center. *(ii) Diagonal view*. One participant *[P1]* preferred to stand in the empty corners of the city to obtain an overview. *(iii) Omnidirectional view*. Most participants (*i.e., [P3]*, *[P4]*, and *[P5]*) felt free to explore the particularities of the city by using most of the space available in the physical room. Certainly, navigation alone is not a measure of engagement by itself. Users could move for other reasons without engaging in the activity. However, we believe the combination of objective measures such as navigation with subjective ones such as feelings and time perception do expose their engagement.

**Feelings** When participants were using the visualized system, we asked them to share the feelings. Participant found it "nice to walk" across the system, and felt that it was fun to interact with the system using the arms and the whole body. We also asked participants to identify their feelings when they finished the tasks. We asked them to select the top five strongest feelings from a list of twenty words (proposed to describe gaming experiences [GCC+10]). Frequent feelings were *curious*, *immersed*, *in control*, *excited*, and *challenged*.

**Time perception** The subjective perception of the passage of time changes according to the engagement of users [BBW09]. When users engage with a task, they tend to lose track of time [JCC+08]. Therefore, at various moments of the interview, we asked participants to report how much time they perceived had passed. We asked the first two participants to estimate the time when 10 minutes had passed, and both were correct. We noticed that the time was too little and that participants tend to answer rounded numbers. Therefore, we decided to ask the next four participants to estimate the time when 42 minutes had passed. Three of them perceived

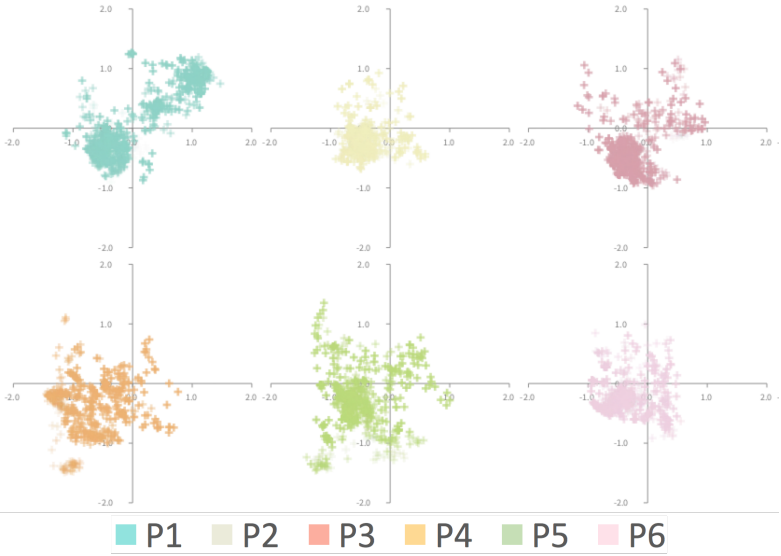



Figure 5.3: Scatterplots that map the location of participants as they move across the physical room during the visualization of ArgoUML.

that 30 minutes had passed, while one participant was much closer and estimated that 40 minutes had passed. We observe that even though participants moderately underestimated the passage of time, they *felt* that time passed fast. One participant said that "time had flown very fast"

## 5.4 Discussion

We revisit our research question to discuss the effectiveness of decisions made during the design of CityVR. Some decisions such as the selected technique, medium and navigation seem very effective. However, others such as selection and inspection produced mixed results. It seems that interaction is the architectural choice that offers the most room for improving engagement.

*Effective choices: medium, technique, and navigation*. We observe that the city metaphor technique fits well to I3D. By scaling the software city visualization to the physically available space, developers can navigate the

system by walking, which eases navigation (compared to the traditional navigation in computer screens that uses mouse and keyboard). Participants required little training before they felt comfortable with the type of navigation and medium, and were excited to use I3D.

*Limited effect choices: code inspection*. The panel attached to the right controller served to inspect the source code of a selected class. Developers were able to scroll through the file by waving their arm and by pushing two buttons placed in the controller. Although most participants described this interaction using terms such as *"appealing"*, *"futuristic"*, and *"novel"*, one participant felt that the code was hard to read because it depended on his ability to maintain his arm steady. That participant suggested that having a large fixed panel would ease code reading. Other participant who felt fatigued, noticed high latency of the view when inspecting large source code. Participants seemed happy to interact with source code freely in the 3D space. A participant said *"this is the first time that I actually see the whole code of a class that large"* That participant also observed that seeing the whole code of classes made it easier to understand when a class has too many lines of code and needs refactoring.

*Ineffective choices*. In CityVR developers select classes for inspection using a bubble, which they create and drag to buildings to obtain details-on-demand of represented classes. However, we observed that developers found it difficult to use. Sometimes developers forgot the mechanism to create and drag the bubble. Other times they lost the bubble inside buildings and had to create a new one.

We observe that extending CityVR to other tasks (*e.g.,* testing, debugging) would require to mitigate the cost of forcing developers to leave their IDE. One solution to that problem would be to transfer the whole IDE to the I3D medium. We also observe that even though the isolating effect of I3D can help developers to concentrate, it could also introduce a social debt. We think that collaborative visualization could mitigate that effect. We envision developers in remote locations using virtual reality as well as co-located developers using augmented reality for visualizing systems collaboratively.

## 5.5 Conclusion

We introduced *CityVR* —a tool that implements the software city technique using immersive virtual reality. Through a formative experiment

we analyzed how developers engage with the tool. We found that developers *(i)* felt *curious*, *immersed*, *in control*, *excited*, and *challenged*, *(ii)* spent considerable interaction time navigating and selecting elements, and *(iii)* perceived that time passed faster than in reality, and therefore were willing to spend more time using the tool to solve software comprehension tasks.

Although the results of the experiment are promising, we cannot claim that the city visualization displayed in immersive virtual reality is more effective than when displayed in other media, neither we can identify particular tasks that are boosted by the medium. We investigate such aspects via a controlled experiment that we present in the following chapter.

# 6 The Medium

## 6.1 Introduction

The results of the formative experiment have shown that immersive virtual reality promotes engagement amongst developers when visualizing software systems. In the experiment, participants were able to obtain relevant insights from the system. They felt curious and challenged, and interacted with the visualization to obtain details of the components of the system, while they were willing to spend a long time using the visualization. However, to ascertain the impact of such a medium on the effectiveness of the visualization, a thorough evaluation is required. In the following, we expand on the motivation and describe such evaluation.

When designing visualizations, multiple attributes must be taken into account such as the supported *task* (*e.g.,* software comprehension) and the visualization *technique* (*e.g.,* 3D software cities). Amongst these attributes there is also the display *medium* (*e.g.,* computer screen) on which visualizations are designed to be rendered. The medium has been considered as an attribute in foundational software visualization taxonomies. Roman and Cox [RC93] identified new capabilities offered by emerging computer-based visualizations as opposed to traditional visualizations in


Figure 6.1: Participants visualize software cities for software comprehension tasks using various media. We evaluated how the effectiveness is affected by the medium: ① immersive virtual reality, ② a physical 3D printed model, and ③ a standard computer screen.

paper. Price *et al.* [PBS93] observed that while computer-based visual-

izations can be designed for one medium, they can often be transfered to another. A decade later, Maletic *et al.* [MMC02] envisioned a future in which software visualizations would take advantage of multiple media.

In Chapter 2 we characterized software visualizations using the medium amongst other attributes. Amongst other insights we found that the standard computer screen (SCS) remains the most frequently used medium to render software visualizations. Other media used in a few software visualizations were immersive virtual reality (I3D) [ERS+14], physical 3D printed models (P3D) [FKH15a], large multi-touch tables [AMNB13], and wall-displays [AMN+10]. Nevertheless, the impact of the medium amongst these visualizations is not clear.

We investigate to what degree the choice of a medium affects the effectiveness of visualizations. We consider effective visualizations to be those that excel at: (1) performance (*i.e.,* completion time and correctness), (2) recollection (*i.e.,* recollection of recent events), and (3) user experience (*i.e.,* feelings and difficulties). Consequently, we formulated the following research questions:

*RQ.1)* How does using different media for a software visualization technique affect *completion time and correctness?*

*RQ.2)* How does using different media for a software visualization technique affect *recollection of recent events?*

*RQ.3)* How does using different media for a software visualization technique affect *user experience?*

We investigated these questions via a controlled user experiment. In the experiment we focused on *software comprehension*. That is, the cognitive process in which developers learn about a software artifact to accomplish a task [CDPC11], and the *3D city visualization technique*, which *(i)* has proven to be effective to support software comprehension tasks [WLR11], *(ii)* is available for various media [FKH15b, FKH15a], and *(iii)* is easily transferable from one medium to another.

We selected media used in software visualizations that take different approaches to interaction (*i.e.,* SCS, I3D, P3D) (shown in Figure 6.1). We formulated a set of nine software comprehension tasks inspired by those used in previous studies [WLR11, FKH15c, AMN+10], and we selected a set of open-source software subject systems of various sizes. For each medium we conducted interviews with between-subject groups of nine

developers (*i.e.,* twenty-seven participants in total) to collect data that helped us to answer our research questions.

We found that even though developers using P3D required the least time to identify outliers, they experienced the least difficulty when visualizing systems based on SCS. Moreover, developers using I3D obtained the highest recollection.

The remainder of the chapter is structured as follows: Section 6.2 outlines related work and discusses the need for extended benchmark properties that explicitly include the medium to produce comparable evaluations of software visualizations. Section 6.3 describes the controlled user experiment conducted to evaluate the impact of the medium in the effectiveness of visualizations. Section 6.4 elaborates on the quantitative analysis of the results of the evaluation. Section 6.5 presents a qualitative discussion of the results and describe the threats to validity of our findings. Section 6.7 concludes the chapter.

## 6.2 Related Work

The medium has been identified as an important characteristic in the software visualization community. Price *et al.* [PBS93] proposed a software visualization taxonomy that includes the medium as a dimension. They observed that a primary target medium must be identified for visualizations that eventually could be transfered across multiple media. Maletic *et al.* [MMC02] proposed a complementary taxonomy that also includes the medium as one of the five dimensions that characterize software visualizations. Although these foundational taxonomies have been present in the software visualization community, the medium has not been a main concern among most proposed visualizations.

We now elaborate on related work of the 3D software visualization technique that we use in our experiment.

A review of 3D software visualization was presented by Teyseyre and Campo [TC09]. They classified twenty-two visualization tools based on their expected audience, data source, presentation, interaction, evaluation, and framework used. They observed that the medium plays a key role in the effectiveness of software visualizations. However, all tools included in the overview were designed for one medium (*i.e.,* SCS), and consequently they did not include it as a classification criterion.

3D city visualizations have been proposed extensively to support software comprehension. Knight and Munro [KM00], proposed a visualization that implements the city metaphor to support program comprehension. They observed that virtual reality provides developers orientation when exploring code artifacts. Wettel and Lanza [WL07b] stated that software cities provide developers a physical space with strong orientation points. Panas *et al.* [PEQ+07] proposed visualization to support multiple comprehension tasks using the city metaphor since it helps users to better understand complex situations. However, none of them elaborated on why they decided to use the SCS medium.

Software visualization based on I3D is not new. Maletic *et al.* [MLMD01] proposed an immersive object-oriented software visualization system for comprehension using a *CAVE* setup. Recently, Fittkau *et al.* [FKH15b, FKH17] evaluated the visualization of software cities using the *Oculus Rift* for software comprehension tasks. However, none of them elaborate on the grounds that supported their selected medium.

A few visualizations have proposed P3D as their medium. Huron *et al.* [HCT+14] proposed constructive visualization as a paradigm for simple creation of flexible and dynamic visualizations (*e.g.,* using Lego bricks). Fittkau *et al.* [FKH15a] used a physical 3D printed model of a software city that they compared to visualization in a computer screen. Their evaluation showed little differences between the performance of visualizations displayed on SCS versus P3D. In this work, we study two systems of different size. We not only compare P3D versus SCS, but include I3D. Finally, besides evaluating performance, we also evaluate recollection and user experience, since we believe that software comprehension can benefit from both.

In summary, we observe that even though research in software visualization has spanned various media, little has been done to support developers who are willing to use visualization, to choose the most effective medium for their particular task. Therefore, our interest is to study the impact of the medium in the effectiveness of 3D software visualizations.

## 6.3 Controlled User Experiment

We performed a controlled user experiment that evaluates the impact of the medium in the effectiveness of 3D software cities for comprehension tasks. Now we elaborate on the design of our experiment.

### 6.3.1 Experiment Design

The purpose of our experiment is to evaluate the impact of the *medium* (independent variable) in the effectiveness of software visualizations by comparing *performance*, *recollection* and *experience* (dependent variables). The *performance* of participants was measured in terms of completion time and accuracy. To measure *recollection*, we asked participants in the last part of the session to draw what they remembered of the visualization of the second system (approximately twenty minutes after). Finally, to measure *user experience (i)* during the visualization of each system, participants were asked to score the difficulty of the tasks, and *(ii)* at the end of the visualization of each system participants were asked to identify their top ten experienced feelings (sorted by intensity).

We decided to use between-groups of nine participants. That is, the participants of each group visualize the three systems (listed in Table 6.2) one-by-one solving the nine tasks (listed in Table 6.1) in one medium. We ran four pilot studies and analyzed their outcome. We tried various configurations of the parameters of the visualization technique and selected the one that performed better for navigation and comparison. We fine-tuned the tasks, so the experiment would last around one hour (to avoid fatigue).

When designing our experiment, we noticed that there is a need for a standard protocol to compare evaluations of software visualizations. We observed that Maletic and Marcus [MM03] issued a call-for-benchmarks towards standardizing the evaluation of software visualizations. They proposed four properties that characterize visualizations for benchmarks: *task*, *data set*, *evaluation* and *interaction*. We observe that a developer willing to adopt a visualization technique that is available in various media cannot compare the results of isolated evaluations of visualizations that not only differ in the technique but also in the display medium, thus possibly leading to misleading results. Thus, the need of a standard protocol to compare evaluations of software visualizations that includes the medium explicitly. Consequently, we propose to add explicitly two properties to these benchmarks: *medium* and visualization *technique*. In this way, benchmarks support not only researchers who compare new visualization techniques, but also those who evaluate visualizations across multiple media (as is our goal).

*Extended Benchmark Properties:* We first describe our proposal for the two new added properties (*i.e.,* medium and technique) and then for

each of the three original properties (*i.e.,* interaction, task, and data set) of benchmarks.

**Medium.** Amongst the media used in software visualizations we find immersive virtual reality, physical 3D models, wall displays, multi-touch tables, and standard computer screens [MGN16a]. Since the last three use the screen to display visualizations, we propose the media used in the following setups to conduct the experiment:

*(i) Standard Computer Screen (SCS).* We used an Apple MacBook Pro with a resolution of 1440 x 900 pixels. The visualizations were provided by the CodeCity[1] implementation for Moose 5 on OSX.[2]

*(ii) Immersive Virtual Reality (I3D).* We used an HTC Vive VR Headset with a 2160 x 1200 combined resolution, 90 Hz refresh rate and 110° field of view. We implemented a custom visualization using Unity 5.5 based on models of the cities exported from CodeCity.

*(iii) Physical 3D model (P3D).* We used a Form 2 3D printer by formlabs[3] based on stereolithography (SLA) technology. To implement the visualizations, we exported them from their implementation in Unity (used for I3D) to the Stereo Lithography (STL) format required by the printer using the *pb_Stl*[4] library.

**Technique.** In Chapter 2 we identified sixty-four visualization tools that implement various visualization techniques. We selected from them a visualization technique based on the following criteria: *(C1)* proven effective for software comprehension tasks, *(C2)* suitable for the capabilities of the selected media, and *(C3)* implementations or source code are available. We focused on the most restrictive criterion, namely C2. In the process of selecting a suitable technique we rejected visualizations that: *(i)* support tasks that do not focus on software comprehension, such as Vizz3D [PLL05], or *(ii)* neither provide implementations for all media, such as TraceCrawler [GDG06], nor make their source code publicly available, such as MetricView [TLTC05]. Instead, we observed that 3D city visualizations fulfill all these criteria. Firstly, software cities have proven effective to solve software comprehension tasks

[1] `http://smalltalkhub.com/#!/~RichardWettel/CodeCity`
[2] `http://www.moosetechnology.org/`
[3] `https://formlabs.com/3d-printers/form-2/`
[4] `https://github.com/karl-/pb_Stl`

in terms of performance [WLR11], recollection [ITW01], and user experience [FKH15c] (C1). Secondly, they have proven to be suitable for SCS [Wet10], I3D [FKH15b], and P3D [FKH15a] (C2). Finally, even though we did not find implementations available for all media, the simple design of software cities based on colored cubes and the availability of source code enables their implementation to be easily transferred from one medium to another (C3).

Figure 6.2 shows *CodeCity* [WLR11], a well-known implementation of 3D software cities for SCS. In this visualization metaphor, buildings in the city represent classes in the software. Contiguous buildings in a district represent the classes that belong to a package. The visualization allows developers to analyze software metrics and identify potential design problems such as *god classes*. We configure the visualization in such a way that the height of a building encodes the number of methods (NOM) of the represented class, the size of the square base of a building represents the number of attributes (NOA), and the color encodes the number of lines of code (NLOC). We use a linear scale of five different tones of green as proposed by the ColorCAT [MJSK15] tool for visualizations that support comparison tasks on continuous data. The brighter the color, the higher the value of the metric.

**Interaction.** We confined the interaction to those that are common to all media. Consequently, since P3D does not support selection, the interaction provided to participants in all media was limited to navigation (*e.g.,* rotate, pan, zoom).

**Tasks.** We assume developers who want to contribute to an open-source object-oriented software system need to collect class candidates for potential refactoring. To accomplish this high-level task, they usually define nine specific sub-tasks (listed in Table 6.1) that they have to solve. The visualization helps developers to obtain an overview of the whole software system and spot refactoring candidates.

When developers obtain an overview of a software system, they are able to (1) spot outliers, (2) detect patterns, and (3) quantify elements [ANMT09]. Although some of these tasks can be addressed faster and eventually with more accuracy by other approaches, visualizations enable developers to combine all of them at once. We were inspired by a previous evaluation of CodeCity [WLR11] to design our tasks. We focused on two criteria to select the tasks: *(i)* they can be solved in a reasonable amount of time (*e.g.,* < 5 minutes), and *(ii)* the only interaction needed to solve them is navigation. For each medium (*i.e.,* SCS, I3D, P3D) a different

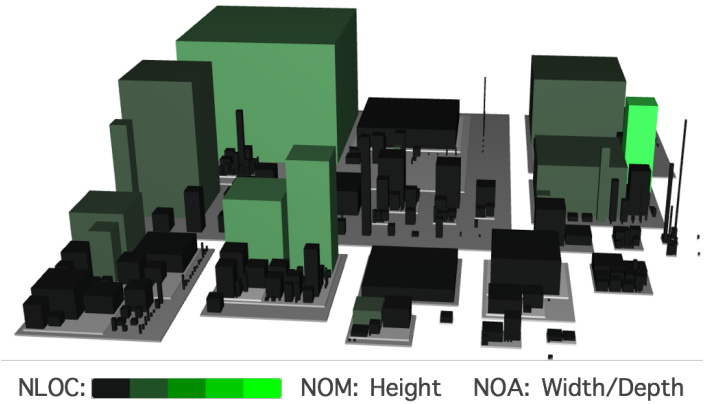



Figure 6.2: Freemind 2.0.9 is the medium size system used in the experiment. The system is visualized as a software city where buildings represent the classes of the system, and districts represent packages. Three software metrics are mapped to attributes of buildings: number of lines of code to the color, number of methods to the height, and number of attributes to the width/depth.

group of participants visualize one at a time the systems (shown in Table 6.2) and solve the tasks (shown in Table 6.1). The tasks are grouped by themes. Tasks *T1-T3* require metric analysis to *find outliers*. Tasks *T4-T6* concern the detection of potential design problems by *identifying visual patterns*. Finally, Tasks *T7-T9* concern *location and quantification*.

**Data set.** We looked for a collection of real-world open source software systems of diverse size. We observed that the *Qualitas Corpus* [TAD$^+$10] fulfills these criteria. We selected three systems (from the Qualitas Corpus) of various sizes that have been used extensively in other studies (shown in Table 6.2).

### 6.3.2 Hypotheses

We hypothesized that the most common medium used in software visualizations, the standard computer screen, is an effective medium. Since the computer display is the main medium used during development, we envisage that interacting with visualizations displayed on the computer screen with a mouse and keyboard will not pose difficulties. We therefore

Table 6.1: Software comprehension tasks that participants have to solve.

| Theme | Rationale | Id | Task |
|---|---|---|---|
| Find Outliers | Classes that exhibit extreme values of software metrics might indicate problems and might represent good candidates for refactoring | T1 | Find the three classes with the highest NOM |
| | | T2 | Find the three classes with the highest NOA |
| | | T3 | Find the three classes with the highest NLOC. If two are in the same range select the one with the lowest NLOC |
| Identify Patterns | The relationship between values of software metrics help developers to identify design problems. The ratios between the metric values produce patterns among the visual representation of entities | T4 | Locate the best candidate for the *god class* smell (hint: god classes contain many methods with many lines of code) |
| | | T5 | Locate the best candidate for the *data* class pattern (hint: a data class has high NOA, and low NOM and NLOC) |
| | | T6 | Locate the longest *facade* class (hint: facade classes have high NOM, and low NOA and NLOC) |
| Locate and Quantify | Help developers to prioritize what is most critical, *e.g.,* a package that contains many *god* classes might be a good candidate for refactoring | T7 | Locate the package with the highest number of classes such that NLOC in the classes are the least |
| | | T8 | Determine the total number of packages this system has |
| | | T9 | Estimate the total number of classes this system has |

Table 6.2: Systems used in the experiment. Participants visualized *Axion* for the training session. *Freemind* and *Azureus* were used for evaluation.

| System | Version | # KLOC | # Classes | # Pkgs. | Size |
|---|---|---|---|---|---|
| Axion | 1.0-M2 | 23 | 223 | 27 | Small |
| Freemind | 2.0.9 | 56 | 881 | 108 | Medium |
| Azureus | 4.8.1.2 | 646 | 6619 | 560 | Large |

conjecture that visualizations using this medium will excel in performance (RQ.1) and user experience (RQ.3), but it is not clear to us how this medium encourages user recollection (RQ.2). We want to know whether media may hinder the performance of visualizations, and if so, to what degree. We ask whether participants who use I3D or P3D might remember more details of the visualized software than participants who use a more conventional medium such as the computer display. We observe that P3D as opposed to I3D and SCS involves two senses: sight and touch. We conjecture that this characteristic promotes recollection. We also hypothesize that non-traditional media such as I3D and P3D might boost user experience. We consequently define the following null hypotheses:

*[H1]* When visualizing software as cities for comprehension, the *time to complete* tasks and the *accuracy* of developers is equal across various media (RQ.1).

*[H2]* When visualizing software as cities for a software comprehension task, the *recollection* of developers is equal across various media (RQ.2).

*[H3]* When visualizing software as cities for a software comprehension task, the user *experience* of developers is equal across various media (RQ.3).

### 6.3.3 Participants

One important goal for using between-groups design in our experiment (*i.e.,* each participant visualizes all systems using a single medium) is that groups have to be as similar as possible [Nie93]. We selected participants of the groups to have a similar distribution of gender and education level. Each group was formed of one post-doc researcher, five PhD students

and three bachelor/master students in computer science. The average age was 28.72 ± 1.43 years, and the average experience as a developer was 8.08 ± 0.77 years. Although participants of SCS reported to be used to the medium, participants of the other two media (*i.e.,* I3D and P3D) reported to be unfamiliar with the medium (we discuss this threat to the validity of our experiment in Section 6.6). Participants were not paid. They were invited and freely opted to participate in the study. Thirteen out of the twenty-seven participants were recruited from the University of Konstanz in Germany. The rest were recruited from the University of Bern in Switzerland. The interviews were conducted from February 2017 to March 2017.

### 6.3.4 Procedure

The experiment was conducted in two locations: one at the University of Konstanz and the other at the University of Bern. The rooms at both locations were of similar size (*i.e.,* 5 m x 5 m approximately) and lighting. During the study only the participant and the experimenter were in the room. The same experimenter conducted the experiment at both locations. A different setup was defined for each medium: for I3D, participants wore a headset and held a controller. Participants interacted with the visualization by walking and crouching. The tasks were displayed within the visualization. A legend with the encoding of the visualization was visible at all times. Participants used the controller to specify their answers to the tasks; SCS participants sat in a chair in front of the computer screen. They interacted with the visualization through the mouse and keyboard. The tasks were handed to them printed on paper. A legend with the encoding of the visualization was visible on a separate screen at all times. Finally, P3D participants sat in front of a desk on which the model was placed. They interacted with the model by holding, rotating and moving it with their hands. The tasks were also handed to them printed on paper. A legend with the encoding of the visualization was visible on a computer screen at all times. Participants had a wooden stick to point in the model to their answers.

We started the experiment by reading an introduction to explain participants the problem domain, the encoding of the visualization, and what they were expected to perform during the experiment. Firstly, participants had a training session where they viewed a visualization of the *Axion* system. They were asked to read one-by-one the tasks aloud, then they had to describe the visual pattern to solve the task, and finally they pointed

to the element that corresponded to their answer. Secondly, participants visualized *Freemind* and solved the tasks one at a time as they did during the training. This time, when they gave their answer to each of the tasks, we asked them how difficult they found the task. We asked them to score their answer on a 5-step Likert scale [Lik32]. When they finished all the tasks we asked them to approach a table where we previously placed 270 labels. Each label contained a word that represents a feeling. We placed positive feelings on the left side of the table and negative ones on the right. Labels were organized into eight groups of positive feelings and also eight of negative ones. Participants were asked to collect ten feelings, experienced during the previous visualization, from the table (without any restriction) and to sort them according to their intensity. Thirdly, participants visualized *Azureus* and repeated the same steps: solve the tasks, score their difficulty and identify the feelings experienced during the visualization. Lastly, to evaluate the recollection of near-time memories, participants were asked to approach a whiteboard and to draw what they remembered from the visualization of *Freemind* (approximately twenty minutes after they finished with the visualization).

### 6.3.5 Data Collection

We collected several data points during the experiment. We *(i)* video recorded participants as they navigated visualizations (*e.g.,* moving across the room in I3D) as well as the view they obtained of the visualization itself (*e.g.,* screen record in SCS), *(ii)* video recorded participants drawing the recollected memories of Freemind, and *(iii)* took pictures of the selected labels that described their experienced feelings during visualizations. We edited the videos to produce single records that contain the whole interview of each participant. We watched each of these records to measure and double-check completion time and accuracy, as well to identify recurrent concepts for qualitative analysis (observed emergent codes).

## 6.4 Results

We performed a statistical analysis of the collected data. To analyze performance, we observed that the results of accuracy did not follow a normal distribution. We then analyzed accuracy using Kruskal-Wallis' test [KW52]. We also observed that the rest of the dependent variables (*i.e., completion*

*time*, *recollection* and *experience*) satisfy *(i)* independent observations of between-groups design, *(ii)* homogeneous variances of dependent variables (validated using Lavene's test [$L^+60$]), and *(iii)* normal distribution of dependent variables (validated using Shapiro-Wilk's test [SW65]). Accordingly, we used the one-way Analysis Of Variance (ANOVA) to test these hypotheses, followed by Tukey's HSD for comparing differences between groups using a different medium. In either case, we chose a 95% confidence interval ($\alpha$ = .05) to evaluate whether there are statistically significant differences in *H1* performance (shown in Figures 6.3a and 6.3b), *H2* recollection (shown in Figure 6.4), and *H3* experience (shown in Figures 6.5a and 6.5b) between visualizations used to solve comprehension tasks among different media.

### 6.4.1 Performance (RQ.1)

Table 6.3 shows the results of the statistical tests that we carried out to analyze performance. We study performance by analyzing: *completion time* and *accuracy*.

#### Completion Time

Firstly, independent of the size of the system, there is a statistically significant variation of the time to *identify outliers* (T1-T3) among media, which was significantly greater than the variation of the time within each medium (Freemind: $f$=8.01, $p$=.00069 <0.05; Azureus: $f$=4.69, $p$=.012 <0.05). Thus, we *reject H1* for tasks T1-T3. Specifically, we found significant differences between P3D and I3D, and also between SCS and I3D but not between SCS and P3D. Secondly, in both software systems the variation of the time to *detect patterns* (T4-T6) among media was less than the variation of the time within each medium. Thus, we cannot *reject H1* for tasks T4-T6. Finally, in Freemind, there is a statistically significant variation of the time to *locate and quantify* classes (T7-T9) among media, which was significantly greater than the variation of the time within each medium ($f$=6.19, $p$=.0032 <0.05). Thus, we *reject H1* for tasks T7-T9. Specifically, we also found significant differences between SCS and I3D, and also between SCS and P3D but not between P3D and I3D. However, in Azureus, the variation of the time among media was less than the variation of the time within each medium. Thus, we cannot *reject H1* for tasks T7-T9. Figure 6.3a shows a box plots chart with the results of the

Table 6.3: Summary of the results of performance in terms of completion time and accuracy. The cases in which we found significant differences among the media are highlighted with a gray background.

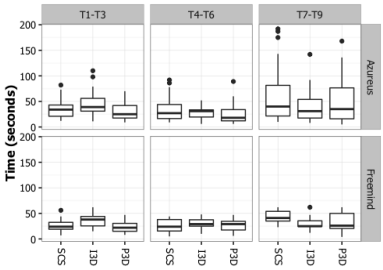
| Task | System | Performance | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Completion Time | | | | | | Accuracy | | |
| | | ANOVA | | | Tukey's HSD | | | Kruskal-Wallis | | |
| | | *p* | $F_{2,78}$ | mean | P3D-I3D | SCS-I3D | SCS-P3D | *p* | $\chi^2$ | median |
| **T1-T3** | **Freemind** | .00069 | 8.01 | 28.23 | .00089 | .0092 | .74 | .69 | .73 | 1.0 |
| | **Azureus** | .012 | 4.69 | 37.27 | .00069 | .096 | .65 | .055 | 5.8 | 1.0 |
| **T4-T6** | **Freemind** | .30 | 1.23 | 27.17 | – | – | – | .62 | .95 | 1.0 |
| | **Azureus** | .11 | 2.27 | 29.11 | – | – | – | .41 | 1.8 | 1.0 |
| **T7-T9** | **Freemind** | .0032 | 6.19 | 35.05 | .92 | .0053 | .019 | .01 | 9.2 | .86 |
| | **Azureus** | .20 | 1.65 | 50.54 | – | – | – | .02 | 7.8 | .53 |

time that participants required to complete the tasks during the experiment.

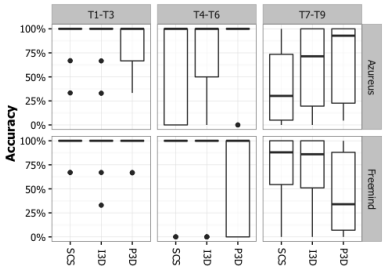
> Developers who visualize software cities for comprehension require the least time using **P3D** and **SCS** to *identify outliers*.

### Accuracy

The variation of the accuracy to *find outliers* (T1-T3), and *find patterns* (T4-T6), amongst media was less than the variation of the accuracy within each medium. Thus, we cannot *reject H1* for tasks T1-T6. Independent of the size of the system, there is a statistically significant variation of the accuracy to *locate and quantify* classes (T7-T9) among media, which was significantly greater than the variation of the accuracy within each medium (Freemind: $\chi^2$=9.2, $p$=.01 <.05; Azureus: $\chi^2$=7.8, $p$=.02 <.05). Thus, we *reject H1* for tasks T1-T3. Specifically, we found significant differences between SCS-P3D, and P3D-I3D, but not between SCS-I3D. Figure 6.3b shows a box plots chart with the results of the accuracy of participants during the experiment.




(a) Completion time of the participants in the experiment. Box plots are grouped by the theme of tasks (vertically). Rows contain the results that correspond to a different system.




(b) Accuracy of the participants in the experiment. Box plots are grouped by the theme of tasks (vertically). The results of each system are split into rows.

Figure 6.3: Performance

### 6.4.2 Recollection (RQ.2)

During software comprehension developers do not know what information might become relevant to remember. We therefore did not ask participants to remember details of the visualization. Instead, at the end of the interview we asked them to draw on a whiteboard what they remembered from the *Freemind* system (approximately twenty minutes after they finished with the visualization). Most participants said that they did not remember anything. However, after a few seconds they started to remember some details and drew some aspects of the visualizations on the board. We quantitatively analyzed the drawings by measuring two aspects of them *(i)* amount of used ink, and *(ii)* number of identified design problems.

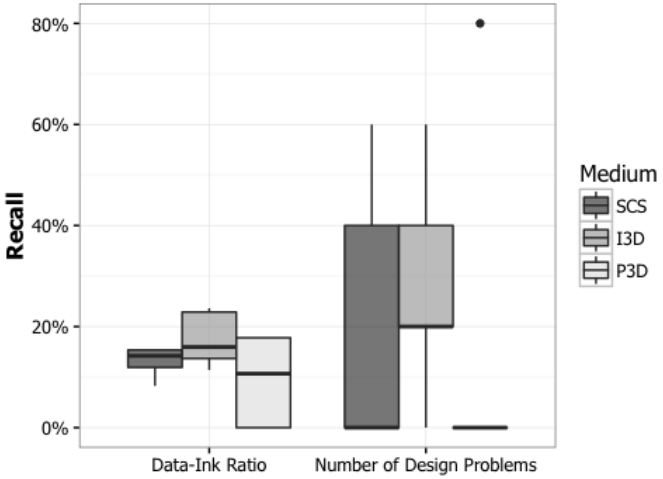



Figure 6.4: The mean recollection of the five most frequent candidates of design problems found in Freemind (skewers show the standard deviation). One means the five candidates recollected, while zero means none.

We were inspired by the data-ink ratio [Tuf01], which is a metric that measures the ratio between the amount of ink (number of non-white pixels) used to create a visualization and the amount of white space (number of white pixels) in the canvas. We argue that a high data-ink ratio of the drawings can be an indication of a good recollection.

Therefore, we analyzed the color statistics of pictures of the drawings using an online color summarizer[5]. We observed that the variation of the recollection among media was significantly greater than the variation of recollection within each medium ($F_{2,24} = 4.82$, $p = .017$). Thus, we *reject H2*. We found significant differences between P3D-I3D ($p = .014$) but not between SCS-P3D ($p = .47$) and SCS-I3D ($p = .16$). We also noticed that most drawings depicted the classes that are candidates of design problems (*e.g.,* god class, data class, longest facade) that participants had to find to solve the tasks. We measured their frequency and report the results in Figure 6.4. We observed that I3D has the highest recollection, followed by SCS and P3D, and that recollection decreases when visualizing larger systems (*i.e.,* Azureus). We did not find significant variances in the recollection of design problems ($p = .25$).

Developers who visualize software cities for comprehension obtain a significantly higher recollection when using **I3D** than when using **P3D**.

### 6.4.3 User Experience (RQ.3)

We measured two attributes that contribute to user experience: *difficulty* and experienced *feelings*. During the experiment *(i)* after each task we asked participants to rank the experienced difficulty using a 5-step Likert scale, and *(ii)* when participants finished all the tasks of one of the systems we asked them to identify their top ten strongest feelings experienced during the visualization.

#### Difficulty

Firstly, independent of the size of the system, the variation of the experienced difficulty in *finding outliers* (T1-T3) among media was significantly greater than the variation of the difficulty within each medium. Thus, we *reject H3* for tasks T1-T3. Specifically, in Freemind we found significant differences between SCS and I3D, and also between P3D and I3D but not between SCS and P3D; in Azureus we found significant differences only between SCS and I3D, but not between others. Secondly, in Freemind the variation of the experienced difficulty in *finding patterns* (T4-T6) among media was less than the variation of the difficulty within each medium.

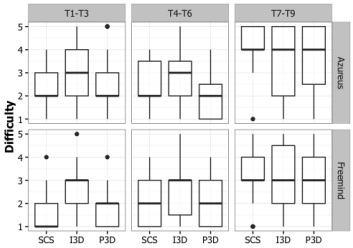
[5] `http://mkweb.bcgsc.ca/color-summarizer/`

Table 6.4: Summary of the results of user experience in terms of difficulty and feelings. The cases in which we found significant differences among the media are highlighted with a gray background.

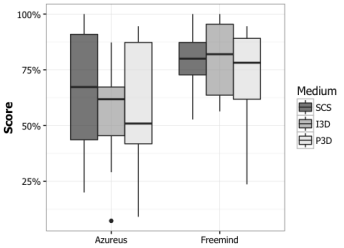
| System | Tasks | User Experience | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Difficulty | | | | | | Feelings | | |
| | | ANOVA | | | Tukey's HSD | | | ANOVA | | |
| | | ***p*** | $F_{2,78}$ | **mean** | **P3D-I3D** | **SCS-I3D** | **SCS-P3D** | ***p*** | $F_{2,78}$ | **mean** |
| **Freemind** | **T1-T3** | 9.69e-05 | 10.43 | 2.04 | .0011 | .00023 | .89 | .49 | .75 | .77 |
| | **T4-T6** | .20 | 1.6 | 2.0 | - | - | - | | | |
| | **T7-T9** | .21 | 1.61 | 2.21 | - | - | - | | | |
| **Azureus** | **T1-T3** | .022 | 3.99 | 2.38 | .42 | .018 | .29 | .57 | .58 | .59 |
| | **T4-T6** | .02 | .39 | 4.2 | .051 | .99 | .037 | | | |
| | **T7-T9** | .14 | 2.00 | 3.77 | - | - | - | | | |

Thus, we cannot *reject H3* for tasks T4-T6; in Azureus the variation of the experienced difficulty in *finding patterns* (T4-T6) among media was significantly greater than the variation of the difficulty within each medium. Thus, we *reject H3* for tasks T4-T6. Specifically, we found significant differences only between SCS and P3D, but not between others. Finally, independent of the size of the system, the variation of the experienced difficulty to *locate and quantify* classes (T7-T9) among media was less than the variation of the difficulty within each medium. Thus, we cannot *reject H3* for tasks T7-T9.

> Developers who visualize software cities for comprehension tasks perceive the least difficulty to *identify outliers* using **SCS**.




(a) Difficulty experienced by participants. Box plots are vertically grouped by the theme of tasks. The overall difficulty is higher in Azureus than in Freemind.




(b) Feelings' score experienced by participants. Bars show the mean results, and skewers show the standard deviation.

Figure 6.5: User Experience

### Feelings

Figure 6.6 shows a bar chart with the feelings experienced by the participants of the experiment. We included in the chart 74 frequent feelings that arise in various media, and excluded 40 feelings that were experienced only in a single medium.

We defined the *score* metric shown in Equation 6.1 to rank the experience of participants. The score is a weighted sum of the top ten strongest feelings that participants experienced during the visualization of the sys-

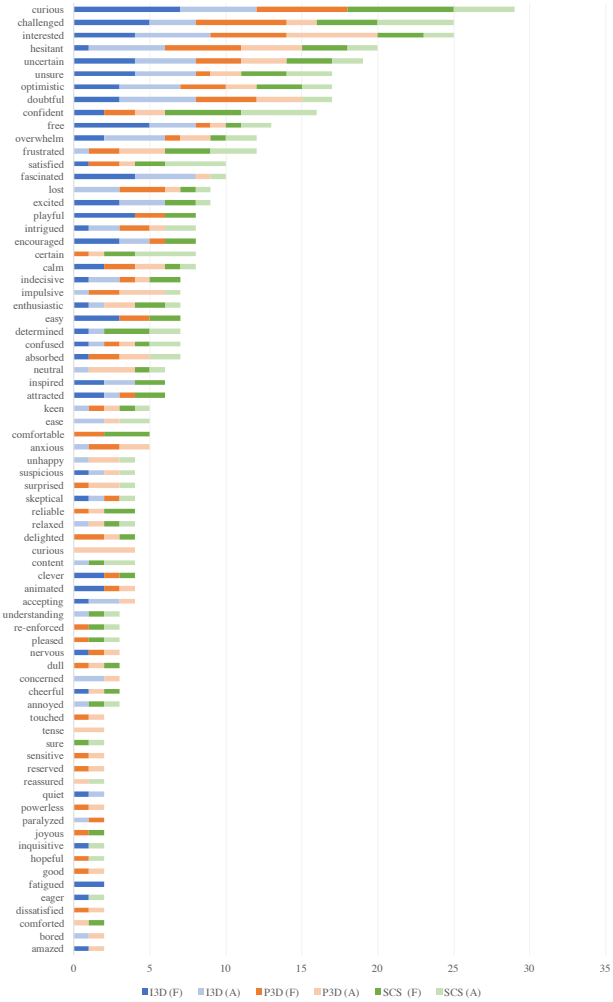



Figure 6.6: The frequent feelings experienced by participants who visualized Freemind (F), and Azureus (A) using various media.

tems. The score takes into account the intensity of the feeling: position one (*i.e.,* weakest feeling) to ten (*i.e.,* strongest feeling), and the type of feeling: positive (*i.e.,* 1) and negative (*i.e.,*-1). Independent of the size

of the system, the variation of the score of the experienced feelings of participants among media was less than the variation of the score within each medium (Freemind: $F_{2,78}$ = .75, $p$ = .49; Azureus: $F_{2,78}$ = .58, $p$ = .57). Thus, we cannot *reject H3*.

$$score = \sum_{i=1}^{10} i \times type(feeling_i)$$
$$type(feeling) = \begin{cases} 1 & \text{if feeling is positive} \\ -1 & \text{if feeling is negative} \end{cases} \tag{6.1}$$

We observed that the highest frequency of positive feelings is offered by SCS, in which users feel *confident*, *certain* and *satisfied* and a few times *frustrated*, *unsure*, and *overwhelmed*. Participants of I3D experienced balanced feelings. Sometimes they felt *interested*, *fascinated* and *optimistic*, and in some others cases they felt *doubtful*, *hesitant*, and *uncertain*. Participants of P3D reported the largest number of negative feelings of which the most frequent words were *hesitant*, *frustrated* and *impulsive*.

*Curious* and *challenge* are the two most frequent feelings identified among all media. After visualizing *Freemind* (*i.e.,* the medium size system) 67% of participants selected *curious* and 48% *challenge* (41% selected both simultaneously). Then, after participants visualized *Azureus* (*i.e.,* the largest system in the study) 41% of them selected *curious* and 37% *challenge* (19% selected both simultaneously).

## 6.5 Discussion

We now present a qualitative analysis of the results. We split the analysis by the concerns that we investigated through our research questions.

### 6.5.1 Performance (RQ.1)

We discuss the completion time and accuracy of participants based on the *theme* of tasks, the *size* of systems and the *medium* used. We also elaborate on the strategies and reflections made by participants.

#### Completion Time

We found that tasks that require little navigation (*i.e.,* finding outliers T1-T3) can be completed in the least time when visualizing systems using P3D

or SCS. However, we did not find significant differences of the variance of the time between them. For tasks that required more navigation (*i.e.,* to find design problems T4-T6) the results were mixed. The least time to solve the tasks in the medium sized system is obtained when visualizing on SCS. In contrast, SCS performed badly for the large system, for which the least time to solve the tasks is provided by P3D. One participant who was locating a package (task T7) observed that navigation in SCS makes it *"difficult to get to that part of the city"* On the contrary, a participant who used P3D to find the longest facade class (tasks T6) reflected that *"it is very easy to find these [types] of classes"* We observe that not all participants using SCS, who spent a longer time navigating a system, achieved a higher accuracy than participants using other media. A good balance is offered by I3D, for which one participant observes that *"depth helps a lot to identify packages"*.

**Accuracy**

We did not find significant differences in the variance of the accuracy among tasks that involve finding outliers and patterns (T1-T6). Answers across all the media used in the experiment were highly accurate. Instead, the results of the tasks related to location and quantification (T7-T9) show that participants are more accurate when quantifying elements in medium sized systems than in larger ones. Curiously, while in the medium sized system participants obtained the highest accuracy using SCS, they obtained the lowest one using SCS to visualized the large system. The opposite occurred with P3D. A more balanced result is obtained with I3D. We observed that the highest accuracy to assess the size of systems (location and quantification tasks T7-T9) was obtained by participants who compared a current visualized system to the one visualized during the training session (*i.e.,* Axion). Besides, participants who spent a longer time analyzing a system provided highly accurate estimations. One of them developed an algorithm that consisted of mentally dividing the city visualization into a number of sections with an approximately similar number of buildings and then multiplying the number of sections by the number of buildings. Interestingly, the result was the most accurate among participants using SCS and the top three across all media.

### 6.5.2 Recollection (RQ.2)

Participants were asked to draw on the whiteboard only what they freely remember from the second visualized system. We did not force them to guess an answer. In fact, a few of them did not draw anything. Among the majority that remembered some aspects of the visualized systems, their strongest memories were about the classes spotted when solving design problems tasks (*i.e.,* T4-T6). Most participants were unable to build an overview of the whole system, but had scattered memories of parts of it. Sometimes recollected memories were placed in a wrong location. Surprisingly, some of them remembered unexpected aspects of visualizations such as a thin line crossing the top of a building in P3D, a tiny crowded package in I3D. It suggests that recollection of memories might be boosted by encouraging users to identify particularities of the visualized systems. A few participants mentioned that they would *"expect a better recollection of memories if the tasks would encourage them to reason about the system as a whole"* Although the number of details and accuracy of the memories of participants varied, we can observe a trend. Visualization using I3D produced more detailed and significantly more accurate memories than visualization using P3D. Only three participants were unable to draw their recollection of the system who all used P3D.

### 6.5.3 User Experience (RQ.3)

Although we did not measure the interest of participants objectively, we assessed their interest by analyzing their questions and behavior. We noticed that even before the experiment participants who visualized systems using I3D were very interested. Participants who used P3D were less interested. Participants who visualized using SCS showed the least interest. A participant who ran the experiment using SCS asked to try the visualization in I3D just for fun. Participants perceived that the difficulty of tasks increased when they moved from the visualization of Freemind (medium size system) to Azureus (large system). Similarly, the same occurred with the number of negative feelings that also increased. We also analyzed distinct feelings that emerged in only one medium but not in the others. We think those feelings represent advantages and disadvantages that a medium impose. Feeling *quiet* is the most distinctive advantage of the I3D medium, and feeling *sure* (*i.e.,* certain) is so for SCS. The former might relate to the unique characteristics of being immersive in the visualization, while the latter

might reveal the certainty felt by users of traditional computer interfaces. Several distinctive feelings arise when using P3D that also might relate to the nature of the medium. Participants who used P3D positively felt *sensitive* and *touched*, and negatively felt *dissatisfied* and *powerless*. We noticed differences in the reported difficulty of tasks in terms of *(i)* **Size** of systems. Tasks were perceived to be less difficult in the medium size system (*i.e.,* Freemind) than in the large system (*i.e.,* Azureus); *(ii)* **Theme** of tasks. Tasks sorted by themes were perceived as increasing in difficulty. That is, tasks that concern *(a)* to identify outliers (T1-T3) were the least difficult, *(b)* to detect patterns (T4-T6) were of moderate difficulty, and *(c)* to locate and quantify (T7-T9) classes were the most difficult; and, *(iii)* **Medium**. Participants who used I3D consistently perceived tasks more difficult than participants who used other media. Between SCS and P3D participants had mixed perceptions depending on the type of task. Tasks concerned with identifying outliers (T1-T3) were perceived to be more difficult when using P3D, while tasks to detect patterns (T4-T6) were considered more difficult when using SCS.

We observed that even though participants who used I3D found most tasks consistently more difficult than participants using other media, they reported the most positive feelings, and their expression seemed happier during the visualization than participants who used other media. In summary, we consider that I3D provided the best overall experience to participants, closely followed by SCS and P3D.

Surprisingly, two of the three main concepts that influence engagement in computer games are the two most frequently selected by participants: *curious* and *challenge* [Mal80]. The third concept, which is *fantasy*, defined as "an illusory appearance",[6] is also inherent to visualizations. We observe that software visualizations could benefit from computer game techniques to increase the effectiveness of visualizations.

## 6.6 Threats to Validity

There are five main threats to the validity of our experiment. The first is *(i)* bias in the selection of groups. To mitigate this we formed similar groups in terms of education level, gender, age and experience in software development. The second threat is *(ii)* tasks might not be realistic. We

[6] "fantasy | phantasy, n." OED Online. Oxford University Press, March 2017. Web. 6 April 2017.

reduced this threat by defining types of tasks that have been previously used in other experiments and studies [AMN+10, WLR11, FKH15a]. The third threat is *(iii)* construct validity. The similarity and the quality of the implemented visualizations across the various media may have effected the performance of participants. To mitigate this, we ensured a high quality by testing the implemented visualizations with pilot users. We also transfered the visualizations to all media by automatic procedures. Consequently, the position, size of buildings was the same. Although color was automatically transfered to visualizations in I3D and SCS, we manually colored (*i.e.,* painted) visualizations in P3D. The fourth threat concerns to the *(iv)* method for measure recollection. We used the data-ink ratio from pictures taken to the drawings made by participants. These results might be affected by the size of the drawing, the use of the canvas to lay out recollected elements and the willingness of participants to spend time depicting a detailed drawing (the more time they spent, the more use of ink). The fifth thread is composed by *(v)* environmental aspects such as the room, light and experiment length might be different. Although we interviewed participants in two different locations, we chose rooms with similar characteristics (*i.e.,* size, light, level of noise), conducted the experiment following the same checklist, read the same introduction during the tutorial, displayed the same legend of the encoding used in the visualization in a second screen during the whole experiment, and offered to have a break, drinks and snacks to avoid fatigue to all participants. The same experimenter also conducted a pilot experiment with four participants to identify a suitable length for the experiment (approximately one hour), and fine-tune the tasks. Another thread that we observe is the that *(vi)* the novelty of the medium might have affected the perception of participants. Although we noticed the excitement of participants who were using a medium for the first time (*e.g.,* P3D), we observed that same excitement in participants who did the experiment using a medium familiar to them (*e.g.,* SCS). The final thread is *(vii)* any given participant did not have the opportunity to compare two or more media. We considered that the *learning effect* would hinder the quality of the results. Instead we opted for a between-groups design. That is, each participant visualized systems using a single medium.

## 6.7 Conclusion

Many visualizations have proven to be effective in supporting various software related tasks. Although multiple media can be used to display visualizations, most of software visualizations use a standard computer screen. We hypothesize that the medium used to present visualizations has a role in their effectiveness.

We investigated our hypotheses by conducting a controlled user experiment. In the experiment we focused on the 3D city visualization technique that has proven effective for software comprehension tasks. We deployed 3D city visualizations across a standard computer screen (SCS), immersive virtual reality (I3D), and a physical 3D printed model (P3D). For each medium we asked a different group of nine participants to perform a set of nine comprehension tasks and complete a questionnaire. We measured the effectiveness of visualizations in terms of performance (*i.e.,* completion time and correctness), recollection (*i.e.,* recollection of recent events), and user experience (*i.e.,* feelings and difficulties). We found that *(i)* even though developers using P3D required the least time to identify outliers, *(ii)* they perceived the least difficulty when visualizing systems based on SCS. Moreover, *(iii)* developers using I3D obtained the highest recollection.

# 7 Conclusion

When understanding large and complex software systems developers can benefit from software visualization tools. A visualization gives developers a tangible representation that they can analyze and use to discuss aspects of a software system. A software visualization can be used both *(1)* to discover unknown aspects of a system, and *(2)* to communicate the discovered insights to others. Yet, most software developers are not aware of visualization tools, and therefore software visualization is not within their toolbox. We observe two main issues: *(1)* due the lack of organization amongst proposed software visualizations developers struggle to find a suitable visualization tool that supports their particular problem, and *(2)* there is limited evidence of the effectiveness of a proposed software visualization.

In the following we elaborate on the contributions that resulted from our research, and describe future directions.

## 7.1 Contributions

Our contributions are the following:

*(1)* We studied the characteristics of the proposed software visualization. We identified opportunities for researchers in the field by exposing software engineering concerns with little attention by proposed visualization. Moreover, we found that few software visualizations have used a *medium* different than the standard computer screen (Chapter 2).

*(2)* We systematically reviewed the proposed software visualizations to analyze the evidence presented to validate the effectiveness of visualization approaches. We proposed guidelines for future evaluation of software visualization tools (Chapter 3).

*(3)* We elaborated on our efforts to fill the gap between proposed software visualizations and their practical application. We discussed a meta-visualization approach that connects live visualization examples to keywords collected from their supported questions. Also, we presented a curated catalog of 70 software visualization tools ready-to-use. Moreover, we showed preliminary results when encapsulating the characteristics of visualizations in an ontology (Chapter 4).

*(4)* We introduced *CityVR*, a prototype that implements a well-known software visualization technique (*i.e.,* the city metaphor) in immersive virtual reality. We conducted a formative experiment to identify the strengths of the medium (Chapter 5).

*(5)* We evaluated, in a controlled experiment, the effectiveness of our prototype compared to similar visualizations displayed on the standard computer screen, and a physical 3D model. We proposed extended benchmarks for future evaluation of software visualization tools (Chapter 6).

## 7.2 Future Work

We reflect that software visualization is a green field, in which there are several avenues to further investigate that split into three main categories:

*(i)* understanding and modeling software visualizations. In this avenue, we plan to promote the adoption of our software visualization ontology that encapsulates the characteristics of proposed tools, by developing a meta-visualization tool for supporting developers who seek visualizations that support particular development concerns.

*(ii)* facilitating the evaluation of software visualizations to increment the evidence of their effectiveness. We plan to create a software visualization corpus of annotated actionable software visualization tools ready for use in evaluations.

*(iii)* prototyping visualizations that stress various: tasks, media, and techniques. We believe that the effectiveness of some of the proposed software visualization tools can be increased by identifying more appropriate tasks, media, and techniques. In this avenue, we plan to start with the evaluation of a prototype that uses the city metaphor in immersive augmented reality to explore suitable tasks.

### 7.2.1 Software Visualization in Virtual Reality

Frameworks used to build visualizations in immersive virtual reality lack support for collaborative visualization. That is, to allow users to interact with each other in real time while visualizing a software system. We plan to develop such support. The prototype will allow remote users of the same application to see and interact with each other in the virtual world.[1]

### 7.2.2 Software Visualization in Augmented Reality

Most 3D software visualization is displayed using the computer screen medium. However, when displayed on the screen, 3D visualizations are constrained to a few screen inches and interaction limited to the capabilities of a mouse and a keyboard. Augmented reality technology allows a visualization to use the 3D physical space available, and allows users to interact using body motion such as hand gestures, gaze, and walking. We believe these characteristics promote the usability of visualization for software comprehension tasks. We plan to develop a prototype tool that implements the city metaphor to support comprehension of static aspects of software.

## 7.3 Summary

In this dissertation we analyze the state-of-the-art in software visualization. We introduce a taxonomy to classify software visualization approaches. We

[1] `http://scg.unibe.ch/wiki/projects/mastersbachelorsprojects/immersive-3D-collaborative-software-visualization`

discuss software engineering domains on which visualization approaches have focused, and compare them to the domains that get the most attention from practitioners. We expand our investigation by analyzing the evaluations used to obtain evidence of the effectiveness of proposed software visualizations. We discuss common pitfalls that we find amongst evaluations, and outline guidelines for researchers in the field who need to evaluate their software visualization approaches. We elaborate on our attempts to fill the gap between proposed software visualizations and their practical application. We think that the adoption of software visualization tools can be increased by *(1)* providing means to identify suitable visualizations for particular software engineering tasks, and *(2)* increasing the evidence of the effectiveness of proposed visualizations. For the latter we argue that the medium used to display software visualizations plays a role in their effectiveness to support software engineering tasks.

We explore our hypothesis via a formative experiment in which we implement the city metaphor visualization displayed with immersive virtual reality to support on software comprehension tasks. In the experiment, we observe that the selected medium excels at user experience, and we ask whether various media can increase the effectiveness of visualizations for particular tasks. To evaluate the impact of the medium in the effectiveness of visualizations for software comprehension, we conduct a controlled experiment in which developers visualize software cities displayed on three media (*i.e.,* standard computer screens, immersive virtual reality, physical 3D models). We observe that physical models boost completion time of tasks that involve identifying outliers, however, such tasks are considered the least difficult when the visualization is displayed on the standard computer screen. Moreover, we observe that immersive virtual reality excels at promoting recollection.

# Bibliography


[ABC+13] Vanessa Peña Araya, Alexandre Bergel, Damien Cassou, Stéphane Ducasse, and Jannik Laval. Agile visualization with Roassal. In *Deep Into Pharo*, pages 209–239. Square Bracket Associates, September 2013.

[AJdS+16] Hakam W Alomari, Rachel A Jennings, Paulo Virote de Souza, Matthew Stephan, and Gerald C Gannod. vizSlice: Visualizing large scale software slices. In *Proc. of VISSOFT*, pages 101–105. IEEE, 2016.

[AMN+10] Craig Anslow, Stuart Marshall, James Noble, Ewan Tempero, and Robert Biddle. User evaluation of polymetric views using a large visualization wall. In *Proc. of SOFTVIS*, pages 25–34, New York, NY, USA, 2010. ACM.

[AMNB13] C. Anslow, S. Marshall, J. Noble, and R. Biddle. SourceVis: Collaborative software visualization for co-located environments. In *Proc. of VISSOFT*, pages 1–10, September 2013.

[ANAV10] Vander Alves, Nan Niu, Carina Alves, and George Valença. Requirements engineering for software product lines: A systematic literature review. *Information and Software Technology*, 52(8):806–820, 2010.

[ANMT09] Craig Anslow, James Noble, Stuart Marshall, and Ewan Tempero. Towards visual software analytics. 2009.

[AS04] Robert Amar and John Stasko. A knowledge task-based framework for design and evaluation of information visualizations. In *Proc. of INFOVIS*, pages 143–150. IEEE, 2004.

[B+56] Benjamin S Bloom et al. Taxonomy of educational objectives. vol. 1: Cognitive domain. *New York: McKay*, pages 20–24, 1956.

[BBW09] Daniel Baldauf, Esther Burgard, and Marc Wittmann. Time perception as a workload measure in simulated car driving. *Journal of Applied Ergonomics*, 40(5):929–935, 2009.

[BC04] Emily Brown and Paul Cairns. A grounded investigation of game immersion. In *Proc. of CHI*, pages 1297–1300. ACM, 2004.

[BGBG16] Gergo Balogh, Tamás Gergely, Arpád Beszédes, and Tibor Gyimóthy. Using the city metaphor for visualizing test-related metrics. In *Proc. of SANER*, volume 2, pages 17–20. IEEE, 2016.

[BK01] S. Bassil and R.K. Keller. Software visualization tools: survey and analysis. In *Proc. of IWPC*, pages 7 –17, 2001.

[BM07] Doug A Bowman and Ryan P McMahan. Virtual reality: how much immersion is enough? *Computer*, 40(7), 2007.

[BTK11] Enrico Bertini, Andrada Tatu, and Daniel Keim. Quality metrics in high-dimensional data visualization: An overview and systematization. *Transactions on Visualization and Computer Graphics*, 17(12):2203–2212, 2011.

[CDPC11] Gerardo Canfora, Massimiliano Di Penta, and Luigi Cerulo. Achievements and challenges in software reverse engineering. *Commun. ACM*, 54(4):142–151, April 2011.

[CGNS15] Andrei Chiş, Tudor Gîrba, Oscar Nierstrasz, and Aliaksei Syrel. GTInspector: A moldable domain-aware object inspector. In *Proc. of SPLASH*, pages 15–16, New York, NY, USA, 2015. ACM.

[COR16] CORE, 2016.

[Dat02] Ameya Vivek Datey. Experiments in the use of immersion for information visualization, 2002.

[Die07] Stephan Diehl. *Software Visualization*. Springer-Verlag, Berlin Heidelberg, 2007.

[DPH10] Wim De Pauw and Steve Heisig. Zinsight: a visual and analytic environment for exploring large event traces. In *Proc. of SOFTVIS*, pages 143–152, New York, NY, USA, 2010. ACM.

[DPKM06] Wim De Pauw, Sophia Krasikov, and John Morar. Execution patterns for visualizing web services. In *Proc. of SOFTVIS*, New York NY, September 2006. ACM Press.

[DT13] Daniel J Dubois and Giordano Tamburrelli. Understanding gamification mechanisms for software development. In *Proc. of FSE*, pages 659–662. ACM, 2013.

[DZHC17] Stéphane Ducasse, Dmitri Zagidulin, Nicolai Hess, and Dimitris Chloupis. *Pharo by Example 5.0*. Square Bracket Associates, 2017.

[ERS+14] B. Ens, D. Rea, R. Shpaner, H. Hemmati, J. E. Young, and P. Irani. ChronoTwigger: A visual analytics tool for understanding source and test co-evolution. In *Proc. of VISSOFT*, pages 117–126, September 2014.

[EY15] Niklas Elmqvist and Ji Soo Yi. Patterns for visualization evaluation. *Proc. of INFOVIS*, 14(3):250–269, 2015.

[Fal02] Nils Faltin. Structure and constraints in interactive exploratory algorithm learning. In *Software Visualization*, pages 213–226. Springer, 2002.

[Fin03] Arlene Fink. *The survey handbook*, volume 1. Sage, 2003.

[FKH15a] Florian Fittkau, Erik Koppenhagen, and Wilhelm Hasselbring. Research perspective on supporting software engineering via physical 3D models. In *Proc. of VISSOFT*, pages 125–129. IEEE, 2015.

[FKH15b] Florian Fittkau, Alexander Krause, and Wilhelm Hasselbring. Exploring software cities in virtual reality. In *Proc. of VISSOFT*, pages 130–134. IEEE, 2015.

[FKH15c] Florian Fittkau, Alexander Krause, and Wilhelm Hasselbring. Hierarchical software landscape visualization for system comprehension: a controlled experiment. In *Proc. of VISSOFT*, pages 36–45. IEEE, 2015.

[FKH17] Florian Fittkau, Alexander Krause, and Wilhelm Hasselbring. Software landscape and application visualization for system comprehension with ExplorViz. *Information and Software Technology*, 87:259–277, 2017.

[FM10] Thomas Fritz and Gail C. Murphy. Using information fragments to answer the questions developers ask. In *Proc. of ICSE*, pages 175–184, New York, NY, USA, 2010. ACM.

[For10] Camilla Forsell. A guide to scientific evaluation in information visualization. In *Proc. of IV*, pages 162–169. IEEE, 2010.

[FR91] Thomas M. J. Fruchterman and Edward M. Reingold. Graph drawing by force-directed placement. *Softw. Pract. Exper.*, 21(11):1129–1164, November 1991.

[FSWH16] Michael D Feist, Eddie Antonio Santos, Ian Watts, and Abram Hindle. Visualizing project evolution through abstract syntax tree analysis. In *Proc. of VISSOFT*, pages 11–20. IEEE, 2016.

[GCC+10] Jeremy Gow, Paul Cairns, Simon Colton, Paul Miller, and Robin Baumgarten. Capturing player experience with postgame commentaries. In *Proc. of CGAT*, 2010.

[GDG06] Orla Greevy, Stéphane Ducasse, and Tudor Gîrba. Analyzing software evolution through feature views. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(6):425–456, 2006.

[GHM05] Keith Gallagher, Andrew Hatch, and Malcolm Munro. A framework for software architecture visualization assessment. In *Proc. of VISSOFT*, pages 76–81. IEEE CS, September 2005.

[Gru93] Tom Gruber. What is an ontology. *WWW Site http://www-ksl.stanford.edu/kst/whatis-an-ontology.html (accessed on 07-09-2004)*, 1993.

[GTS10] Lars Grammel, Melanie Tory, and Margaret-Anne Storey. How information visualization novices construct visualizations. *Transactions on Visualization and Computer Graphics*, 16(6):943–952, 2010.

[HCT+14] Samuel Huron, Sheelagh Carpendale, Alice Thudt, Anthony Tang, and Michael Mauerer. Constructive visualization. In *Proc. of DIS*, pages 433–442. ACM, 2014.

[HDSP09] Salima Hassaine, Karim Dhambri, Houari Sahraoui, and Pierre Poulin. Generating visualization-based analysis scenarios from maintenance task descriptions. In *Proc. of VISSOFT 2009*, pages 41–44. IEEE, 2009.

[HHPS07] Kasper Hornbæk, Rune Thaarup Høegh, Michael Bach Pedersen, and Jan Stage. Use case evaluation (UCE): A method for early usability evaluation in software development. In *Proc. of IFIP*, pages 578–591. Springer, 2007.

[HMA15] Nathan Hawes, Stuart Marshall, and Craig Anslow. CodeSurveyor: Mapping large-scale software to aid in code comprehension. In *Proc. of VISSOFT*, pages 96–105. IEEE, 2015.

[HRW00] Martin Höst, Björn Regnell, and Claes Wohlin. Using students as subjects — a comparative study of students and professionals in lead-time impact assessment. *Empirical Software Engineering*, 5:201–214, 2000.

[ID91] Alfred Inselberg and Bernard Dimsdale. Parallel coordinates. In *Human-Machine Interactive Systems*, pages 199–233. Springer, 1991.

[IIC+13] Tobias Isenberg, Petra Isenberg, Jian Chen, Michael Sedlmair, and Torsten Möller. A systematic review on the practice of evaluating visualization. *Transactions on Visualization and Computer Graphics*, 19(12):2818–2827, 2013.

[ITW01] Pourang Irani, Maureen Tingley, and Colin Ware. Using perceptual syntax to enhance semantic content in diagrams. *Computer Graphics and Applications*, 21(5):76–84, 2001.

[JCC+08] Charlene Jennett, Anna L Cox, Paul Cairns, Samira Dhoparee, Andrew Epps, Tim Tijs, and Alison Walton. Measuring and defining the experience of immersion in games. *International Journal of Human-Computer Studies*, 66(9):641–661, 2008.

[JS91] Brian Johnson and Ben Shneiderman. Tree-maps: a space-filling approach to the visualization of hierarchical information structures. In *Proc. of VIS*, pages 284–291, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[KDV07] Andrew J. Ko, Robert DeLine, and Gina Venolia. Information needs in collocated software development teams. In *Proc. of ICSE*, pages 344–353, Washington, DC, USA, 2007. IEEE Computer Society.

[Kee07] Staffs Keele. Guidelines for performing systematic literature reviews in software engineering. Technical report, Technical report, EBSE Technical Report EBSE-2007-01, 2007.

[Kei02] Daniel A Keim. Information visualization and visual data mining. *Transactions on Visualization and Computer Graphics*, 8(1):1–8, 2002.

[KJ18] Justin Kelly and Christian Jacob. evoExplore: Multiscale visualization of evolutionary histories in virtual reality. In *Proc. of EvoMUSART*, pages 112–127. Springer, 2018.

[KK96] Daniel A Keim and H-P Kriegel. Visualization techniques for mining large databases: A comparison. *Transactions on Knowledge and Data Engineering*, 8(6):923–938, 1996.

[KM99] Claire Knight and Malcolm Munro. Comprehension with [in] virtual environment visualisations. In *Proc. of ICPC*, pages 4–11. IEEE, 1999.

[KM00] Claire Knight and Malcolm Munro. Virtual but visible software. In *Proc. of IV*, pages 198–205. IEEE, 2000.

[KM07] Holger M. Kienle and Hausi A. Muller. Requirements of software visualization tools: A literature survey. *Proc. of VISSOFT*, pages 2–9, 2007.

[KM10] Holger M Kienle and Hausi A Müller. The tools perspective on software reverse engineering: requirements, construction, and evaluation. In *Advances in Computers*, volume 79, pages 189–290. Elsevier, 2010.

[KMM07] Holger M Kienle, Hausi A Muller, and Johannes Martin. Dependencies analysis of azureus with rigi: Tool demo challenge. In *Proc. of VISSOFT*, pages 159–160. IEEE, 2007.

[KW52] William H Kruskal and W Allen Wallis. Use of ranks in one-criterion variance analysis. *Journal of the American Statistical Association*, 47(260):583–621, 1952.

[L+60] Howard Levene et al. Robust tests for equality of variances. *Contributions to Probability and Statistics*, 1:278–292, 1960.

[LBI+12] Heidi Lam, Enrico Bertini, Petra Isenberg, Catherine Plaisant, and Sheelagh Carpendale. Empirical studies in information visualization: Seven scenarios. *Transactions on Visualization and Computer Graphics*, 18(9):1520–1536, 2012.

[LD03] Michele Lanza and Stéphane Ducasse. Polymetric views—a lightweight visual approach to reverse engineering. *Transactions on Software Engineering*, 29(9):782–795, September 2003.

[LHIE18] Roberto Erick Lopez-Herrejon, Sheny Illescas, and Alexander Egyed. A systematic mapping study of information visualization for software product line engineering. *Journal of Software: Evolution and Process*, 30(2), 2018.

[Lik32] Rensis Likert. A technique for the measurement of attitudes. *Archives of Psychology*, 22(140):1–55, 1932.

[LLG06] Mircea Lungu, Michele Lanza, and Tudor Gîrba. Package patterns for visual architecture recovery. In *Proc. of CSMR*, pages 185–196, Los Alamitos CA, 2006. IEEE Computer Society Press.

[LM10] Thomas D. LaToza and Brad A. Myers. Hard-to-answer questions about code. In *Proc of. PLATEAU*, pages 8:1–8:6, New York, NY, USA, 2010. ACM.

[LMSW03] Rob Lintern, Jeff Michaud, Margaret-Anne Storey, and Xiaomin Wu. Plugging-in visualization: experiences integrating a visualization tool with eclipse. In *Proc. of SOFTVIS*, pages 47–ff. ACM, 2003.

[Mac86] Jock Mackinlay. Automating the design of graphical presentations of relational information. *Transactions On Graphics*, 5(2):110–141, 1986.

[Mal80] Thomas W Malone. What makes things fun to learn? heuristics for designing instructional computer games. In *Proc. of SIGSMALL*, pages 162–169. ACM, 1980.

[MBK+04] John Maloney, Leo Burd, Yasmin Kafai, Natalie Rusk, Brian Silverman, and Mitchel Resnick. Scratch: A sneak preview. In *Proc. of International Conference on Creating, Connecting and Collaborating through Computing*, pages 104–109. IEEE Computer Society, 2004.

[MBN18] Leonel Merino, Alexandre Bergel, and Oscar Nierstrasz. Overcoming issues of 3D software visualization through immersive augmented reality. In *Proc. of VISSOFT*, page in review. IEEE, 2018.

[MFB+17] Leonel Merino, Johannes Fuchs, Michael Blumenschein, Craig Anslow, Mohammad Ghafari, Oscar Nierstrasz, Michael Behrisch, and Daniel Keim. On the impact of the medium in the effectiveness of 3D software visualization. In *Proc. of VISSOFT*, pages 11–21. IEEE, 2017.

[MGAN] Leonel Merino, Mohammad Ghafari, Craig Anslow, and Oscar Nierstrasz. A systematic literature review of software visualization evaluation. *The Journal of Systems and Software*, page in review.

[MGAN17] Leonel Merino, Mohammad Ghafari, Craig Anslow, and Oscar Nierstrasz. CityVR: Gameful software visualization. In *Proc. of ICSME*, pages 633–637. IEEE, 2017.

[MGN16a] Leonel Merino, Mohammad Ghafari, and Oscar Nierstrasz. Towards actionable visualisation in software development. In *Proc. of VISSOFT*. IEEE, 2016.

[MGN+16b] Leonel Merino, Mohammad Ghafari, Oscar Nierstrasz, Alexandre Bergel, and Juraj Kubelka. MetaVis: Exploring actionable visualization. In *Proc. of VISSOFT*. IEEE, 2016.

[MGN17] Leonel Merino, Mohammad Ghafari, and Oscar Nierstrasz. Towards actionable visualization for software developers. *Journal of Software: Evolution and Process*, 30(2):e1923–n/a, 2017.

[MHB10] Emerson Murphy-Hill and Andrew P Black. An interactive ambient visualization for code smells. In *Proc. of SOFTVIS*, pages 5–14. ACM, 2010.

[MIK+16] Anna-Liisa Mattila, Petri Ihantola, Terhi Kilamo, Antti Luoto, Mikko Nurminen, and Heli Väätäjä. Software visualization today: systematic literature review. In *Proc. of International Academic Mindtrek Conference*, pages 262–271. ACM, 2016.

[MJSK15] Sebastian Mittelstädt, Dominik Jäckle, Florian Stoffel, and Daniel A Keim. ColorCAT: Guided design of colormaps for combined analysis tasks. In *Proc. of Eurographics*, volume 2, 2015.

[MKS+14] Richard Müller, Pascal Kovacs, Jan Schilbach, Ulrich W Eisenecker, Dirk Zeckzer, and Gerik Scheuermann. A structured approach for conducting a series of controlled experiments in software visualization. In *Proc. of IVAPP*, pages 204–209. IEEE, 2014.

[MLMD01] Jonathan I Maletic, Jason Leigh, Andrian Marcus, and Greg Dunlap. Visualizing object-oriented software in virtual reality. In *Proc. of IWPC*, pages 26–35. IEEE, 2001.

[MLN15] Leonel Merino, Mircea Lungu, and Oscar Nierstrasz. Explora: A visualisation tool for metric analysis of software corpora. In *Proc. of VISSOFT*, pages 195–199. IEEE, 2015.

[MM03] Jonathan I Maletic and Andrian Marcus. CFB: A call for benchmarks-for software visualization. In *Proc. of VISSOFT*, pages 113–116. Citeseer, 2003.

[MMC02] Jonathan I. Maletic, Andrian Marcus, and Michael Collard. A task oriented view of software visualization. In *Proc. of VISSOFT*, pages 32–40. IEEE, June 2002.

[MSGN16] Leonel Merino, Dominik Seliner, Mohammad Ghafari, and Oscar Nierstrasz. CommunityExplorer: A framework for visualizing collaboration networks. In *Proc. of IWST*, pages 2:1–2:9, 2016.

[Mun08] Tamara Munzner. Process and pitfalls in writing information visualization research papers. In *Information visualization*, pages 134–153. Springer, 2008.

[Mun14] Tamara Munzner. *Visualization analysis and design*. CRC press, 2014.

[Mus15] Mark A Musen. The Protégé project: a look back and a look forward. *AI matters*, 1(4):4–12, 2015.

[NDG05] Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gîrba. The story of Moose: an agile reengineering environment. In *Proc. of ESEC/FSE*, pages 1–10, New York, NY, USA, September 2005. ACM Press. Invited paper.

[Nie93] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann, 1st edition, September 1993.

[NM+01] Natalya F Noy, Deborah L McGuinness, et al. Ontology development 101: A guide to creating your first ontology, 2001.

[NTM+13] Renato Lima Novais, André Torres, Thiago Souto Mendes, Manoel Mendonça, and Nico Zazworka. Software evolution visualization: A systematic mapping study. *Information and Software Technology*, 55(11):1860–1883, 2013.

[PAM14] Julia Paredes, Craig Anslow, and Frank Maurer. Information visualization for agile software development. In *Proc. of VISSOFT*, pages 157–166. IEEE, 2014.

[PBS93] Blaine A. Price, Ronald M. Baecker, and Ian S. Small. A principled taxonomy of software visualization. *Journal of Visual Languages and Computing*, 4(3):211–266, 1993.

[PEQ+07] Thomas Panas, Thomas Epperly, Daniel Quinlan, Andreas Saebjornsen, and Richard Vuduc. Communicating software architecture using a unified single-view visualization. In *Proc. of ICECCS*, pages 217–228. IEEE, 2007.

[PGBP15] Oscar Pedreira, Félix García, Nieves Brisaboa, and Mario Piattini. Gamification in software engineering–a systematic mapping. *Journal of the American Society for Information Science and Technology*, 57:157–168, 2015.

[PJ09] Yunrim Park and Carlos Jensen. Beyond pretty pictures: Examining the benefits of code visualization for open source newcomers. In *Proc. of VISSOFT*, pages 3–10. IEEE, 2009.

[PLL05] Thomas Panas, Rüdiger Lincke, and Welf Löwe. Online-configuration of software visualization with Vizz3D. In *Proc. of SOFTVIS*, pages 173–182, 2005.

[PSM07] Harkirat Padda, Ahmed Seffah, and Sudhir Mudur. Visualization patterns: A context-sensitive tool to evaluate visualization techniques. In *Proc. of VISSOFT*, pages 88–91. IEEE, 2007.

[Raa12] Felix Raab. CodeSmellExplorer: Tangible exploration of code smells and refactorings. In *Proc. of VL/HCC*, pages 261–262. IEEE, 2012.

[RBLN04] Dheva Raja, Doug Bowman, John Lucas, and Chris North. Exploring the benefits of immersion in abstract information visualization. In *Proc. Immersive Projection Technology Workshop*, pages 61–69, 2004.

[RC93] G-C Roman and Kenneth C Cox. A taxonomy of program visualization systems. *Computer*, 26(12):11–24, 1993.

[Rei03] Steven P. Reiss. Visualizing Java in action. In *Proc. of SOFTVIS*, pages 57–66, 2003.

[Rei05] Steven P. Reiss. JOVE: Java as it happens. In *Proc. of SOFTVIS*, pages 115–124, 2005.

[Rei14] Steven P Reiss. The challenge of helping the programmer during debugging. In *Proc. of VISSOFT*, pages 112–116. IEEE, 2014.

[RH09] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131, 2009.

[RW+11] Nornadiah Mohd Razali, Yap Bee Wah, et al. Power comparisons of shapiro-wilk, kolmogorov-smirnov, lilliefors and anderson-darling tests. *Journal of Statistical Modeling and Analytics*, 2(1):21–33, 2011.

[SBCS14] A. Seriai, O. Benomar, B. Cerat, and H. Sahraoui. Validation of software visualization tools: A systematic mapping study. In *Proc. of VISSOFT*, pages 60–69, September 2014.

[SBFB16] Rodrigo Schulz, Fabian Beck, Jhonny Wilder Cerezo Felipez, and Alexandre Bergel. Visually exploring object mutation. 2016.

[SCG+15] Aliaksei Syrel, Andrei Chiş, Tudor Gîrba, Juraj Kubelka, Oscar Nierstrasz, and Stefan Reichhart. Spotter: towards a unified search interface in IDEs. In *Proc. of SPLASH*, pages 54–55, New York, NY, USA, 2015. ACM.

[SCGM00] John T. Stasko, Richard Catrambone, Mark Guzdial, and Kevin Mcdonald. An evaluation of space-filling information visualizations for depicting hierarchical structures. *International Journal Humain-Computer Studies*, 53(5):663–694, 2000.

[SDP+09] Beatriz Sousa Santos, Paulo Dias, Angela Pimentel, Jan-Willem Baggerman, Carlos Ferreira, Samuel Silva, and Joaquim Madeira. Head-mounted display versus desktop for 3D navigation in virtual reality: a user study. *Multimedia Tools and Applications*, 41(1):161, 2009.

[SHH+05] Dag IK Sjøberg, Jo Erskine Hannay, Ove Hansen, Vigdis By Kampenes, Amela Karahasanovic, N-K Liborg, and Anette C Rekdal. A survey of controlled experiments in software engineering. *Transactions on Software Engineering*, 31(9):733–753, 2005.

[Shn96] Ben Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *IEEE Visual Languages*, pages 336–343, College Park, Maryland 20742, U.S.A., 1996.

[SLB14] Mojtaba Shahin, Peng Liang, and Muhammad Ali Babar. A systematic review of software architecture visualization techniques. *Journal of Systems and Software*, 94:161–185, 2014.

[SMDV06] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. Questions programmers ask during software evolution tasks. In *Proc. of FSE*, pages 23–34, New York, NY, USA, 2006. ACM.

[SOF16] SoftVis, 2016.

[SOT08a] Mariam Sensalire, Patrick Ogao, and Alexandru Telea. Classifying desirable features of software visualization tools for corrective maintenance. In *Proc. of SOFTVIS*, pages 87–90. ACM, 2008.

[SOT08b] Mariam Sensalire, Patrick Ogao, and Alexandru Telea. Classifying desirable features of software visualization tools for corrective maintenance. In *Proc. of SOFTVIS*, pages 87–90. ACM, 2008.

[SOT09] Mariam Sensalire, Patrick Ogao, and Alexandru Telea. Evaluation of software visualization tools: Lessons learned. In *Proc. of VISSOFT*, pages 19–26. IEEE, 2009.

[SRGBSA12] M.A. Sicilia, D. Rodríguez, E. García-Barriocanal, and S. Sánchez-Alonso. Empirical findings on ontology metrics. *Expert Systems with Applications*, 39(8):6706 – 6711, 2012.

[SS11] Ahmed Sfayhi and Houari Sahraoui. What you see is what you asked for: An effort-based transformation of code analysis tasks into interactive visualization scenarios. In *Proc. of SCAM*, pages 195–203. IEEE, 2011.

[SvG05] Margaret-Anne D. Storey, Davor Čubranić, and Daniel M. German. On the use of visualization to support awareness of human activities in software development: a survey and a framework. In *Proc. of SOFTVIS*, pages 193–202. ACM Press, 2005.

[SVW14] Marcelo Schots, Renan Vasconcelos, and Cláudia Werner. A quasi-systematic review on software visualization approaches for software reuse. *Technical report*, 2014.

[SW65] S. S. Shapiro and M. B. Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, 52(3-4):591, 1965.

[SW14] Marcelo Schots and Claudia Werner. Using a task-oriented framework to characterize visualization approaches. In *Proc. of VISSOFT*, pages 70–74. IEEE, 2014.

[TAD+10] E. Tempero, C. Anslow, J. Dietrich, T. Han, Jing Li, M. Lumpe, H. Melton, and J. Noble. The Qualitas Corpus: A curated collection of Java code for empirical studies. In *Proc. of APSEC*, pages 336 –345, December 2010.

[TC09] Alfredo R Teyseyre and Marcelo R Campo. An overview of 3D software visualization. *Transactions on Visualization and Computer Graphics*, 15(1):87–105, 2009.

[TGG08] Roberto Theron, Antonio Gonzalez, and Francisco J Garcia. Supporting the understanding of the evolution of software items. In *Proc. of SOFTVIS*, pages 189–192. ACM, 2008.

[TKII17] Boris Todorov, Raula Gaikovina Kula, Takashi Ishio, and Katsuro Inoue. SoL Mantra: Visualizing update opportunities based on library coexistence. In *Proc. of VISSOFT*, pages 129–133. IEEE, 2017.

[TLTC05] Maurice Termeer, Christian F.J. Lange, Alexandru Telea, and Michel R.V. Chaudron. Visual exploration of combined architectural and metric information. 0:11, 2005.

[Tuf01] Edward R. Tufte. *The Visual Display of Quantitative Information*. Graphics Press, 2nd edition, 2001.

[VIS16] VISSOFT, 2016.

[VTvW05] Lucian Voinea, Alex Telea, and Jarke J. van Wijk. CVSscan: visualization of code evolution. In *Proc. of SOFTVIS*, pages 47–56, St. Louis, Missouri, USA, May 2005.

[VW06] Jarke J Van Wijk. Views on visualization. *Transactions on Visualization and Computer Graphics*, 12(4):421–432, 2006.

[Wet10] Richard Wettel. *Software Systems as Cities*. PhD thesis, University of Lugano, Switzerland, September 2010.

[WFGH11] Jacob O Wobbrock, Leah Findlater, Darren Gergle, and James J Higgins. The aligned rank transform for nonparametric factorial analyses using only anova procedures. In *Proc. of SIGCHI*, pages 143–146. ACM, 2011.

[WFRFN] Jorge A Wagner Filho, Marina F Rey, Carla MDS Freitas, and Luciana Nedel. Immersive visualization of abstract information: An evaluation on dimensionally-reduced data scatterplots.

[WL07a] Richard Wettel and Michele Lanza. Program comprehension through software habitability. In *Proc. of ICPC*, pages 231–240. IEEE CS Press, 2007.

[WL07b] Richard Wettel and Michele Lanza. Visualizing software systems as cities. In *Proc. of VISSOFT*, pages 92–99, 2007.

[WLR11] Richard Wettel, Michele Lanza, and Romain Robbes. Software systems as cities: a controlled experiment. In *Proc. of ICSE*, pages 551–560, New York, NY, USA, 2011. ACM.

[WRH+00] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering*. Kluwer Academic Publishers, 2000.

[WRH+12] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[Yin13] Robert K Yin. *Case study research: Design and methods*. Sage publications, 2013.

[YM98] Peter Young and Malcolm Munro. Visualising software in virtual reality. In *Proc. of IWPC*, pages 19–26. IEEE, 1998.

[YP15] Alfa Yohannis and Yulius Prabowo. Sort attack: Visualization and gamification of sorting algorithm learning. In *Proc. of VS-Games*, pages 1–8. IEEE, 2015.

[ZW98] Marvin V. Zelkowitz and Dolores R. Wallace. Experimental models for validating technology. *Computer*, 31(5):23–31, 1998.

# Erklärung

gemäss Art. 28 Abs. 2 RSL 05

| | |
|---|---|
| Name/Vorname: | Merino Leonel |
| Matrikelnummer: | 11-117-041 |
| Studiengang: | Informatik |

Bachelor ◯ Master ◯ Dissertation ◉

Titel der Arbeit: The Medium of Visualization for Software Comprehension

LeiterIn der Arbeit: Prof. Dr. Oscar Nierstrasz

Ich erkläre hiermit, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäss aus Quellen entnommen wurden, habe ich als solche gekennzeichnet. Mir ist bekannt, dass andernfalls der Senat gemäss Artikel 36 Absatz 1 Buchstabe r des Gesetzes vom 5. September 1996 über die Universität zum Entzug des auf Grund dieser Arbeit verliehenen Titels berechtigt ist. Ich gewähre hiermit Einsicht in diese Arbeit.

Bern, 05.06.2018

Ort/Datum

Unterschrift

# Curriculum Vitae

## Personal Information

| | |
|---|---|
| Name: | Leonel Alejandro Merino del Campo |
| Date of Birth: | 12.06.1979. |
| Place of Birth: | San Bernardo, Chile |
| Nationality: | Chilean |

## Education

| | |
|---|---|
| 2014–2018 | **PhD in Computer Science**<br>University of Bern<br>Switzerland |
| 2007–2008 | **MSc in Computer Science**<br>École des mines de Nantes<br>France |
| 2007–2008 | **MSc in Computer Science**<br>Vrije Universiteit Brussel<br>Belgium |
| 1999–2006 | **BSc in Computer Science**<br>University of Chile<br>Chile |