Bird-Eye Views of Object-Oriented Software

Nour Jihene Agouf

Université de Lille, Faculté des Sciences et Technologies

École doctorale MADIS Lille

Bird-Eye Views of Object-Oriented Software

Vues à Vol d'Oiseau sur les Systèmes Orientés Objet

THÈSE

présentée et soutenue publiquement le 12 décembre 2023

pour l'obtention du

Doctorat de l'Université de Lille

(spécialité informatique)

par

Nour Jihene AGOUF

Composition du jury

Président :	Jean-Michel Bruel (Professeur - Université de Toulouse)
Rapporteurs :	Christelle Urtado (Professeur - IMT Mines Alès) Jean-Remy Falleri (Professeur - Université de Bordeaux)
Directrice de thèse :	Anne Etien (Professeur, Université de Lille)
Co-Directeur de thèse :	Stéphane Ducasse (Directeur de Recherche, INRIA Lille)
Invité :	Arnaud Thiefaine (Coach Craft – Arolla)

Centre de Recherche en Informatique, Signal, et Automatique de Lille — UMR 9189 CRIStAL
INRIA Lille - Nord Europe

Acknowledgments

First and foremost, I extend my heartfelt gratitude to my esteemed supervisors, Stéphane Ducasse, for his immense support from the inception of my Ph.D. journey, guidance throughout the entire research process, and securing funding for my thesis. I will forever be grateful. To Anne Etien, whose continuous mentorship, understanding, and empowering feminine energy that profoundly shaped my development and enriched my experience. I am profoundly grateful for the immense knowledge and expertise I have gained under their tutelage.

I would like to express my sincere appreciation to Arolla software company for their generous sponsorship of my Ph.D. and to the remarkable individuals at Arolla, notably Olivier Moglia, Cyrille Martraire, and Arnaud Thiefaine, for their support and collaboration. Additionally, I am grateful to the entire RMoD team at Inria for providing a nurturing and stimulating work environment, fostering my growth, and fostering a good sense of community.

I am indebted to my beloved mother, whose support throughout my academic pursuits has been a source of strength and inspiration. Her invaluable lessons on the significance of hard work and perseverance have been instrumental in shaping my character. I would also like to express my gratitude to my dear sister and the exceptional group of supportive friends, both within France and internationally, whose encouragement has made this journey all the more meaningful. To my friends within the RMoD team, with whom I have shared countless meals, engaging discussions, and joyous laughter, I am truly grateful for the camaraderie and support they have provided.

Lastly, I would like to humbly acknowledge myself for maintaining self-belief, patience, and a positive mindset throughout this arduous journey. These qualities have been instrumental in overcoming challenges and persevering towards the successful completion of my Ph.D.

Abstract

Software maintenance is a challenging task. It requires reading and understanding the software besides source code investigations and analysis. Maintainers usually rely on tools such as IDEs, tests and debuggers to navigate through the source code, understand its logic, detect different anomalies and correct them , etc. However, such techniques can be time-consuming for maintainers and companies. In fact, according to the existing literature, more than half of the time dedicated to software maintenance is spent reading and understanding the source code before making any changes or decisions on the software.

This thesis takes a distinct approach to software maintenance by offering novel visualizations that answer to maintainers' needs. Each visualization is dedicated to a specific task inspired by the needs of the software maintainers of our industrial partners. The research done in this thesis focuses on one main objective which is detecting software violations. Our definition of violations, however, concerns three main challenges: *detecting architectural* violations, *naming conventions* violations and (anti-) naming patterns and finally, violations of *clean code* principles inside classes. To address these challenges, targeted visualizations are proposed: *Clis-ervo* for the detection of architectural violations in client-server software, *Class-Name Distribution* (CnD) for the detection of naming violations, and innovation of the patrimonial *ClassBlueprint* Visualization (CBv2) for detecting bad quality classes' source code. These visualizations assist maintainers in understanding and improving software systems, ultimately leading to more efficient and sustainable maintenance processes.

Each of the visualizations is validated independently with software maintainers and a diversity of projects including the ones of our industrial partners.

Keywords: Program visualizations, program comprehension, software violations

Résumé

La maintenance logicielle est une tâche complexe. Elle requiert la lecture et la compréhension du programme ainsi que des investigations et des analyses du code source. Les mainteneurs se reposent généralement sur des outils tels que les environnements de développement intégrés (IDEs), les tests et les débogueurs pour naviguer à travers le code source, comprendre sa logique, détecter différentes anomalies et les corriger, etc. Cependant, ces techniques sont chronophages pour les mainteneurs et les entreprises. En effet, selon la littérature existante, plus de la moitié du temps consacré à la maintenance logicielle est attribué à la lecture et à la compréhension du code source avant de prendre des décisions ou d'apporter des modifications au logiciel.

Cette thèse aborde la tâche de maintenance logicielle de manière nouvelle en proposant des visualisations novatrices qui répondent aux besoins des mainteneurs. Chaque visualisation est dédiée à une tâche spécifique inspirée des besoins des mainteneurs de logiciels de nos partenaires industriels. La recherche menée dans cette thèse se concentre sur un objectif principal qui est la détection de violations logicielles. Notre définition des violations, cependant, concerne trois principaux défis : la détection de violations *architecturales*, de violations de *conventions de nommage* et de (anti-)patterns de nommage, ainsi que de violations des principes du *code propre* à l'intérieur des classes. Pour relever ces défis, des visualisations ciblées sont proposées : *Cliservo* pour la détection des violations architecturales des programmes client-serveur. La *ClassName Distribution* (CnD) pour la détection de violations de nommage, et une amélioration de la *ClassBlueprint* originale (CBv2) pour la détection du code des classes de mauvaise qualité. Ces visualisations aident les mainteneurs à comprendre et à améliorer les systèmes logiciels, conduisant finalement à des processus de maintenance plus efficaces et durables.

Chacune des visualisations est validée indépendamment auprès des mainteneurs de logiciels et sur une diversité de projets, y compris ceux de nos partenaires industriels.

Mots clé: Visualizations des programmes, comprehension des programmes, violations logiciel

Contents

List of Figures

List of Tables

Listings

CHAPTER 1

Introduction

Contents

1.1 Maintenance and Program Comprehension

Any software is susceptible to become legacy software. Parnas *et al.,* [Parnas, 1994] state that software age over time, meaning that their structure and quality degrade and complexity increases unless work is done to maintain and reduce it [Lehman, 1996]. By recognizing the propensity of software to become legacy software, researchers and practitioners advocate for proactive maintenance practices, emphasizing the importance of reducing software complexity, optimizing software structures, and improving software quality. Such measures contribute to the longevity, stability, and maintainability of software systems, ensuring they can effectively adapt to evolving requirements, technology advancements, and changing user needs throughout their lifespan.

One of the primary challenges software maintainers face is the comprehension of abstractions and/or intricate code assets within the software before implementing any further modifications [Pigoski, 1997, von Mayrhauser and Vans, 1995]. Indeed, program comprehension plays a vital role in software maintenance, regardless of the specific maintenance activity involved. It serves as a foundational process that enables maintainers to understand and navigate the software codebase effectively. Program comprehension is essential for various maintenance tasks. This understanding is crucial for maintainers to identify the root causes of issues, implement

necessary changes, and ensure the overall stability and reliability of the software system. According to the existing literature, more than half of the time dedicated to software maintenance is spent on reading and understanding the source code itself [Pigoski, 1997, Shari et al., 1998]. Software maintainers rely on tools such as integrated development environments (IDEs) like Eclipse and IntelliJ, as well as debuggers, to aid in source code comprehension [Maalej et al., 2014]. However, this approach contributes to the time and energy-consuming nature of the maintenance process.

In this thesis, we diverge from the conventional systematic and opportunistic approach [Littman et al., 1987] that relies on tools such as IDEs, debuggers, and executable tests for comprehending source code. Instead, we address the needs of maintainers to comprehend aspects of the source code using *Software Visualizations* [Diehl, 2002, 2007a,b, Stasko et al., 1998, Ware, 2000, 2004]. The study of software visualization is concerned with the representation of software systems by employing visual abstractions of various artifacts related to software and its development process [Diehl, 2007b]. These visual abstractions encompass aspects such as the distribution of properties [Ducasse et al., 2006], evaluations of code quality [Ducasse and Lanza, 2005, Tamer et al., 2021], and depictions of software architecture [Boccuzzo and Gall, 2007, Kobayashi et al., 2013, Wettel and Lanza, 2008]. The primary objective of software visualization is to provide a visual framework that facilitates a better understanding of software systems and aids in the analysis, exploration, and communication of their underlying structures and characteristics. The main motivation for using software visualization in this work is to help software maintainers in comprehending different aspects of software systems during the software development process to facilitate software maintenance tasks, hence minimizing the cost of maintenance and its evolution [Diehl, 2007b, Gallagher et al., 2008].

1.2 Challenges in Early Stages of Maintainance

Understanding thousands of lines of source code takes an enormous amount of time and effort [Maalej et al., 2014, Sommerville, 2000, Storey et al., 1999, Votipka et al., 2020], and the understanding process purely depends on the experience, skills, and even the mood of the maintainer [Sneed, 1996]. Roehm *et al.,* [Roehm et al., 2012] report that experienced developers understand abstractions of the code while less experienced look further into the code details. In their study on how professional developers comprehend source code, participants explained that understanding the rationale behind the code is very “exhausting”. Latozaz *et al.,* [LaToza et al., 2006] support such findings. They recognize that understanding the rationale behind the code is also a big problem. In fact, a complete understanding

of the software is often not only unnecessary but impossible [Lakhotia, 1993]. This phenomenon can be attributed to the inclination of developers to prioritize understanding the application architecture over delving into intricate code details.

In the realm of software maintenance, maintainers often need to extract specific information and artifact properties from the software codebase. This process involves navigating through the source code or employing queries to retrieve the desired information. By focusing on mining specific information, maintainers aim to gain insights into various aspects of the software system, enabling them to make informed decisions and perform targeted maintenance activities. For example, a maintainer may be interested in identifying the most referenced class within the codebase [Zaidman et al., 2005]. By mining this information, they can gain an understanding of the central components or core functionality of the software. This knowledge is valuable when prioritizing maintenance efforts, as modifications to heavily referenced classes may have a significant impact on the overall system behavior. Similarly, maintainers may wish to determine which attribute within a class is the most accessed. This information can provide insights into the critical data elements and their usage patterns. By identifying frequently accessed attributes, maintainers can focus their attention on optimizing the performance or ensuring the correctness of these crucial data components during maintenance activities. In addition to class references and attribute access, maintainers may explore other specific information or artifact properties. They may seek to identify invoked methods, locate potential code smells or anti-patterns, and detect unused or uncover dependencies between different software components. These mining activities allow maintainers to comprehensively understand the software system structure, behavior, and potential issues such as dead code and code smells classification. This allows maintainers to gather valuable insights and make data-driven decisions during the maintenance process. It helps in focusing their efforts on critical areas, prioritizing tasks, and optimizing their maintenance activities. Additionally, it aids in identifying areas of improvement, potential risks, and opportunities for enhancing software quality and performance. Hence, within the scope of this thesis, our primary emphasis lies on extracting intricate information pertaining to the software source code from the codebase, alongside identifying violations inherent within the code. In the scope of this thesis, the term *violation* encompasses three distinct facets within the software. Firstly, *architectural* violations such as illegal dependencies between software entities and/or suboptimal packaging structures. Secondly, violations pertain to deviations from *naming conventions*. Lastly, violations encompass instances where good programming practices are not adhered to. This forms the basis for our overarching research:

How can we assist software maintainers in understanding the software code violations to eventually make the appropriate decisions?

Within this context of the architectural violations, [Samarthyam et al., 2016] reports one of the main crucial tasks faced by software maintainers in codebase comprehension is the identification and rectification of architectural violations within the software system. [Bass et al., 1998] clarify that these violations can manifest as deviations from prescribed architectural patterns or design principles, compromising the maintainability, scalability, and overall quality of the software. This leads us to the first research question:
RQ1. *What visualization can help assist software maintainers in detecting software architectural violations in client-server software?*

Another significant challenge in software maintenance is the assessment and appropriateness of class names' nature within the codebase. Class names are the primary key abstractions of the software components. Inaccurate or misleading identifier names can hinder program comprehension and impede the maintenance process [Nguyen et al., 2020]. This drives us to the second research question:
RQ2. *How a visualization can help assist maintainers in assessing the regularity of class name conventions and characterizing violations?*

Finally, the interior quality of class implementations plays a crucial role in software maintainability and readability [Lehman, 1980, Martin, 2003]. Maintainability issues, such as overly complex or convoluted code, can hinder comprehension and increase the effort required for future modifications. The third research question that underpins this study is:
RQ3. *How a visualization can assist software maintainers in enhancing the class implementation interior quality?*

Overall, these three research questions are interrelated as they all contribute to the broader objective of assisting software maintainers in understanding and improving software systems. By addressing architectural violations, assessing class name regularity, and enhancing class implementation interior quality, the study aims to enhance the challenges in software maintenance and contribute to the effectiveness of software maintenance processes, leading to more efficient and sustainable software systems. In this thesis, we provide a *set of dedicated visualizations* to answer our research questions, locate and highlight code change opportunities.

1.3 Modeling Bird-Eye View

To tackle the aforementioned challenges, we employ visualization techniques as a means of mitigation. We present a collection of panoramic views referred to as

Bird-eye views[1], encompassing various perspectives of the software, with the purpose of assisting software maintainers in achieving enhanced code comprehension.

In summary, each of the presented views in our study is designed to address each of the research questions previously elucidated (Section 1.2). The first retrieves the high-level structural characteristics of the software, thereby emphasizing and exposing architectural violations. By visualizing the overall system, the second view concentrates on assessing naming conventions and identifying erroneous class names, considering both package and inheritance perspectives. Lastly, we offer a view that encompasses class elements, incorporating inheritance relationships to optimize the retrieval of pertinent information, thus facilitating improved comprehension. This view serves the purpose of detecting violations of good programming practices while providing in-depth insights. Through these tailored visualizations, our study provides software maintainers with powerful tools for tackling the aforementioned research questions and enhancing their understanding of the software codebase. The visualizations are presented as follows:

1. *Client-Server Visualization (Cliservo)* [Agouf et al., 2023]: is a visualization that illustrates the high-level architectural model of Client-Server software based on low-level relations between software components and elucidates its architectural violations;

2. *ClassNames Distribution (CnD)* [Agouf et al., 2022a]: is a package-centered visualization based on the distribution of the vocabulary used in a project taking an inheritance perspective. It presents a comprehensive depiction of the entire system, enabling users to discern suboptimal class names and subsequently undertake the necessary corrective actions;

3. *ClassBlueprintV2 (CBv2)* [Agouf et al., 2022b]: is a redesign of the patrimonial ClassBlueprint [Ducasse and Lanza, 2005, Lanza and Ducasse, 2001] which focuses on individual class structure visual representation. It classifies class components (methods and attributes) into categories and puts light on the call graph between these components. The *ClassBlueprintV2* follows its ancestor by offering a more precise categorization of method types and metrics that help assess class code quality and detect bad practices.

This thesis represents a collaborative effort between the Evref (previously named RMoD) [2] team within the research institution Inria-Lille[3] and the Arolla software

[1]Bird Eye Views: A bird eye view refers to a high-level perspective or overview of a situation. It provides a broad and panoramic understanding, allowing one to see the overall structure, patterns, and relationships without delving into the fine details.

[2]https://rmod.gitlabpages.inria.fr/website/

[3]https://www.inria.fr/fr/centre-inria-de-luniversite-de-lille

company[4] as the industry partner, renowned for their expertise in advanced software development techniques. This unique collaboration bridges the gap between academia and industry, providing valuable insights into the challenges faced by software maintainers in real-world projects.

Furthermore, Evref maintained valuable partnerships with other industrial organizations such as Arolla, Berger Levrault, and Dedalus which further enhanced the diversity of the issues addressed and resolved within this thesis. By harnessing the combined knowledge, this allowed for a broader exploration of software maintenance problems across various domains and contexts, enriching the research with a wider range of perspectives and insights. These additional collaborations allowed us to work closely with these industrial partners, we gained privileged access to actual software projects and had the opportunity to engage with the maintainers directly. This direct interaction drove us to understand the pain points and struggles experienced by software maintainers, facilitating in-depth discussions and the exploration of suitable solutions.

1.4 Contributions

The contributions of this thesis can be summarized as follows:

- *A visualization for Client-Server Architecture* — a novel visualization for the detection of architectural violations validated on real industrial projects (Chapter 3);
- A visualization for naming convention assessment, the *ClassNames Distribution* also helps in identifying naming (anti-)patterns. The visualization is validated using both qualitative and quantitative evaluations on important Pharo projects and a large set of Java projects (Chapters 4 & 5);
- An enhancement of a prominent visualization, the *Class Blueprint* (*Best Conference Paper for VISSOFT22*) (Chapter 6);

1.5 Structure of the Thesis

The thesis is organized as follows:

- Chapter 2 discusses the state of the art in the scope of the elucidated challenges;

[4] https://www.arolla.fr

- Chapter 3 presents the *Cliservo* to analyzing the high-level structure of the software.
- Chapter 4 presents the *ClassNames Distribution* to assessing the correctness of class names.
- Chapter 5 validates the *ClassNames Distribution* using qualitative and quantitative evaluations on both Pharo and Java projects.
- Chapter 6 presents the new generation of the *ClassBlueprintv2* in understanding class interior decors and the call-graph between class entities.
- Chapter 7 summarizes and concludes the work presented in this thesis and proposes future work.

1.6 List of Publications

The list of papers published in the context of the thesis is listed below in chronological order:

1. Nour Jihene Agouf, Stéphane Ducasse, Anne Etien, Abdelghani Alidra, and Arnaud Thiefaine. Understanding Class Name Regularity: A Simple Heuristic and Supportive Visualization. *Journal of Object Technology*, 21:1312–1330, 2022a. doi: 10.5381/jot.2022.21.1.a2
2. Nour Jihene Agouf, Stéphane Ducasse, Anne Etien, and Michele Lanza. A New Generation of Class Blueprint (best paper award). In *IEEE Working Conference on Software Visualization (VISSOFT)*, 2022b
3. Nour Jihene Agouf, Soufyane Labsari, Stéphane Ducasse, Anne Etien, and Nicolas Anquetil. A visualization for client-server software assessement. In *IEEE Working Conference on Software Visualization (VISSOFT)*, 2023.

CHAPTER 2

State of The Art of the Early Challenges in Maintenance

Contents

This chapter provides an overview of relevant works that pertain to the subjects addressed in this thesis. These topics encompass three main areas: *software architecture*, *the evaluation of identifier name quality* (including classes, packages, methods, etc.), and a condensed work on *software visualizations* for assessing source code quality. Through this comprehensive exploration of related works, we aim to build upon existing knowledge, identify research gaps, and contribute to the understanding and advancement of software architecture, identifier name quality assessment, and class code quality.

2.1 Software Architecture

This section focuses on software architecture, which involves the design and structure of software systems. By reviewing this body of work, we aim to gain insights into established practices, challenges, and potential solutions related to software architecture. This section presents the related work to software architecture recovery and the monitoring of the architecture of the software according to distinct software metrics such as architectural smells and violated dependencies. We also present a pile of work around software architecture visualizations.

2.1.1 Architectural Recovery

Several approaches recover software architecture from different perspectives [Ducasse and Pollet, 2009a]. Some tools have been implemented to recover the architecture

of the software such as Rigi [rigi design-recovery], SNIFF [Tak, 1996], Rose [Egyed and Kruchten, 1999], Dali [Kazman and Carriere, 1998], and Software Bookshelf [Finnigan et al., 1997]. In addition to more recent industrial tools that have been proposed for the recovery of software architecture such as MicroART [Granchelli et al., 2017] and ARCADE [Schmitt Laser et al., 2020]. The focus of these tools primarily lies on the comprehension of the architecture. When it comes to addressing architectural violations, however, it may not be their direct functionality.

Other work relies on component-based software, Zhang *et al.,* [Zhang et al., 2010] propose the Dedal architecture recovery model for component-based developed software to support design decisions by separating the representations of architecture specifications, configurations, and assemblies. Allier *et al.,* [Allier et al., 2011] help understand the architecture of a software system by grouping methods in terms of their owner classes to identify the interfaces of service candidates based on the internal structure of components.

A mix of industry and research approaches is beneficial for enhancing the general understanding and analysis of software. When it comes to specific architectural paradigms however like client-server and/or layered patterns, this blend might fall short as they do not account for the unique characteristics, challenges, and needs of each architecture type nor the detection of the architecture violations.

2.1.2 Monitoring Software Quality

When studying the architecture of software some researchers focus on assessing the quality of the software through its architecture. Maria *et al.,* outlines code smells as indicators of architecture degradation [Macia et al., 2012]. Some of the code smells presented in their paper are God classes, Large classes, misplaced classes, and long parameter lists , etc. Meanwhile, the Hotspot Detector [Mo et al., 2015] detects smells at file and package levels. Lippert *et al.,* [Lippert and Roock, 2006] also identified architectural smells at various levels: inside inheritance hierarchies, inside and between packages, subsystems, and layers. The low-level source code analysis is indeed an indicator of the quality of the global architecture of the software.

The work presented by [Macia et al., 2012] in supporting the decision that code smells could affect the overall architecture of the software with regard to classes could be further exploited in the analysis of class code and their impact on the architecture of the software since they present the building blocks of object-oriented software.

In 2020, Randevix *et al.,* [Randevik and Olson, 2020] extended the ArchUnit Java library [1] to support validation of security architectural constraints and called it

[1] https://www.archunit.org/userguide/html/000_Index.html

SecArchUnit. The authors compare SecArchUnit with both SonarQube and PMD and found that their tool was able to detect violations concerning the flow of information in contrast to SonarQube and PMD. Additionally, The tool builds a model of the entire system allowing architects to analyze the dependencies across several classes.

Other commercial tools such as JArchitect [2] help in the static analysis of Java applications. JArchitect is designed to identify quality issues and visualize various aspects of software architecture thereby helping maintainers in monitoring and improving the design of Java applications.

2.1.3 Using Metaphors

Some researchers propose visual aids in the forms of metaphors, for instance, the prominent city metaphor popularized by Wettel *et al.,* [Wettel and Lanza, 2007]. In their work, they present the CodeCity [Wettel and Lanza, 2008] that displays software classes as buildings and packages as the ground foundation on which they are built [Wettel and Lanza, 2007]. CodeCity was later adopted by many researchers to visualize the architecture of software programs [Kobayashi et al., 2013, Pfahler et al., 2020] and was also applied to virtual reality [Fittkau et al., 2015].

In their work, Kobayashi *et al.,* [Kobayashi et al., 2013] introduce the SArF map clustering technique, inspired by the Codecity [Wettel and Lanza, 2007], and which groups together classes with the same features on the same grounds, separated by a street representing the relevance between these features.

Moreover, Sazzadul *et al.,* propose EvoSpaces [Alam and Dugerdil, 2007] approach inspired by the city metaphor but dedicated to C/C++ projects. It displays the architecture and metrics of software in 3D to help quickly understand the software under analysis and the relationship between files. They also offer a night view which displays the execution trace of the software. Another use of the City metaphor is the CityVR proposed by Merino *et al.,* [Merino et al., 2017] which is an interactive 3D visualization tool that implements the city metaphor technique using virtual reality. Dhambri *et al.,* [Dhambri et al., 2008] propose the 3D VESO visualization inspired by the city metaphor which detects and classifies software anomalies with regard to the source code of the software (low-level).

Other original metaphors include Balzer *et al.,* [Balzer et al., 2004] who introduce the Software Landscapes visualization for the structure of large software systems and *Software Feather* [Beck, 2014] which maps class and interface metrics as feathers, one of its main purposes is for users to apprehend to recognize which feather corresponds to which class in the system. Another metaphor derived from nature is *Software Forest* where Atzberger *et al.,* [Atzberger et al., 2021] display

[2] https://www.jarchitect.com

software as a forest by mapping properties of entities such as the size and trend data. Boccuzzo *et al.,* present CocoViz [Boccuzzo and Gall, 2007], a visualization inspired by daily life graphical elements (houses, spears, or tables) to convey information about the program components and the program vulnerabilities.

The use of metaphors to represent software components can be a powerful tool for conceptual understanding and communication, especially in helping stakeholders from various backgrounds (including non-technical ones) grasp abstract software concepts. However, as the statement suggests, this high-level representation can gloss over the intricate details and inner workings of the software, leading to potential shortcomings in deep analysis and understanding of the software and its architecture. Moreover, the lack of a specialized approach that targets certain types of architectures results in a deficiency of detailed information that could be extracted from the software.

2.1.4 Dependencies

Anomalies in dependencies (overly complex dependencies, hidden or implicit dependencies, unintended dependencies , etc) between software components can lead to tight coupling between modules or components making the software harder to understand, maintain, and modify. For instance, a release of a newly tested version of the software eventually crashes because of a dependency on an outdated version of a library. The risks of such a scenario might lead to unanticipated behavior.

Due to the importance of studying the dependencies between the components of the software in both research and industrial companies, researchers propose several visualizations for the analysis of these dependencies between components. Hunter [Dias et al., 2020] which focuses on understanding the dependencies between software components. In their article, Dias *et al.,* employ a distinct visual approach to illustrate the structure and relationships within software components. By using color variations, they differentiate between various file branches, allowing users to quickly identify distinct segments or categories within the software's structure. Simultaneously, the size of individual nodes signifies the volume of source code lines present in each file. Larger nodes point to files with more lines of code, hinting at potentially more complex or significant portions of the software. When a file is selected the visualization highlights all dependencies of the file within the displayed nodes. Such an approach however of studying dependencies is merely for JavaScript projects. Also, the detection of improper dependencies is not automated hence the onus of identifying such dependencies ultimately falls on the user. Tamer's *et al.,* [Tamer et al., 2021] propose a visualization to assess software quality and the dependencies of composed-based JavaScript React applications. Such visualizations use the node-link diagram to demonstrate the connection between software components. Dennie *et al.,* [Reniers et al., 2011] present another approach

called SolidSX that offers a modular analysis of the structure, dependencies, and metrics of software components. While these approaches have demonstrated efficacy, they are specifically tailored for web application components and not suited for object-oriented programs.

In the world of object-oriented programs, Daniel *et al.,* [Daniel et al., 2014] worked on visualizing Java projects using the Polyptychon tool but focusing on the incremental exploration of a project by constraining dependencies. Given a hierarchical information space of software components, Polyptychon constrains the visible dependencies to be related to the child nodes of a specified components node. Erdemir *et al.,* [Erdemir et al., 2011] offer E-Quality, a graph-based visualization that extracts quality metrics and class relations from Java source code and thus does not extract quality metrics of the class body. Fontana *et al.,* [Fontana et al., 2017] propose the *Arcan* tool for the detection of architectural dependencies. Samarthyam *et al.,* [Samarthyam et al., 2016] motivate the need for refactoring of software code smells that decay the system quality from a high-level perspective by analyzing the impact of the evolution of both the Windows operating system and JDK. They report an elevated complexity and unhealthy dependencies between modules. Such approaches however study the dependencies independently from the architecture of the software.

Despite the variety of available tools and methods, many existing approaches analyze dependencies independently from the recovery of the software's architecture. Developers and architects often find it challenging to navigate the intricate web of connections within a software system to ensure all components work cohesively. As a result, the understanding of dependencies remains isolated from the broader architectural context of the software. Furthermore, the focus of some approaches on specific programming languages or project types further limits the applicability of these tools and methods in a broader context, underscoring the need for more versatile and comprehensive approaches for analyzing software dependencies and architecture.

2.2 Assessing Identifier Names Quality

While there is a large body of work on identifiers this section reports a state-of-the-art around distinct identifier names.

2.2.1 Method names.

In a notable contribution to the field, Liu et al [Liu et al., 2019] introduced a pioneering approach centred on the application of machine learning techniques to identify and subsequently refactor inconsistent method names. By employing ma-

chine learning algorithms, the authors adeptly enhance the automation of the renaming process, ensuring greater consistency and adherence to naming conventions across software projects. Nguyen *et al.,* [Nguyen et al., 2020] agree that misleading names in projects confuse developers. They present a tool called MNire to suggest and predict method names by extracting the tokens from the words used in different contexts of the method: the body of the method (i), method parameter(s) type(s) & return type (ii) and the enclosing class name (iii). From each context token, a sentence is formed. The tool then summarizes these sentences from which it suggests a method name using a machine learning model called Encoder-Decoder. To check the consistency of the name they compute the similarity between the newly suggested name and the actual name of the method.

They use the same method set as Liu *et al.,* [Liu et al., 2019] and found that their tool is more efficient in detecting inconsistent naming. The reason for such an improvement is the use of program entity names.

Li *et al.,* [Li et al., 2021] took the previous research to a further stage where the proverb *Show me your friends, I'll tell you who you are* can be applied to method name consistency checking and suggestion. Indeed, they do not only study the method program entities but also their surroundings: the caller and the callee methods and the sibling methods in the enclosing class. They present a tool called DEEPNAME, which was evaluated with a large dataset of over 14M methods, and they found that for consistency checking it improves the state-of-the-art approaches by 2.1% in recall, 19.6% in precision, and 11.9% in F-score.

Allamanis *et al.,* [Allamanis et al., 2015] adopted a more semantical approach to suggest accurate methods and class names. This approach uses a log-bilinear neural language model that learns which names are semantically similar by calculating the statistical co-occurrences of the tokens in the source code. Semantically similar tokens are assigned to locations that they refer to as embeddings. Therefore tokens with similar embeddings tend to be used in a similar context. Furthermore, they use a sub-token model which introduces neologisms–words that were not used in the training corpus. Their results show that their model can suggest accurate method names according to the source code of the method. However, For class name suggestions, positive results were obtained using the sub-token model that generates neologisms.

Alsuhaibani *et al.,* [Alsuhaibani et al., 2021] gathered standards from the literature and asked 1100 professional software developers to determine whether these standards are accepted and used in practice. They found that half of the organizations that participants work for do not define a strict method naming standard.

Isobe *et al.,* [Isobe and Tamada, 2018] worked on retrieving names from obfuscated programs by identifier renaming methods (IRM). In their paper, they focus on restoring method names from their operation code list, especially, de-obfuscating verbs in method names and proposing verbs of similar meanings to the original

verbs.

Alsuhaibani *et al.,* [Alsuhaibani et al., 2021] finding that many organizations lack strict method naming standards highlights the broader issue in naming conventions and in assessing these naming conventions, extending the concern to class name identifiers.

2.2.2 About class names

Butler *et al.,* proposed several studies around class names identifiers. In 2009, they [Butler et al., 2009] found that flawed identifiers in Java classes were associated with low-quality source code according to static analysis. They provide a list of naming style violations (capitalization anomalies, consecutive underscores, dictionary words, excessive words, external underscores, type encoding, long identifier name, naming convention anomaly, number of words, numeric identifier name, short identifier name) and correlated violations as found in FindBug reports. In 2010, they [Butler et al., 2010] extended their previous work on class name analysis to method identifiers: they investigated whether method identifier quality correlates to low quality. They propose diagnostic tests to identify which particular identifier naming flaws could be used as a lightweight diagnostic of potentially problematic Java source code for maintenance. By 2011, Butler et al [Butler et al., 2011a] unveiled an innovative approach, proposing an automated way to tokenize identifier names. This advancement symbolizes the culmination of their ongoing work, reflecting a natural progression from identifying and understanding the issue, to providing practical, automated solutions for addressing identifier naming flaws. This sequence of research and development not only underscores the team's commitment to enhancing code quality and ease of maintenance but also reinforces the critical role of precise and consistent identifier naming in software development.

In the same year, the same authors [Butler et al., 2011b] studied the class naming conventions where they identified conventional patterns found in the use of parts of speech. They also identified the origin of words used in class names within the name of any superclass and implemented interfaces to identify patterns of class name construction related to inheritance. They analyzed 120,000 unique class names of 60 projects and investigated with one project whether classes following unconventional naming schemes should be subject to renaming. They used a PoS (part of Speech) tagger and identified the patterns by which component words from the superclass or implemented interfaces are repeated in class identifier names. In general, while the works of Butler *et al.,* correlate bugs to class names, they do not support the understanding of a naming convention and its violation within a hierarchy, and in the presence of packages that can impose local naming conventions or the creation of subconcepts.

Singer and Kirkam [Singer and Kirkham, 2008] identified a link between Java

class names and the micro-patterns found in the implementation using the approximation that Java class names are of the form `JJ*NN+`, where JJ represents an adjective and NN a noun. The link was based on the assumption that the rightmost noun is an indicator of the class implementation, and no detailed analysis of the class identifier names was undertaken. This study is a component of the research conducted in this thesis, however, their work is limited to the analysis of Java class names and does not encompass a comprehensive examination of Java class naming conventions, patterns, and anti-patterns.

Identifier naming conventions were used by Abebe *et al.,* [Abebe et al., 2009] to identify smells. The smells are predicated on deviations from suggested identifier naming conventions that arise from programming conventions, and, to a lesser extent, deviation from established conventions arising from identifier naming practice. Again, none of the previously mentioned work discusses the regularity of naming conventions and their violation patterns. Moreover on class names, Anslow *et al.,* [Anslow et al., 2008] present a short paper on class name visualizations: they use a tag cloud to compare class words used in class names of Java 1.1 and 16 and a tree map of the ordering of words used in class names of the Java API specification. Yano *et al.,* [Yano and Matsuo, 2015] adapted TF-IDF (a frequency-based information retrieval filtering technique that extracts characterizing words for a document in a group of documents) and extended SArF [Kobayashi et al., 2013], a CodeCity like 3D [Wettel and Lanza, 2007] visualization, and proposed better map labeling. Their visualization is related to lemmas of class/method names. This visualization however does not help one to understand the inconsistencies in class names.

Despite all the deep dive into class identifiers, current research has not touched much on the details of naming conventions and how they are violated within a hierarchy, especially when dealing with packages that bring their own set of naming rules or create subconcepts. There's also a noticeable gap in comprehensive study beyond just Java class names, extending to a detailed analysis of class naming conventions, patterns, and anti-patterns. Finally, the visualizations used in the studies do not aid in understanding the inconsistencies in class names nor the detection of the naming (anti-)patterns.

2.3 Software Visualization

Visualizations facilitate program comprehension because they provide a graphical view of the software rather than an alphabetical sequence of source code text. There is an extensive body of work related to the software visualization [Caserta and Zendra, 2011, Merino et al., 2019, Spence, 2001, Stasko et al., 1998, von Landesberger et al., 2011, Ware, 2000]. Additionaly, several articles provide or visualize information on software files, classes, and/or packages. Many of these approaches

address software co-change, looking at coupling from a temporal perspective [Abdeen et al., 2014, Beyer, 2005, Ducasse et al., 2006, Eick et al., 2002, Froehlich and Dourish, 2004, Storey et al., 2005, Voinea et al., 2005, Xie et al., 2006]. Kienle and Müller [Kienle and Müller, 2010] present requirements for reverse engineering tools and their evaluation. According to Ghanam *et al.,* [Ghanam and Carpendale, 2008] researchers and architects are more interested in visualizing the high-level design of the software, which motivates the work of this thesis in presenting Bird-eye views of software programs based on distinct problematic maintenance propositions.

Marcus *et al.,* [Marcus et al., 2003] propose a matrix-based representation of files. Each dot (small box) represents a line, and its color conveys one kind of semantic information (if statement for example). They propose a 3D version of the matrix-based structure. The idea behind the matrix-based presentation is to be able to offer a compact representation of code entities.

Lanza's Polymetric views enrich simple program visualizations such as inheritance trees with metrics [Lanza and Ducasse, 2003]. In Polymetric views, the shape of the classes can represent class metrics such as the number of instance variables, methods, and lines of code. Polymetric views leverage multiple software metrics to provide a comprehensive visual representation of software entities and relationships, enhancing understanding, and facilitating analysis and decision-making in software development and maintenance. Fernandez extended VisualIDs as a glyph technique to cope with structural software elements. The authors use them to identify classes with the same dependencies and classes with a similar set of methods [Fernandez et al., 2016a]. Glyph could be used to convey class identifiers. Furthermore, Ignacio *et al.,* [Fernandez et al., 2016b] extend visualIDs as a glyph technique to cope with structural software elements. The authors use them to identify classes with the same dependencies and classes with a similar set of methods. These visualizations are undeniably of significance and have demonstrated their efficacy, despite not centering on the specific code of the class itself. In essence, while Lanza's Polymetric views [Lanza and Ducasse, 2003] and Fernandez's glyph technique [Fernandez et al., 2016a] employ different visualization strategies, both contribute significantly to the field of software visualization. They offer unique, yet complementary, perspectives and tools for analyzing and understanding software structure.

Additionally, the ClassBlueprint [Ducasse and Lanza, 2005, Lanza and Ducasse, 2001] visualizes the internal implementation of a class in terms of method calls and field accesses– in addition, methods are annotated with colors giving semantical information about the methods. While the ClassBlueprint has gained recognition in the field of program visualization, the model has become antiquated as it does not accommodate the representation of newly emerged concepts in classes such as dead code identifications, tests and non-tested methods, cyclomatic complexity of meth-

ods , etc. Anslow *et al.,* [Anslow et al., 2013] propose the SourceVis visualization platform which is designed for multiple users supporting multiple visualization types and displaying such visualizations on large multi-touch tables. SourceVis proposes a reimplementation CLASS BLUEPRINT based on its original layers. The SourceVis platform aims to consolidate multiple visualizations for its users. Given that a decade has passed since its original conception, updating the collection of visualizations is advisable. Moreover, the authors might consider introducing new visualizations to reflect the platform's evolution and current trends.

2.4 Conclusion

In terms of the state-of-the-art of architectural recovery, some approaches are considered outdated by researchers (such as Rigi [rigi design-recovery]), hence the emergence of new approaches to recover the software architecture. Including the inception of some studies that monitor the quality of the software using metrics from the low-level source code which influence the high-level conceptualization of the software. Other innovative approaches view software concepts through real-world objects using metaphors to convey information about software. Such aesthetic and illustrative representation can gloss over the intricate details and inner workings of existing architectural patterns. This emphasizes the need to recover software architectures according to their patterns type using visual aids, meaning customized visualizations for different types of architectures (client-server, layered, micro-services , etc) which could give more insight into the architectures of software and the different violations. In addition, to evaluations on real-world projects as Merino *et al.,* [Merino et al., 2018] state, most visualizations are poorly evaluated as they found that 62% of the proposed software visualization approaches (SOFTVIS/VISSOFT) either do not include any evaluation or include a weak evaluation (*i.e.,* anecdotal evidence, usage scenarios).

Pertaining to software identifiers more particularly class name identifiers, many researchers propose approaches that analyze class names, while others also offer visualizations. According to the literature, however, none of these approaches takes into account the influence of the hierarchy on the class names nor the packaging. The class name is an indicator of the responsibility of the class however because its behavior is primarily shaped by the hierarchy to which the class belongs as well as its membership to the package such characteristics can have an impact on the class name. Additionally, none of the previously mentioned approaches address the detection of patterns and violations of naming conventions. Naming conventions are adopted by companies however it is literally unclear whether these conventions are in practice respected or violated.

Ultimately, merely studying the outside relationships of the components of the software overlooks the details that could enhance the global architecture of the software. Because classes represent the building blocks of object-oriented software, assessing the quality of their codes gives more insight into the holistic view of the software and its architecture. One of the pillar approaches in understanding the internal structure of class codes is the ClassBlueprint visualization [Ducasse and Lanza, 2005, Lanza and Ducasse, 2001]. The ClassBlueprint visualization goes beyond mere structural visualization, it offers a deep dive into the responsibilities and behaviors of classes based on their hierarchy and packaging. This aspect is crucial in understanding how each component fits and functions within the grand architectural layout, further enriching the bird's eye view. Even so, it has been more than twenty years since the visualization was first presented along with the extensive evolution in practices applied to class code, which the visualization did not pursue.

CHAPTER 3

A Visualization for Client-Server Architecture Assessement

Contents

Maintaining large legacy systems often requires understanding their architecture [Clements et al., Lehman and Ramil, 2001, Lung and Kalaichelvan, 2000a, Sommerville and Sawyer, 1997]. This is important since legacy system architecture decay over time and architecture violations may dramatically impact planned renovation actions [Medvidovic et al., 2003a, Perry and Wolf, 1992a]. Merely reading source files is time-consuming and often highly inefficient.Visualizations have been proposed as a tool to support architecture understanding [Diehl, 2007b, Knight and Munro, 1999, Koschke, 2003, Langelier et al., 2005, Wettel and Lanza, 2007, 2008]. Such visualizations, however, do not take into account the specificities of client-server applications and thus miss the identification and understanding of such software architecture violations.

In this chapter, we propose CLISERVO, a new visualization to help software maintainers detect architectural violations in client-server systems. CLISERVO classifies client-server entities into different levels of dependencies, shared entities, or ambiguous entities (*e.g.,* entities that belong abnormally to different parts) and highlights illegal dependencies between layers (Section 3.3). We first explore

the role of software architecture in the context of software maintenance and investigate the needs that drive the selection of different software architectures (Section 3.1). Additionally, we provide a comprehensive overview of the layered architecture commonly adopted in client-server systems (Section 3.2.2). We present both visualization configurations from different viewpoints (Sections 3.4 and 3.5) and extract insights into the application of the CLISERVO on our industrial partners' software. Indeed, we validate the visualization on three projects of our industrial partners. (Section 3.6). We further discuss the threats to the validity of the experiment (Section 3.8) and the conclusion in Section 3.9. This work has been published in the *IEEE Working Conference on Software Visualization (VISSOFT2023)* [Agouf et al., 2023].

3.1 Investigating the Role of Software Architecture in Software Maintenance

During the software lifecycle, the architecture often becomes inaccurate, resulting in architectural erosions [Medvidovic et al., 2003b, Perry and Wolf, 1992b]. Consequently, recovering the existing architecture of legacy software is challenging [Ducasse and Pollet, 2009b]. Tools have been implemented to recover architecture such as Rigi [rigi design-recovery], SNIFF [Tak, 1996], and Rose [Egyed and Kruchten, 1999]. However, software architecture is a very fuzzy notion that lives mostly in the mind of the beholder. There is no one-size-fits-all, universal, definition of what is software architecture (see for example the 4+1 model [Kruchten, 1995]). Moreover, software architecture is materialized by coding conventions (such as class names, package dependencies, etc) that are often not documented, not explicit in the code, and violated by programmers [Koschke and Simon, 2003]. Therefore, a high-level design description plays, de facto, an important role in successfully understanding and reasoning about large and complex software systems [Clements et al., Lung and Kalaichelvan, 2000b].

Software architecture is a critical aspect of software system design, encompassing the foundational decisions that shape its overall structure and behavior. It defines how the components, interactions, and functionality of a system are organized and managed. The choice of software architecture is influenced by various criteria and considerations that developers carefully evaluate. Among the different software architectures available, several widely used ones stand out. These include Monolithic architecture, Microservices architecture, Service-Oriented Architecture, and Client-Server Architecture, etc. Each architecture offers a distinct approach to structuring and managing software systems, addressing specific requirements and challenges. The selection of a software architecture is typically a topic of discussion and analysis in the early stages of software development. De-

velopers engage in comprehensive assessments to determine the most suitable architecture for their project. Factors such as system requirements, complexity, scalability, integration needs, and development team expertise are taken into account during this decision-making process. As the software evolves and requirements evolve, developers may decide to migrate from one architecture to another. For example, a development team may opt to migrate from a Monolithic architecture to a more scalable and modular Microservices or Client-Server Architecture, following established architectural patterns and practices such as integrating a communication protocol between the client and the server.

3.2 Client-Server Architecture

The client-server architecture is widely recognized as a reliable and efficient approach for designing and implementing software systems [Hanson, 2000]. It revolves around the separation of the user interface (client) from the back-end processing and data management (server), enabling clear role differentiation and facilitating effective collaboration between client-side and server-side components. Some of the key factors that drive the adoption of client-server architecture include the scalability of the software since it distributes the processing functionalities and data management across multiple servers which enables the systems to handle increasing loads and accommodate a growing number of users without compromising its performance and responsiveness. The client-server architecture offers modularity in maintenance because of the separation between the client and server components, thus, developers can update and enhance the user interface independently of the server-side logic, simplifying the development process and facilitating easier maintenance and updates.

3.2.1 Layered Design

The client-server applications typically exhibit a layered architecture where each layer performs specific functions and interacts with adjacent layers in a well-defined manner. Each layer is established based on predefined rules that are determined and maintained by the software maintainers. The layered nature of client-server applications provides several benefits. It promotes modular design, allowing developers to focus on specific layers and components independently. This modularization enhances code maintainability, reusability, and testability. It also facilitates system scalability and flexibility, as changes or updates in one layer typically do not require modifications in other layers.

Figure 3.1 depicts an example of architectural layers in one of our industrial partners projects. The project is composed of two main components: the Client

application and the Server application. These two components establish communication and exchange information using a designated communication protocol and corresponding interfaces. Each component, namely the Client and Server applications, is further structured into distinct layers, with each layer being assigned specific responsibilities. In the Client application, the User Interface (UI) layer interfaces with the Business layer (front-end), which, in turn, interacts with the service interfaces. The interfaces within the Client application facilitate communication with the corresponding server-side components. The Server application, on the other hand, comprises multiple layers, each fulfilling a specific role. These layers include the implementations of the services, responsible for handling specific functions or operations, the Business layer (back-end) which encapsulates the business rules and logic, and the Data Access Objects (DAOs) layer responsible for managing the transfer of data objects.

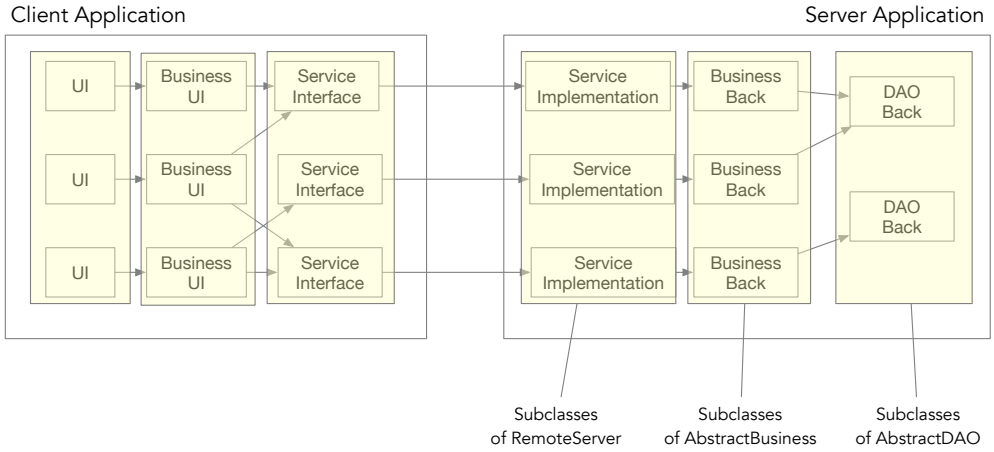

Figure 3.1: A Typical example of the Client-Server Architecture layers

This layered organization promotes a clear separation of concerns and enables modular development and maintainability. It allows for independent development and testing of different layers, facilitating easier updates and modifications. By dividing the application into well-defined layers, the project maintainers can focus on specific functionalities within each layer, leading to improved code organization, reusability, and scalability.

3.2.2 Data Transfer Objects taking Part in Client-Server Applications

The client-server architecture is a distributed application structure that divides tasks or workloads between providers of a resource or service, called *servers*, and service

requesters, called *clients*. However, such an architecture mostly relies on three main parts. The *client* part requests content or service from a server. In contrast, the *server* part runs programs to answer client requests. Finally, the data transfer objects (DTOs) are resources transferred by the client part to the server to execute the programs on a given data or on the opposite transferred by the server to the client to display them corresponding to the *shared* part of the system. In the next section, we show that assigning a class of a given layer is a challenging task.

3.2.3 Challenges for Layers Identification

These three parts (client, server, and shared) may eventually be well-identified during the design phase, through architectural rules. Some rules are structural *e.g.,* a DTO inherits from a specific class or a client class implements a dedicated interface. Other rules concern the behavior, *e.g.,* a class for which at least a method is called by a server class, belongs to the server part or, a class for which at least a method calls a client class is considered client side. However, these rules are not always documented. And even so, over time and several evolutions, these rules are violated. Thus in practice, the separation between the different parts is fuzzy and is no more clearly identified. For example, on real systems, it is not rare that some classes belonging to a (sub)package of a client may in fact play the role of server and vice versa. Some DTOs may be only used by one part, or even not used by either part. Finally, other elements that DTO may be shared between the client and the server or may be so complex that some of their methods play the role of the client and others of the server. The belonging of these elements to exactly one part may often need further investigation.

If a class satisfies only an architectural rule defining a layer, it is easy to assign it to this layer. In case a class satisfies several rules corresponding to different layers, there is an *ambiguity* and it is not possible to clearly identify the layer the class belongs to: violations are thus observed, since normally, a class should belong to a single layer.

In the context of a future migration such as the migration from a client application to another (*e.g.,* from GWT to Angular [Verhaeghe et al., 2021]) or the decomposition into micro-services of the server part, the shared elements which are not DTOs or the violations between client and server parts are problematic. In practice, they correspond to violations of software architectural rules.

3.2.4 Layers in the Server Part

Even when the client and server parts are clearly separated, for example, because they structurally belong to two different projects, architecture violations may occur. Indeed, the server part may be decomposed into several layers, such as the server

interacting with the client part, the services corresponding to the core program, and the database access objects (DAOs) corresponding to the interface between the server and the database. Each part corresponds to a layer and the communications between them are strictly defined. The server elements can call services, which in their turn can call DAOs. All other communication between layers is considered a violation. For example, a DAO is not allowed to have dependencies on services or server elements.

Due to the challenges to assign classes to different architectural parts and understanding architecture violations, there is a need to support maintainers to understand why a class may play different roles and how rules are violated.

3.3 A Dedicated Client-Server Architecture Visualization: CLISERVO

We propose a dedicated visualization, CLISERVO, to support client-server navigation and architectural violation identification. Figure 3.2 displays an annotated version of one of the two views of CLISERVO applied to a real industrial system. The remainder of the chapter presents in details the different aspects of the visualization: Its two *configurations*, its two *viewpoints*, and the *levels* that can be applied to support the understanding of application architects.

This visualization structures the software into layers and uses the traditional node-link diagram to connect the layers' components. Such layers rely on rules that can be *structural i.e.,* a class belongs to a specific hierarchy, or *behavioral i.e.,* a class calls or is used by another one. The constraints imposed by the membership to a specific hierarchy are stronger (since the behavior and the state of the classes are shared) than those resulting from the use of/by a specific class. Consequently, we consider that the assignment to a layer is sure when it relies on a structural rule, and uncertain when it is based on behavioral rules. such rules take into account the way the different framework expresses client-server relationships - *e.g.,* inheritance to certain classes as in frameworks such as GWT. In addition, these architectural rules are often adapted to reflect the knowledge of the system based its maintainers.

CLISERVO offers two configurations: first, the *Big-Picture* showing all the parts (client, server, shared and purgatory) (See 3.3.1) and the *Server focus* one showing on the server side (See 3.3.2), as we will show now.

3.3.1 Configuration 1: Big-Picture with Four Main Layered Parts

In the Big-Picture configuration, CLISERVO splits the system into four main layered parts (see Figure 3.3): *Client side*, *Server side*, *Shared Space* and *Purgatory*. The client and server parts are layered because each element implied in a relation

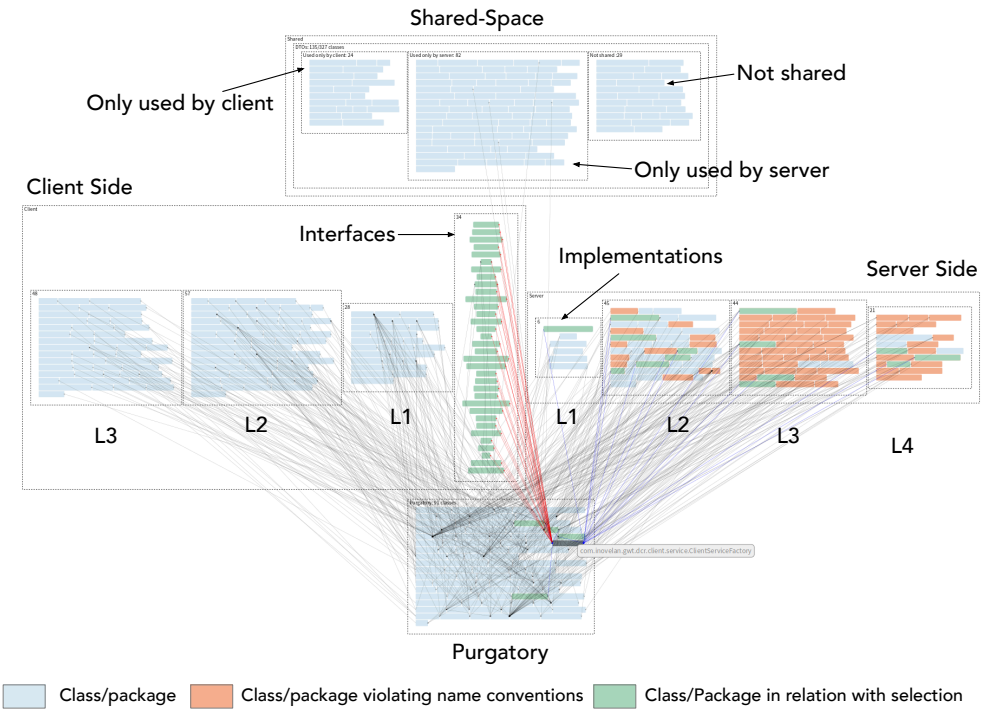

Figure 3.2: An annotated version of CLISERVO Big-Picture from Violation Viewpoint applied to WD: Four parts (*Client*, *Server*, *Shared* and *Purgatory*) and their relation/violation (L = Level). . Note that the class names are deliberately blurred on purpose to respect the company constraints.

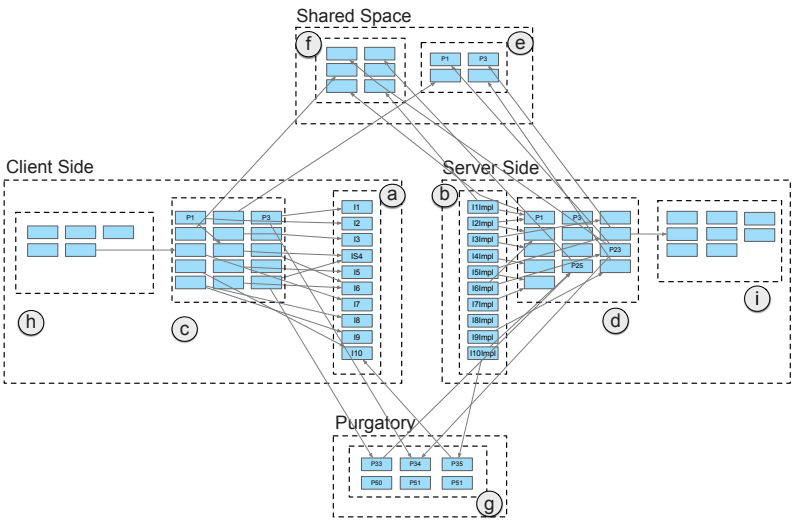

Figure 3.3: Big-Picture in CLISERVO: It features four parts (Client Side, Server Side, Shared and Purgatory) – the Server side includes layers b, d and i; the Client side, a, c, h.

may be called by other elements (acting as subsequent layers). We explain each part:

- **Client part:** This layered part contains the core set of classes or interfaces, establishing the communication with the server part (a). In addition, it includes the classes using this core, either directly or indirectly through other classes (c, h). If the core part relies on a structural architectural rule, the remainder is built successively, level by level by considering first the direct clients of these original interfaces and then the direct clients of these resulting classes and so on.

- **Server part:** The server part has a similar structure as the client part. The core part is composed of (implementation) classes that enable communication with the client part (b). The remainder of the server part contains classes used by these original implementations directly or indirectly (d, i). This part is built successively level by level.

- **Shared part:** This part contains DTO (Data Transfer Object) classes *i.e.,* software resources which *can* be used by both client-server entities (e, f). Hence, this part contains DTOs that are meant to be shared but may not be actually shared in the project.

- **Purgatory:** The purgatory part gathers elements that are not DTOs and whose classification into the client or the server parts is not clear (g). These elements are used by at least one entity of the server part and use at least one entity of the client part. Such entities are in relation with both client and server entities leading to ambiguity about their layer affiliation. Further analysis of such entities is needed to be attributed to their correct layer.

3.3.2 Configuration 2: Server-focus with Three Layers

It is possible to focus only on the server part, which is also composed of layers: *Server*, *Services* and *DAOs*.

- **Server:** This layer corresponds to what has been previously described in the Big-Picture visualization (Section 3.3.1). It includes the core part, *i.e.,* the (implementation) classes enabling communication with the client part, and contains classes directly or indirectly used by the core (L1 and L2 respectively).

- **Services:** This layer corresponds to the services of the server application. It relies on a structural rule and thus corresponds to a specific class hierarchy.

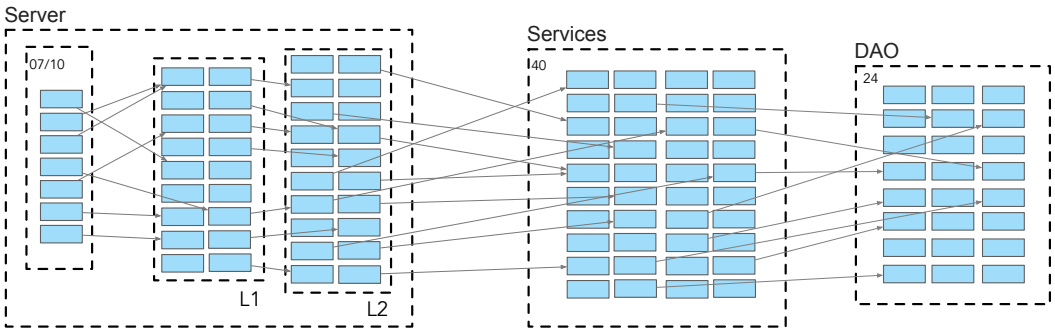

Figure 3.4: Illustration of the visualization Server-focus.

- **Data Access Object (DAOs):** Entities of this layer represent the classes accessing the database of the software. They also result from a structural rule and represent a specific class hierarchy, such as the AbstractDao class in *Hibernate*.

3.3.3 Two Viewpoints for each Configuration

In addition to the *Big-Picture* and *Server-focus* configurations, the visualization features two viewpoints: *a general component overview view* and a *architectural violation view*. Both viewpoints use the same graphical elements (nodes and arrows) to quickly convey information about the software, the main difference is that the second viewpoint does not display all elements of the software but the necessary information to detect violations.

- **Big-Picture in architectural Violation View.** When looking at *violations* applied to the Big-Picture overview, only the elements in relation to the purgatory are displayed in the client and server parts. Only the classes used by the server part and using the client part are considered ambiguous and are put in purgatory. However, all the elements in relation to these classes require specific attention to determine which relation is a mistake to remove a class from purgatory and put it either in the client or server part. In addition, the DTO part is decomposed into three sets, (i) the DTOs only linked with the client part, (ii) the DTOs only linked with the server part, and (iii) the DTOs linked with neither the client nor the server part. Indeed, since DTOs are considered shared, these three cases correspond to violations. Finally, since the client and server parts mainly rely on behavioral architectural rules, for each class, we check if its position in the client or server part is consistent with the name of the package in which it is. If it is not the case, there is a violation that is expressed by displaying the class with another color. For example, a class in the server part cannot be in a subpackage of a client package.

- **Big-Picture in General Component View.** The Big-Picture in the General

Component View depicts ALL entities of the software, meaning entities in violations and entities that are directly or indirectly in relation with the core components (interfaces in client and their implementations in server). Including entities in purgatory. In this mode, the DTO part is not decomposed into three layers but groups all DTO entities in one layer, highlighting the ones in violation in a different color. This view could be of use in case the user wishes to see the violations present in the system and how they interact with the other components.

- **Server-focus in architectural Violation View.** In the Server-focused configuration from the architectural violation viewpoint, only the server elements calling directly a DAO (without going through a class of the Services layer) or called by a Services or DAO element are displayed in the visualization. Similarly, only the service class called by a DAO is displayed. All other elements respect the architectural rules and consequently are not displayed to simplify the visualization and scale.

- **Server-focus in General Component View.** This view focuses on the server part and offers a view of the entire entities in the layers of the server. Including the different violations explained in the violation view. All elements with respect or deviating from the architectural rules are consequently displayed.

In addition to the aforementioned viewpoints, the visualization also offers the hybrid view that allows the selective display of violation view and general component view in different layers of the software as shown in Figure 3.6.

3.3.4 Nodes, Links, and Interaction

Inside layers, we place nodes to represent classes or packages. Such nodes are connected by links.

Nodes. The visualization presents classes, interfaces, and packages as nodes. Except if the communication protocol or the used framework imposed that the communication between the client and the server parts is performed through interfaces, the other nodes inside the layers correspond to classes or packages. For readability and scalability reasons, if the number of classes to represent in a layer is too high, classes are grouped in their corresponding packages. This threshold can be modified in the settings of the visualization. By default, we use 100.

The visualization is modular in displaying the following levels of the next distances of the core entities, meaning that the decomposition of the layers into levels is built progressively level proceeding or following a level for client and server parts, respectively. Consequently, new ambiguous entities in Purgatory may be added at each iteration when a new distance is computed.

Links. Given the use of the conventional node-link diagram in the visualization, it is evident that an arrowed line means the dependency between interconnected nodes. To avoid overloading the visualization with links, by default, only the dependencies from or to classes of the purgatory or in violation of a rule are displayed. For instance, in Figure 3.2 the class ClientServiceFactory (selected class in purgatory) depends on all 34 interfaces defined on client part, hence the red arrows and green color of nodes. On the other hand, 14 class nodes on the server part depend on ClientServiceFactory, hence the blue arrows and green color of nodes.

Interactions. Since the visualization is built as a tool, different interactions with its elements are provided. We list a few of them:

- The user can explore (sub-)nodes or links *i.e.,* investigate their properties and possibly access the corresponding source code.
- The user chooses to hide or show the links coming out or into a single node.
- The user chooses to hide or show all links coming out or into a layer thus improving clarity and to not clutter the visualization with links.
- A package node can be expanded to show its internal sub-nodes. Such an expanding feature is also applicable to groups of nodes in layers.
- The user can progressively display the following nodes of the next dependency level by interacting with it.

The visualization relies on an abstract representation of the software using the Famix metamodel Ducasse et al. [2011] and is integrated into the Moose metaplatform Anquetil et al. [2020].[1] Once the model is built, we import it into the Moose platform.

3.4 Big-Picture Visualization Configuration in Action

In this section, we present the application of CLISERVO to industrial systems. For confidential reasons, we changed the name of the software systems and cannot mention the name of the companies. Moreover, to respect the company constraints we blurred the names in the figures. While during our analyses we applied different configurations of CLISERVO, for space reasons, we report the general overview of CLISERVO visualization viewpoint on OJ project and the violation viewpoint on WD project. These projects are presented in the next sections.

[1]Moose is an extensive platform for software and data analysis: https://moosetechnology.org

3.4.1 OJ: Distribution System of Updates

OJ is a system of updates distribution, built for a list of clients such as small and big city halls. It was initially developed by a single developer in 2008 replacing an old updates downloading system. It was continuously maintained by the same developer then maintainers changed throughout the years. However, only ten maintainers were responsible for this system throughout the years, with few system evolutions because the system was not exposed to changing needs. It was conceived for one thing (distribution of updates) which is still functioning correctly. The system is decomposed into two projects, the client and the server project. The client project is built using GWT, and the server project is built in Java. The two projects use the RPC protocol automatically generated by the framework to ensure communication between the client and the server.

OJ counts 3277 classes spread in 599 packages. There are 3 packages named client and 2 named server packages. Altogether, the client packages contain 1051 classes. The server packages count 102 classes.

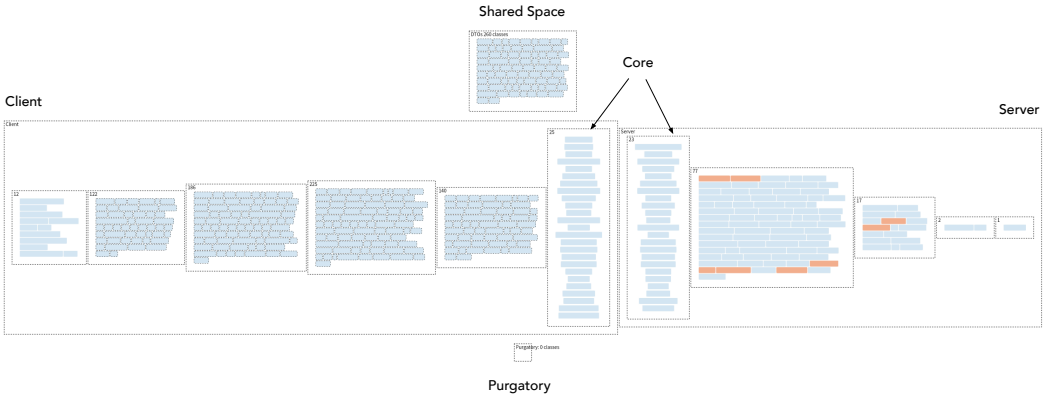

Figure 3.5: The Big-Picture of CLISERVO visualization for the OJ software system in Global Component Overview (all nodes of the project)

Analysis: Big-Picture in general component overview viewpoint. In Figure 3.5, we use the general component overview, where all the components are displayed. We added all levels to the visualization, in addition to the core interfaces and implementations respectively. The server part holds only four levels, whereas the client has five.

There are two main dashed boxes each containing at most five dashed smaller boxes. The big left dash box corresponds to the client part. The right one refers to the server. In each of these two boxes, the vertical dashed boxes on the right for the client and the left for the server correspond to the core implementing the communication protocol. The other dashed boxes correspond to the indirection levels successively computed. In the server part, some classes are in red, meaning that they are considered in the server part, but belong to client packages.

The DTOs have not been separated into several groups since they are all used by both the client and the server parts.

The purgatory is empty meaning that all the communications between the client and the server parts are done via the interfaces and the implementations as foreseen by GWT. Consequently, the separation between the two parts is pretty clear and net.

The general architecture quality of the OJ system (resulting from its specific development) is a kind of Graal for a client-server application: there is no serious and random violations as in systems such as the one shown in Figure 3.2.

3.4.2 WD: a Multidisciplinary Healthcare Management System

With WD, doctors can plan meetings, enrol patients in a session, register proposed decisions, and validate and publish decisions. This software system is largely used in hospitals and has been developed around 18 years ago using a client-server architecture. The client part has been developed with GWT, the server part is in full Java, and the communication protocol uses RPC. The system evolved; new functionalities have been added. But also, technology has changed: GWT is no more maintained by Google. New versions of browsers do not support well the Typescript code transpiled from Java. It becomes urgent to migrate the client part to a new technology. In parallel, the company wants to modify the server part. However, after multiple evolutions, the client-server architecture drifted. It is impossible to just remove the client parts and replace them.

WD counts 6030 classes spread in 1044 packages. There are 8 packages named client and 7 named server. Altogether, the client packages contain 3016 classes. The server packages count 389 classes.

Analysis: Big-Picture in violation viewpoint. As for OJ, in Figure 3.2, we added four levels to the visualization, in addition to the core interfaces and implementations respectively. However, due to the state of the communication between the client and the server parts, we used the violation viewpoint.

On top, there are the DTOs. As explained before, in the violation viewpoint, they are separated into groups between those that are used only by the client (24), those only used by the server (82) and those not shared (29). The other DTOs (192) are not displayed in the visualization since, as expected, they are used by elements of the client and the server parts. Only the link from the purgatory at the bottom and the DTOs are displayed in the figure.

At the bottom, there is the purgatory containing the classes whose categorisation is ambiguous. At this level of indirection from the core, there are 91 classes. They are used by the server part and they use the client part. Only the links from the purgatory to the client classes or from the server classes are displayed. Since the figure presents the violation viewpoint, all the classes in the client (respectively server) part are targets (respectively source) of such a link. We see that the sepa-

ration between the client and the server parts is unclear. The communication does not always respect the framework and goes through other classes represented in purgatory. In addition, a lot of classes in the server part belong to client packages.

Note that the violation viewpoint, in this case, enabled the architect to focus on interest points to correct the architecture. When the violation viewpoint shows no more entity, the separation between parts is clear, and adopting the general component overview makes sense for OJ.

3.5 Financial System: Server Focus Visualization Applied

EGF is a financial management system for local authorities. Once again, for confidential reasons, the name of the application has been changed and the figure anonymized. This application has been developed in full Java using the RMI protocol. The client and the server parts are physically separated into two different projects. This physical separation has consequences on the software logic: The categorization of each entity is clear, it either belongs to the server or the client part. Consequently, we focus on the server part, containing 4028 packages among which 180 are named dao and 352 service for a total of 11424 classes.

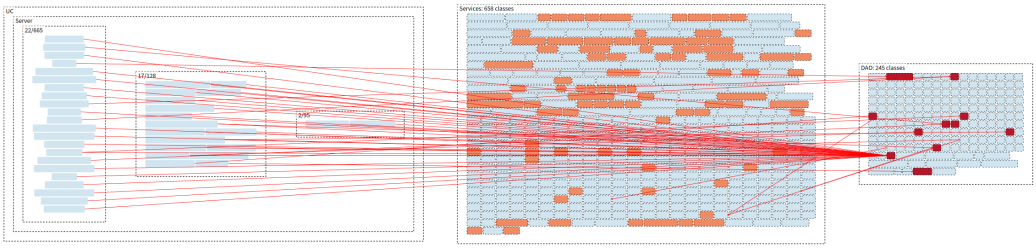

Figure 3.6: The server focus of CLISERVO visualization for the EGF software system from the Violation viewpoint (limited to nodes participating to the violations).

Analysis: Server focus in hybrid viewpoint. Figure 3.6 shows the Server-Focused visualization on the EGF industrial case.

To better highlight the differences, the visualization is displayed in an hybrid viewpoint: The server part is displayed in the violation viewpoint – this is why the user can see the ratio of displayed classes in each layer. On the opposite, the services and the DAO parts are displayed using the general viewpoint. All the entities of these parts are displayed. Since they are too many, classes are grouped by packages.

When the services bypass the services objects and talk directly to the DAO, these DAO are displayed in dark red, or more precisely, the packages, containing

these classes are like so. In addition, to know on which Service objects to focus among all, entities in violation are in orange (those calling a server entity or called by a DAO).

3.6 Cliservo: Mining Architectural Insights on Industrial Projects

In this section, we report our analysis of the results of CLISERVO on industrial projects. This was validated with the current maintainers of each project.

The validation followed the steps: (1) the authors performed the analysis of the three systems using CLISERVO. This phase lasted half a day. The analysis produced a list of potential violations and points to clarify; (2) all the raised points were discussed and validated with the current maintainers of the systems during half a day each. This phase leads to the opening of bug tickets or the identification of serious and larger problems for some cases (*i.e.,* migration to other front-ends). The subsequent sections summarize our findings.

3.6.1 WD results

The results of the WD project are depicted in Figure 3.2. The tool helps the maintainers to reclassify the messy package structure where client and server names were mixed in ad-hoc fashion. In addition, 29 unused DTOs were identified, 24 DTOs only used by the client, and 82 only used by the server. Furthermore, 91 purgatory elements were identified:

- 17 entities using only one client entity and only one server entity,
- 4 entities using only one client entity and used by very few server entities,
- 13 entities using several client entities but only used by one server entity,
- 2 entities using several client entities but very few server entities,
- 4 entities using both client and server entities and being used by very few server entities,
- the rest are entities automatically categorized in purgatory because of indirect relations. For instance, an entity using only clients is used by an entity in purgatory.

Note that the server entities being used are mostly entities categorized in the server part but in client packages depicted in orange in Figure 3.2. The tool helped

current project maintainers identify the idle DTOs and those in violation because they are not actually shared between both client and server parts. Moreover, they were mostly interested in the ambiguous entities in Purgatory since their belonging represents an important problem for them to eventually investigate and classify.

3.6.2 OJ results

The results of the OJ project are depicted in Figure 3.5. The OJ system is an example of a well-conceived project with few evolutions and a limited number of different maintainers throughout the years. CLISERVO showed some classes in client packages are referenced from the server side (colored in orange in Figure 3.5). During the CLISERVO validation, the maintainers explained that these classes are referenced by server entities to avoid code duplication: Instead of creating a new class with the same code, maintainers preferred to refer directly to these classes.

The visualization also showed classes with the DTO keyword in their names appearing in the server side instead of on top in the Shared Space. This means that they do not follow the rules of inheritance on which the Shared Space is built. This was a false positive of the way the classification is done by our tool. Indeed the maintainer explained that such classes do not belong to the OJ system but to a framework developed by the company from GWT components to share common functionalities over projects.

3.6.3 EGF results

The tool helps the maintainers to spot violations. An exceptional class from the DAO part is accessed by *all* the server implementation classes. One of the maintainers explained that each of their applications has some known and accepted design issues. This violation is one of them. However, CLISERVO also revealed seven other direct links from the implementations to the DAOs layers which are indeed considered as relations not respecting the architecture of their software.

Moreover, the tool showed eight direct links from classes in the first level of the server (classes directly used by the implementations) to the DAOs. One maintainer confirmed that such links do not represent violations because the calling classes found in the server layer are de facto classes that belong to the Services layer, however, organized according to a different structural packaging set (classes of the service subpackaging). Changing the rule of the Services layer to include classes with such a structure helps avoid these false positives. Nonetheless, to our knowledge of the system and communication with its maintainers, the naming conventions are not perfectly respected so adjusting the tool to an inconsistent rule might produce more unintended consequences.

Finally, CLISERVO helped the maintainers identify five confirmed violations coming from the DAOs to the services layer caused by the intertwining entities. The fact that the code was too coupled and spaghetti-like made it impossible for the maintainer to decide what to do.

3.7 Discussion

This chapter introduces a novel visualization technique, referred to as CLISERVO, which facilitates the recovery of client-server application architecture. By extracting structural and behavioral information from the software source code, CLISERVO constructs the different layers of the client-server components. Furthermore, the visualization incorporates Data Transfer Objects (DTOs) present in the codebase, identifying both violations and adherence to architectural rules. Ambiguous entities that raise questions regarding their proper placement are also represented in a dedicated category called "purgatory." To enhance scalability, ease of focus, and usability, CLISERVO offers two configurations: the Big-Picture view, which encompasses all software components (client, server, DTOs, and purgatory), and the Server-focus view, which focuses solely on the server part.

The application of CLISERVO to industrial projects has yielded highly promising results. For example, in the WD project, the visualization exposed DTO-related violations, including the presence of unused entities and entities exclusively used by either the client or the server. These violations undermine the software architectural integrity by introducing unnecessary complexity and consuming additional memory and resources. Moreover, they hinder software comprehension and navigation, especially in large-scale projects like WD, which comprise numerous classes (6030 classes). On the other hand, when applied to the OJ project, which demonstrates a well-conceived and maintained architecture, CLISERVO revealed relatively few violations, primarily false positives stemming from design decisions to avoid code duplication and the inclusion of DTO classes specific to the core software on which OJ is built. This gives insight into the architecture of the software. Additionally, by employing the server-focus configuration, CLISERVO exposed illegal dependencies between layers within the server part of the EGF project, including seven direct links from service implementations to DTOs and five violations from DTOs to the services layer.

Through the application of CLISERVO, this research contributes to the understanding and improvement of client-server architecture in software systems. The visualization not only aids in detecting architectural violations but also facilitates the identification of unnecessary elements, thereby enhancing the maintainability and performance of client-server applications.

3.8 Threats to Validity

In this section, we discuss the threats to the validity of the experiment.

Internal Validity: To what extent we can draw a causal link between the treatment in the experiment and the response? The study exhibits robust internal validity as it employed a rigorous experimental design involving three different software systems from diverse domains, two different companies, and three project teams. The systems were real industry applications, and access was granted to their current maintainers, who possessed distinct levels of knowledge about the software systems. The study enabled architects to draw insightful conclusions and make modifications to the client-server architecture. Furthermore, it is noteworthy that the software systems are currently undergoing a remodularization phase in preparation for future migration. These factors contribute to the study internal validity by providing a controlled environment for evaluating the impact of the tool on architectural decision-making within real-world software systems.

External Validity: Are our results generalizable for practice modernization? The CLISERVO visualization validation was applied on three industrial projects since it was not possible to take open-source projects. Indeed, unfortunately, on public repositories like GitHub, it is not easy to introduce search criteria relative to architectural matters. In addition, once found, access to the developers or to the architects of these projects is almost impossible. Consequently, we focused on industrial partners. The variability of the application domains, the sizes, and the development teams encourages us to believe in the generalization of the CLISERVO visualization.

Construct Validity: Are we asking the right questions? CLISERVO enabled us to identify both cases where the separation between the client and server is clear respecting the used framework and others where it is not the case. The focus on the server part highlighted an expected layered architecture but also enabled the identification of architectural rule violations between these layers. This leads us to the conclusion that the interest of the visualization is justified.

Reliability: To what extent can the results be reproduced when the research is repeated under the same conditions? One potential reliability threat in this study is the potential for variations in the interpretation of the results by maintainers. To address this threat, efforts were made to provide clear guidelines to maintainers, ensuring a standardized understanding and consistent results. By establishing explicit instructions for interpreting the visualization, potential discrepancies in the findings were minimized, enhancing the reliability of the study. Additionally, the availability of the CLISERVO on GitHub and its use of an importer specifically designed for Java projects contribute to the reproducibility and consistency of the study results across different projects and maintainers.

3.9 Conclusion

Understanding source code and recovering its architecture is a challenging task, particularly for complex systems that have been developed over a long period of time or have undergone numerous changes and modifications. We propose a novel visualization called CLISERVO for recovering the architecture of client-server systems. CLISERVO help software maintainers detect architectural violations. It classifies client-server entities into different levels of dependencies, shared entities, or ambiguous entities. In addition to the two configurations: Big Picture and Server-focus, the visualization shows the system from two distinct viewpoints: General Overview and Violation. We validated our approach on three real-world industrial projects with their current maintainers. The validation with the maintainers shows that, in a couple of hours, the CLISERVO users were able to spot architectural violations and to qualify architecture design decisions. It shows also that CLISERVO users were able to conceptualize a coarse-grained quality model of the systems architecture (ranging from 3277 to 6030 classes), clearly identifying situations where future evolutions will be challenging. For future work, we plan to extend the viewpoints and apply the visualization to a larger set of projects.

CHAPTER 4

Understanding Class Name Regularity: A Simple Heuristic and Supportive Visualization

Contents

Good class names play a crucial role in software development as they serve as a means of communication and documentation within the codebase. This puts a strong emphasis on the understandability of source code. Class names constitute one of the first information maintainers have access to. As such, they are important to promote or hinder understandability of a project. Understanding the convention(s) followed by class names is essential to guide maintainers and code reviewers interpretation. Different developers may follow different conventions, or conventions may evolve over time.

To assist code reviewers and software maintainers in understanding the logic and regularity of class names, we present a new visualization, called *ClassName Distribution*. It brings together package and inheritance as structural perspectives on class names. ClassName Distribution allows one to spot naming irregularities in large hierarchies scattered over multiple packages. We first start by presenting the visualization (Section 4.2) and identifying recurrent patterns relative to concept references in class names (Section 4.2.4). We then illustrate the visualization on an important Pharo project continuously maintained by the community (Section 4.3).

We proceed with an illustration of the visualization on a large-scale Java project (Section 4.4). We apply the visualization on distinct versions and monitor class renamings over these versions (Section 4.5). Furthermore, to enhance familiarity with our visualization, we provide an overview of the tool and its interactive features (Section 4.6). The algorithm clarifying elements of the visualization creation is presented afterward (Section 4.7). Both qualitative and quantitative evaluations (Chapter 5) are presented in the next chapter. This work has been published in the *Journal of Object Technology (JOT)* [Agouf et al., 2022a].

4.1 Complexity of Class Name Understanding

The class name is the first piece of information concerning the classes to which the developers have access. A class name identifier is a sequence of "words" that are easily identifiable thanks to the use of naming conventions, such as the camel case or snake case style [Butler et al., 2011b]. For instance, considering the class name FloatingPointException, the word sequence is Floating + Point + Exception [Butler et al., 2011b, Liblit et al., 2006, Singer and Kirkham, 2008]. A class name should be as precise as possible to explain the class behavior while remaining concise, in the sense that it is briefly described, and consistent, in the sense that it is coherent with the system naming convention [Deissenboeck and Pizka, 2006]. Precision and conciseness can be in conflict so developers must make choices to determine the correct class names. In the following, we discuss what is a correct name, and in particular, what can influence how classes are named. Finally, we specify consistent naming or irregularities in class naming.

4.1.1 Illustrative Examples

Imagine the following situation: When reading a code editor project, a maintainer reads a class named NavigationBrowser. He knows that the system was developed using Model-View-Presenter. Now, by just reading the class name he cannot decide whether NavigationBrowser is a model, a view, or a presenter. He is forced to read the class definition to see that NavigationBrowser inherits from SpModelPresenter and to understand that this class belongs to the Presenter part of the triad. A consistent naming following the hierarchy convention such as NavigationBrowserPresenter conveys more precise information and does not force the maintainer to navigate through the class definition.

Another interesting example drawn from a Pharo project is the package ToolDependencyAnalyser-UI. This package defines the class named DANode, with 21 subclasses using the suffix Node. This class inherits from TreeNodePresenter. But in its package, all the subclasses of DANode are consistently named to convey

that they are nodes. In other packages, the names of subclasses of TreeNode-Presenter terminate with the suffix Presenter. It appears that only in the package Tool-DependencyAnalyser-UI, did developers introduce a new concept and it was more important for them to convey the idea that a class represents a Node than a Presenter.

Now in this example, the situation is a bit more complex because this exact same package defines the DAPackageTreePresenter. Therefore, the maintainer may wonder if this class should be renamed to Node or not. However, the class DA-PackageTreePresenter does not inherit from TreeNodePresenter, but belongs to the ComposablePresenter hierarchy.

Stepping back from this example, we see that (1) class names may miscommunicate their roles, (2) developers may introduce new naming conventions and that such conventions may be local to some packages only, and (3) maintainers need to be able to get an overview of the names used by the classes within a project but with a package view and taking hierarchies into account.

4.1.2 About Correct Class Names

In object-oriented languages, classes should have one responsibility [Wirfs-Brock and McKean, 2003]. A class name should concisely explain this responsibility. Consequently, a correct name is a name that enables the developer to understand at a glance the purpose of the class or the concept behind it.

In practice, there is not always one responsibility in the class. In addition, synonyms can be chosen. Consequently, there is not a single correct name per class, and finding one can be complex since several factors may influence the naming as we present hereafter.

4.1.3 Forces Influencing Class Naming

Class names are mainly influenced by three competing forces: packages within the project, naming conventions, and inheritance hierarchies.

Package. Packages, as other grouping entities, such as modules or tags, provide another abstraction level as they are not at the same conceptual level as classes. Packages often reflect several organizations: they are units of *code deployment* or units of code *ownership*. They can also encode team structure, architecture, and stratification [Abdeen et al., 2009, Lanza and Marinescu, 2006, Martin, 2000]. Such roles often impose different naming conventions or new vocabulary on class names. For example, it is not rare to see that inside one package classes inheriting from a superclass get a new suffix but only within the package. This is because the developer wanted to convey a new and different role for the classes.

In addition, in some object-oriented languages, such as Java, packages are namespaces: the name of a class is unique inside a package and two classes inside the same project may have the same name.

Inheritance. Mostly, inheritance corresponds to a concept refining. Subclasses refine a concept defined in the superclass. Consequently, it seems natural that inheritance influences class naming. In their study about the names of Java classes, Butler *et al.,* [Butler et al., 2011b] found that 70-80% of classes that extend a superclass different from Object include one or more words repeated from the superclass name. This is important since a developer can know at a glance to which main concept a class is related.

However, inheritance may have several semantics. When a class extends another class using *subtyping* the initial class name is often extended [LaLonde and Pugh, 1991]. On the contrary, if inheritance is used for mere code reuse the initial name is often fully dropped in the subclass. For example, in historical Smalltalk systems, OrderedCollection is a subclass of ArrayedCollection, which itself is a subclass of Collection (subtyping), while Link (element of LinkedList) is the subclass of Process (subclassing) [Goldberg, 1984].

Naming Convention. A good practice, both in industry and academia, is to use English to name classes. This is to ease the understanding of the code, by international or outsourced teams, or to enhance the spread of open-source projects. Since in English adjectives are put before the noun they qualify (*e.g.,* BigClass or SmallModel), this leads to the hypothesis that a particular role is given to the last noun, meaning the suffix of a class name (last word). In FloatingPointException the class suffix is Exception: this noun suffix stresses that the class is an exception. This hypothesis is supported by the analysis conducted by Butler *et al.,* showing the importance of the suffix in the identifier names of Java classes [Butler et al., 2011b]. Note that in other tongue languages as in French or in Spanish, it is not the case; adjectives are mostly put after the name. In this work, we focus on the code written in English and consequently adopting such a naming convention.

4.1.4 Limitations of the Various Forces in Presence

Inheritance. Often the class name structure evolves along an inheritance tree when important new concepts are introduced. In addition, because of the name length limit [Binkley et al., 2009], such new concepts may lead to the dropping of old names, use of abbreviations, or focus only on new aspects. The problem is that when a developer drops the superclass name from a subclass, he cuts the link to the superclass. Doing so he makes a class name more difficult to understand. To understand the class, another developer is forced to look for its superclass.

Naming Conventions. For various reasons, some conventions put the important noun as the prefix and not the suffix. this is for example the case in Pharo, for classes describing the architecture of every project which are named BaselineOfXXX. They are easily identified by their prefix. In Pharo, it is one of the few well-known exceptions concerning suffix dominance. However, we observed that such cases may often occur in Java (see Chapter 5).

Other Limitations. Other limitations also enter into the class naming.

- **Name length:** Class names are limited by the "reasonable" length of identifiers. This "reasonable" length varies according to programmers, but it introduces a limit on class names.

- **Local/Global perspective.** Naming regularities may significantly change when considered from a local or a global perspective. Looking at names within a single package is different from doing so across a full project.

4.1.5 Our Definition of Class Name Consistency

In this chapter, the following points define what we consider to be class name consistency:

- **Class name only.** We exclusively focus on class names and not class comments, method identifiers [Anquetil and Lethbridge, 1998] or method body vocabulary [Antoniol et al., 2007].

- **Following superclass pattern.** Class names are consistent when the classes of the same hierarchy follow the same *naming pattern*. By pattern here we consider that class names follow either the same prefix or suffix across their hierarchies, *e.g.,* Test* or suffix *Test for all the subclasses of the class AbstractTest. Another example is the subclass DropListView of the class View, which follows a consistent naming.

  When a class suddenly drops a suffix from its superclass, we consider this to be a class name inconsistency [Butler et al., 2011b]. For example, when DropList, a subclass of the class View, is not named DropListView, there is an inconsistent naming.

- **Possible local redefinition.** In addition to the simple pattern mentioned above, we take also into account the possible influences of packaging and inheritance as well as other conditions in some cases personalized by project maintainers.

For example inside a package, if *all* the subclasses of the class Shape are now prefixed using the word Arrow, it is not an inconsistency because we consider that developers have the right to introduce new vocabulary. Note that this local redefinition will be detected by our visualization but we will consider it as a false positive.

As elaborated above and in the next sections, the inconsistencies that our approach detects are only based on the words and their sequences of class names taking into account the inheritance hierarchies and package structure. It means that we do not consider typos: a programmer can name all their classes *Comand, but if he does it systematically we consider that the naming is consistent. However, any deviation from the superclass pattern will be reported as inconsistencies. We also propose a tool that helps the user quickly detect inconsistencies without having to look deep into their project.

4.1.6 Class Name Assessment

Given all the competing forces influencing class naming, it is doubtful that one could come up with one absolute naming convention even for a single project. However, there is a need to assist developers or maintainers in detecting irregularities in class names and naming convention violations.

When auditing code, reviewers are often forced to manually browse the class definition and figuratively climb the inheritance tree to understand the classes they are facing. Checking class names manually is difficult even for a mid-size project composed of several hundreds of classes, structured in multiple class hierarchies of different depths, and distributed over many packages. Just looking at the class name list, even on a per-package basis, might not reveal valuable clues about the conventions used and whether they are consistently followed.

We propose the following rules to help developers review class names inside their project:

- The main concept in a class name is expressed by either the prefix or the suffix. In the remainder of the chaper, we will use the term *spfix* to refer to either the prefix or the suffix.

- Inside a hierarchy, the spfix should be consistent, meaning that it should be unique.

- Since concepts may emerge inside a hierarchy, the preceding rules may be violated. Consequently, to ensure consistency inside a single package, each hierarchy should correspond to one concept and have a single spfix.

4.1.7 A Schematic Project

Before introducing concepts useful to the detection of class name inconsistency, let's consider the hypothetical project depicted in Figure 4.1. It is composed of two packages P1 and P2 and consists of 6 inheritance hierarchies: A, B, C, D, E, and F. Inheritance hierarchies begin right under the Object class otherwise we would always have exactly one inheritance hierarchy.

Inheritance hierarchy root classes are marked with a thick border. Each inheritance hierarchy is marked with a different color (A=yellow, B=green, C=red, D=blue, E=pink, F=purple) to differentiate them.

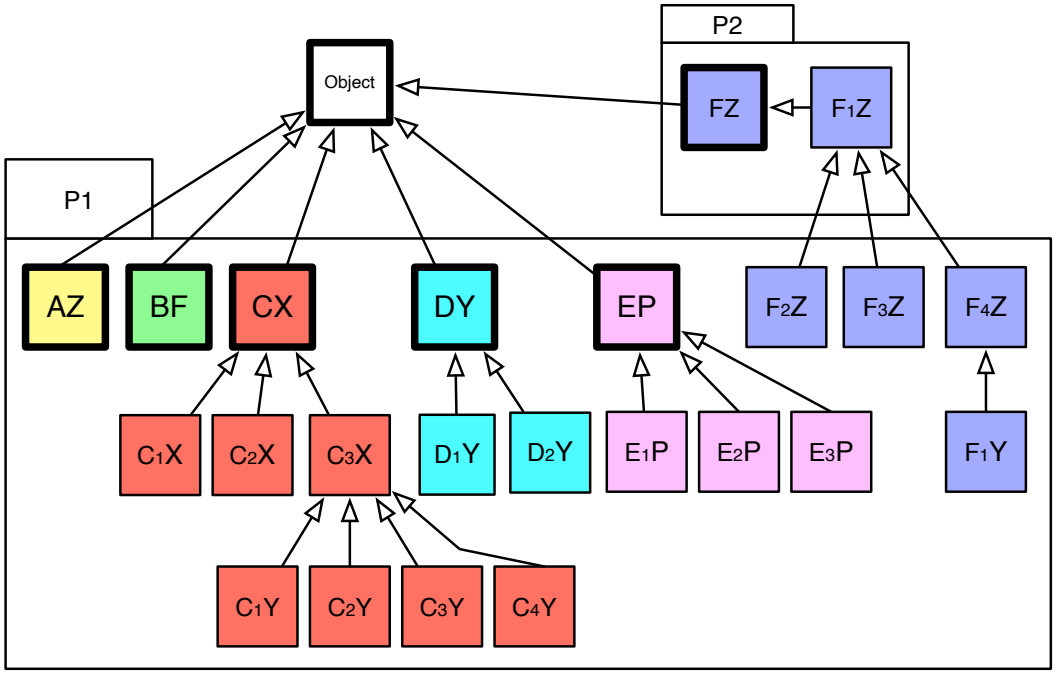

Figure 4.1: A schematic mini project composed of the A, B, C, D, E, and F hierarchies (thick borders denote hierarchy roots).

In this figure, class names follow several conventions: The first letter identifies the inheritance hierarchy (A, B, ...). Note that such a convention exists in real projects, but there is no guarantee that it would be as strictly followed as in our example. The last letter (X, Y, Z, P) represents a suffix. For example, the classes AZ and F_2Z have the same suffix as well as D_1Y and C_4Y. An optional number differentiates sibling classes using the same prefix and suffix.

4.1.8 Class Name Inconsistency Detection

To help developers to detect inconsistencies in class naming, we introduce some concepts and explain them using the schematic project of Figure 4.1.

Mono-class hierarchies. These are hierarchies consisting of only a root class (no subclasses). In our hypothetical example, this is the case for inheritance hierarchies A and B.

Mono-spfix hierarchies. These are hierarchies consisting of several classes that *all* have the same spfix. Inheritance hierarchies D (suffix Y) and E (suffix P) are examples of *mono-spfix* hierarchies. A mono-spfix hierarchy can share its spfix with some other inheritance hierarchies (some C classes have the suffix Y).

Multi-spfix hierarchies. Such hierarchies consist of classes with different words used as spfixes. Multi-spfix hierarchies are important in the sense that they do not follow a clear naming schema and thus may hide a new naming convention or be misnamed. Such hierarchies are exemplified by C (which uses the suffixes X and Y) and F (with the suffixes Z and Y).

The following section tackles the problem of class name irregularities by presenting a new visualization, named *ClassName Distribution*. It helps identify suspicious patterns and class name irregularities. It gives both a local view inside a package, while providing a global view, all along the hierarchies at the level of the project.

4.2 The ClassName Distribution Visualization (CnD)

The ClassName Distribution[1] is a package-centered visualization based on the distribution of the vocabulary used in a project taking an inheritance perspective. This vocabulary consists of the suffixes or prefixes of class names – last and first words respectively, from the original name (using any conventions, camel case, snake case, or others). Indeed if in Pharo, developers use a suffix convention, some projects, in particular in Java, use a prefix convention for some of their hierarchies (*e.g.,*TestReader instead of ReaderTest). The visualization is also interactive and navigable.

4.2.1 Visualization Constraints

Designing a new visualization should take into account several constraints:

- One important constraint of this work is the reproducibility of the visualization. We want maintainers to be able to implement this visualization with their graphical toolkit in a couple of days. Therefore, the layout of the visualization and graphical elements should be as simple as possible. This follows

[1] https://github.com/NourDjihan/ClassNameAnalyser

the design principles of Lanza visualizations such as system complexity and evolution matrix [Lanza, 2001, 2003, Lanza and Ducasse, 2003].

- The visualization should not overwhelm users with too much information (colors, shapes, positions). Its principles should be easy to understand while being able to scale up to large hierarchies or large projects. Our goal is that developers can take 15 minutes to comprehend it and start using it.

- The visualization should take into account screen limits: it should fit on a normal screen and avoid forcing users to navigate or scroll when possible. In addition, the numbers of colors on an average screen quality are limited. There are problems with new colors emerging due to the proximity of different colors. The visualizations should take such parameters into account.

- Furthermore, visualizations may want to exploit the Gestalt principles (such as connectedness, similarity [Peterson and Berryhill, 2013], and proximity) and pre-attentive processing [Healey et al., 1993].

  Researchers in psychology and vision discovered many visual properties that are pre-attentively processed, without actively thinking about them. They are detected immediately by the visual system: viewers do not need to focus their attention on a specific region in an image to determine whether elements with the given property are present or not. An example of a pre-attentive task is detecting a filled circle in a group of empty circles. Commonly used pre-attentive features include length, width, size, shape, filed, curvature, intensity, hue, orientation, motion, and depth of field [Healey et al., 1993, Treisman, 1985]. However, combining them can destroy their pre-attentive power (in a context of filled squares and empty circles, a filled circle is usually not detected pre-attentively).

We are now ready to describe first the layout of a ClassName Distribution (Section 4.2.2) then its color assignment (see Section 4.2.3).

4.2.2 ClassName Distribution Layout

The ClassName Distribution represents the distribution of the class name spfixes throughout the hierarchies of a project structured using packages. To this end, it uses three central visual elements: *class boxes* within *spfix boxes* within *package boxes*. Figure 4.2 represents the ClassName Distribution for our hypothetical project shown in Figure 4.1 and Figure 4.3 represents a real project (Calypso v.6.0, described later).

Class boxes. Class boxes, the smallest boxes, represent the classes of the packages under consideration. They can be seen as atomic "dots". Thicker borders identify inheritance hierarchy root classes. Except in special cases (see Section 4.2.3), there is one color by inheritance hierarchy. Here, the colors match the ones in Figure 4.1 (C=red, F=purple).

Spfix boxes. Spfix boxes, the intermediary boxes, represent class prefixes or suffixes in a given project. They group class boxes (for the considered package) whose name begins or ends with this spfix. The spfix boxes are colored according to the dominant inheritance hierarchy (in the number of classes) that they contain *across the project*. This ensures that a given spfix has the same color in all packages of the project. For example, see the "Query" spfix (blue) in the first and fourth packages of Figure 4.3.

On the visualization, spfix boxes are labeled with the prefix (P), suffix (S), or (P+S) if the same word is used inside the same package as the prefix and suffix. If over the whole visualization, only suffixes or prefixes are used, the letters between parentheses are omitted to not overload the visualization. The user can choose to use only suffix (which is the default mode), only prefix, or both, which is recommended for Java projects. In that latter case, an algorithm determines for each class if the prefix or the suffix should be taken into account (see Section 4.7).

Inheritance hierarchies (thus their colors) are ordered across the project from larger (more classes) to smaller (fewer classes). Spfix boxes, which are also colored, follow the same order that is dictated by their respective colors. This ensures that in different packages, the same spfix always appears in the same order. These consistent ordering and coloring schemes allow one to easily find an spfix in any package. For example, see the "Scope" spfix (magenta) in various packages of Figure 4.3.

Package boxes. Package boxes, the outermost boxes, represent packages and are labeled with the package name. Since package names may be long and for space reasons, we have chosen to possibly abbreviate them. Package boxes contain the spfix boxes of all their classes. A ClassName Distribution can display several packages to offer a general overview of the project (*e.g.,* Figure 4.3), or focus on a particular package. Package boxes are displayed in decreasing order of size (in number of classes).

4.2.3 Colors of the ClassName Distribution

The visualization assigns a color to each class: By default, the color of a class is that of its hierarchy, but there are exceptions. Focusing on the regularity of class name spfixes throughout inheritance trees, we distinguish three situations:

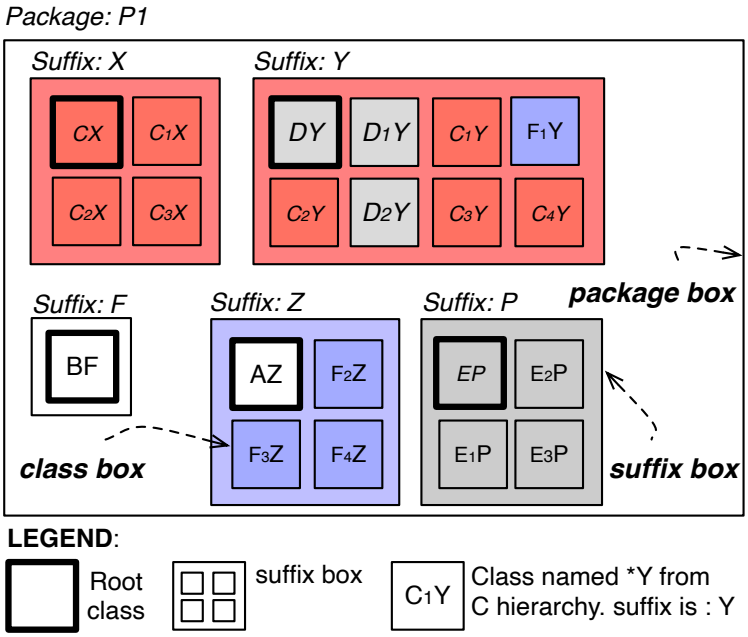

Figure 4.2: ClassName Distribution for package P1 of Figure 4.1: 1 package box, 5 suffix boxes, and 21 class boxes.

Mono-class hierarchies. Such hierarchies are composed of a single class. They are of limited interest: they do contain class name irregularities. They are "colored" *white* to reduce the number of used colors while still giving the information that the class is the only one in its hierarchy. A *mono-class* box may be placed in a dedicated spfix box if its spfix is shared by no other class in the same package. This is the case for class BF in Figure 4.1. A *mono-class* box may also share its spfix with other classes (from different inheritance hierarchies) and thus be placed in the same spfix box as these. For instance, class AZ shares the Z suffix with F_2Z, F_3Z, and F_4Z.

Mono-spfix hierarchies. Mono-spfix hierarchies perfectly adhere to the same naming schema. Since they have no irregularities, they are not noteworthy and are "colored" in *gray* to avoid attracting attention, and reduce the number of colors required for all hierarchies. See hierarchies D and E in Figure 4.1.

Multi-spfix hierarchies. By construction, their classes are grouped in separate spfix boxes. Such hierarchies are assigned a "real" color (not white, nor gray) and all their classes have the same color. In Figure 4.2, hierarchy C is colored in red and its classes are grouped in two distinct spfix boxes X and Y.

Such inheritance hierarchies are mainly discovered when several spfix boxes of the same color appear in a package. For example the several blue spfix boxes in the

first package of Figure 4.3.

(a) *multi-spfix* hierarchies are easily identifiable against *mono-class* and *mono-spfix* hierarchies (colored in white and gray respectively).

(b) The color of the class boxes identifies the *multi-spfix* hierarchy to which the corresponding class belongs.

For technical reasons such as screen quality and the inherent limitations of human eyes, only 24 colors are used and assigned to the 24 biggest multi-spfix hierarchies of the project. All the other multi-spfix hierarchies are in black.

4.2.4 Pattern Definitions

The ClassName Distribution gives an overview of the system hierarchies, their types (*e.g.,* mono-spfix hierarchy, multi-spfix hierarchy, ...), and the distribution of their spfixes across packages.

Based on this visualization, the user can identify inconsistencies and decide whether classes are poorly named or the naming was deliberate. To help users in their tasks, we have detected some *recurrent visual* patterns that can be also characterized by the definition of simple conditions. Such visual patterns may exhibit not only coherent naming situations but also unstructured or inconsistent naming.

- **Homogeneous spfix pattern.** Following our definition of class name consistency, a homogeneous pattern reflects a consistently named hierarchy. All the classes of the hierarchy share the same spfix, and this spfix is dominated by consistent hierarchies. It corresponds to a mono-spfix hierarchy which dominates its own spfix. Concretely, it is a set of gray classes inside a gray spfix box. As explained before ClassName Distribution marks as gray mono-spfix hierarchies, *i.e.,* the hierarchies where all classes have the same spfix. The spfix box is also gray, which means its dominant hierarchy is a mono-spfix one. It shows that a project is following a naming convention (See Figure 4.3).

- **Blob spfix pattern.** This expresses that inside a hierarchy, many classes of the same package use the same spfix. Concretely, it corresponds to a large spfix box where (almost) all classes are of the same color. A few classes of a different color may be allowed. The hierarchy is not homogeneous (otherwise it would be in gray), which means that somewhere in the hierarchy a couple of classes do not respect the largely adopted convention. It can be on purpose, or not. Ideally, there should be only one Blob of a given color per package.

These patterns are good in the sense that they indicate hierarchies are following a naming convention. However, the violation of a naming convention can be spotted by observing the visual patterns explained below:

- **Scattered vocabulary pattern.** In the same package, this pattern is represented by several spfix boxes of the same color containing several classes colored as the spfix boxes. An illustrative example is presented in Figure 4.3. It points to a multi-spfix hierarchy dominating several spfixes. This visual pattern highlights that classes of the same hierarchy do not share the same spfix inside the same package. From that perspective, it identifies an irregular naming convention. This pattern might include a *Blob* which means that a naming convention was followed but not consistently enough.

- **Intruder pattern.** This is represented by one or a couple of class boxes of a different color than the other classes within the same spfix box. It highlights a class violating a naming convention or being placed within the wrong hierarchy (possibly due to single inheritance). Indeed, the naming convention imposes either that all the classes of the hierarchy have the same spfix and thus are colored in gray or that new concepts inside a package have emerged, which is revealed by a *Blob* spfix pattern. An intruder is a class that adopts an spfix and thus a concept dominated by another hierarchy in the same package. Intruders may also point to a bad design choice for example using inheritance instead of delegation.

- **Snowflake pattern.** This is represented by several white classes within an spfix box. This visual pattern highlights a set of mono-class hierarchies sharing the same spfix. As an intruder, it may highlight a design issue. More specifically, when the spfix box is white and contains only mono-classes, it means that several classes share the same spfix and thus the same concept in the same package while being fully independent of an inheritance point of view; there may be a missed opportunity to group them in a new inheritance hierarchy.

- **Confetti pattern.** This visual pattern highlights classes of the same package but several different hierarchies that share the same spfix. As such, they may represent the same concept. This can be the result of two orthogonal decompositions of the domain forced into a single inheritance hierarchy. Graphically, the confetti pattern is easy to spot because it consists of several classes of different hierarchies (colors) within one spfix box.

These visual patterns do not always indicate a naming problem but they often refer to possibly suspicious cases.

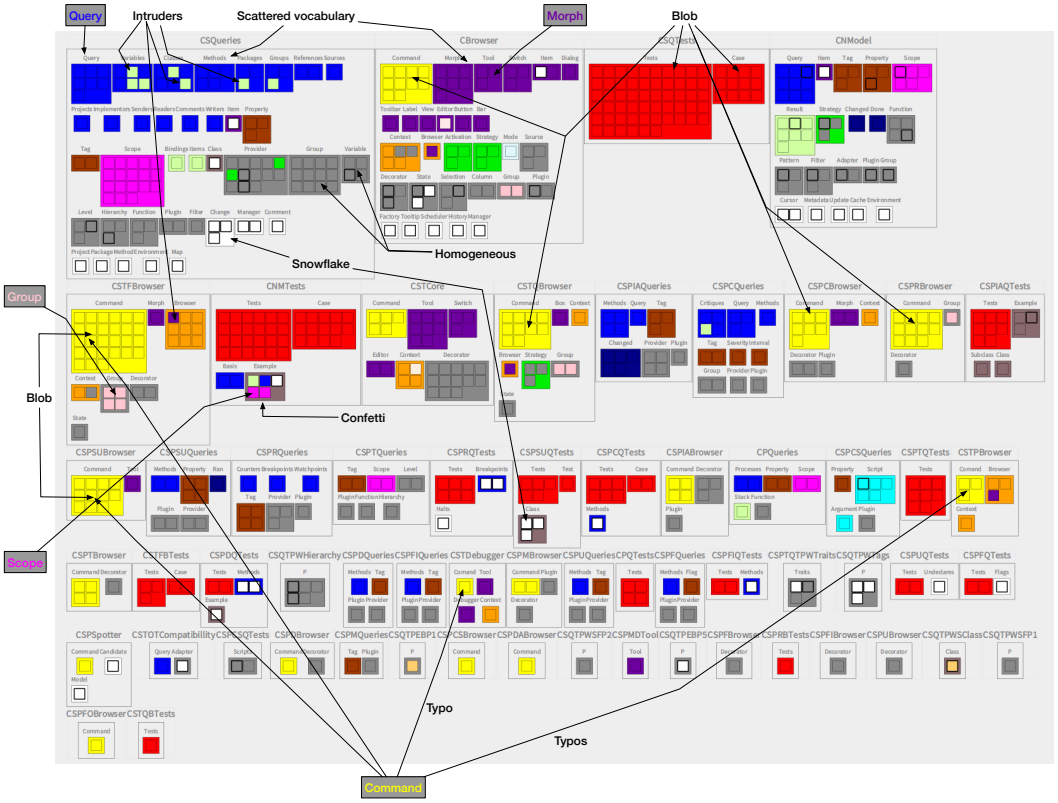

Figure 4.3: Visual patterns & hierarchies in ClassName Distribution of the Calypso project (v6) main packages.

4.3 An Example of a Pharo Project: Calypso

We present the first example of the ClassName Distribution visualization on a real project: Calypso v6. Calypso is an open-source project developed in Pharo. It implements a set of tools to browse source code. Since Pharo 7, it is the default IDE code browser suite. The latest version of Calypso (v9) consists of 758 classes organized in 59 packages and 6,076 methods. It was initially developed by a single engineer for two years, which motivates our choice of analyzing this project. Calypso is now maintained as an open-source project by a community as we will see in Section 4.5.

We take the V6 version because the visualization has been used by Calypso maintainers to rename classes over multiple versions as presented in the subsequent section. In addition, it shows that a program being developed by a single developer does not prohibit inconsistent naming.

In Pharo, there exists an implicit convention that the intent of a class is the suffix of its name. Consequently, in the following analysis, only suffixes are taken into account.

4.3.1 Calypso Hierarchies Analysis

This section analyzes Calypso hierarchies. Figure 4.3 exhibits the following points that we detail after:

- Some hierarchies are large (*i.e.,* lot of classes of the same color) and in contrast, few classes are mono-classes (*i.e.,* classes in white).
- Several hierarchies are consistently named (*i.e.,* gray classes).
- Many classes are spread over several packages, such as the blue, magenta, green, yellow, or red hierarchies.
- Many suffixes are spread over several packages such as Query, Command, or Tests showing a kind of naming convention consistency. In contrast, inside the same package, some suffixes are shared between hierarchies as in the first package of the first row where the Variables suffix is shared between the blue and light green hierarchies. This illustrates naming inconsistencies.

Large hierarchies. A color identifies a hierarchy with inconsistent naming (remember that consistent hierarchies are in gray). Figure 4.3 shows several of them. The tool supports interactions such as the highlighting of specific hierarchies and that such interactions help one to spot names (see Section 4.6). In addition, the high quality of the screen resolution supports the crisp reading of names.

- The **red** hierarchy contains the SUnit[2] test case subclasses. It has three different suffixes, namely Test, Tests, and Case. The Case suffix is due to the superclass being named TestCase in SUnit. Developers usually do not use this suffix but rather Test. The suffix Tests is less used and not promoted by the tutorials on SUnit or its conventions. In particular, the plural should not normally be used.
- The **blue** hierarchy is an important one, distributed over 13 packages (*i.e.,* outer boxes). It has many suffixes such as Query, Classes, Variables, Methods ... (1st package). This hierarchy defines the query object.
- The **yellow** hierarchy is a Command hierarchy, which defines classes in many packages and 17 suffix boxes. It is not homogeneous because of a typo: four classes have a Comand suffix (3rdrow first package and 4th row 10th package, from right; packages are annotated on the figure).

[2]SUnit is the test framework in Pharo

- The **magenta** hierarchy (Scope classes) is almost a mono-suffix hierarchy but for two classes in the Example suffix of CNMTests package (2nd line, 2nd package).
- The **purple** hierarchy (Morph classes[3]) is spread over 20 suffix boxes. Such classes are grouped within a limited number of packages (9). The purple hierarchy inherits from classes of this external package to define new graphical elements (widgets).
- The **pink** hierarchy (Group classes) presented in the CBrowser, CSTFBrowser, CSTQBrowser and CSPRBrowser packages is almost a mono-suffix hierarchy. The spfix box and the hierarchy do not have the same color (respectively gray and pink) because the Group suffix is shared by at least two hierarchies and dominated by a homogeneous one. Concerning the pink hierarchy, it is colored because the root class CmdMenuItem does not have the same suffix (Group). It is the only inconsistency of the hierarchy, but the root class belongs to another project.

Consistently named classes. Figure 4.3 shows multiple gray suffix boxes such as Provider, Group, and Decorator. Such gray suffix boxes tell that these hierarchies are consistently named. There are small hierarchies consisting of a couple of classes (such as Filter) but also large ones that spread over multiple packages *e.g.,* Provider, Decorator.

4.3.2 Calypso Visual Pattern Analysis

We illustrate the visual patterns with Calypso. Figure 4.3 is annotated with visual patterns to ease the reading.

Homogeneous spfix pattern. Concretely, this pattern occurs for example in the first package for suffixes Group, Variable, Level, Hierarchy, Function, Plugin, and Filter.

Blob spfix pattern. For example, such a case is spotted in the yellow Command suffix boxes distributed as *Blobs* in several packages. It is colored which means that the hierarchy is not consistent. The classes that are not in a *Blob* are often the ones with naming inconsistencies. Indeed, there is a misspelling: some classes have the suffix Comand with a single m. The visualization is interactive; a left click on the class highlights the hierarchy classes as shown in Figure 4.7 and puts the suspicious cases in a thicker white border. This is a way to detect misspellings.

[3]Morphic is a core package of the system that defines all the UI element logic.

Moreover, when there is more than one *Blob* of a single hierarchy per package this indicates a possible violation of a naming convention, which is the case for Tests and Case (*e.g.,* big red suffix boxes in the CSQTests and CNMTests packages). Classes of the Case suffix box should be renamed to have the Test suffix to follow the Pharo naming convention.

Intruder pattern. We see an example of this pattern in the first package, with light green classes inside blue suffixes. An intruder is a class that shares a suffix with classes from another hierarchy, which may indicate a bad design, the class being ill-named, or in the wrong hierarchy (possibly due to simple inheritance). It is also the case of the purple class inside an orange *Blob* in the CSTFBrowser package (first package of the second row).

Scattered vocabulary pattern. An example is the blue hierarchy in the first package, including the Query *Blob* which means a naming convention was followed but then split into several spfixes. Another example is the Morph hierarchy (purple) in the second package of the first row (CBrowser), which introduces new suffixes such as Tool and Switch. A closer look at classes of the Tool suffix box reveals a clear violation of the Morph scheme where the class ClyTextMorphTool needlessly introduces a new suffix by putting Morph in the middle of the name. The second violation of the Morph naming is the absence of the suffix Morph illustrated by the presence of View, Label, Button, and Dialog suffix boxes.

Snowflake pattern. An example is the spfix box named Change in the first package. The three classes ClyPackageChange, ClyClassChange and ClyMethodChange were found to have similar getters and setters (affectedPackage, affectedClass, affectedMethod respectively), and a handlesAnnouncement method. It may indicate that there was a missed opportunity to group these classes in a new inheritance hierarchy.

Confetti pattern. The colorful Example suffix (2nd row, 2nd package, last suffix) is one occurrence of this pattern. It shows many hierarchies of different types (multi-spfix and mono-classes), using the same spfix. This means that in the same package, the suffix is associated with many hierarchies.

4.4 An Example of a Java Project: Lucene

We now report on the analysis of the Lucene project. Lucene is an open-source library, in Java, for text indexation and search. We studied the 4,508 classes distributed over 287 packages of June 2021 version, without considering interface

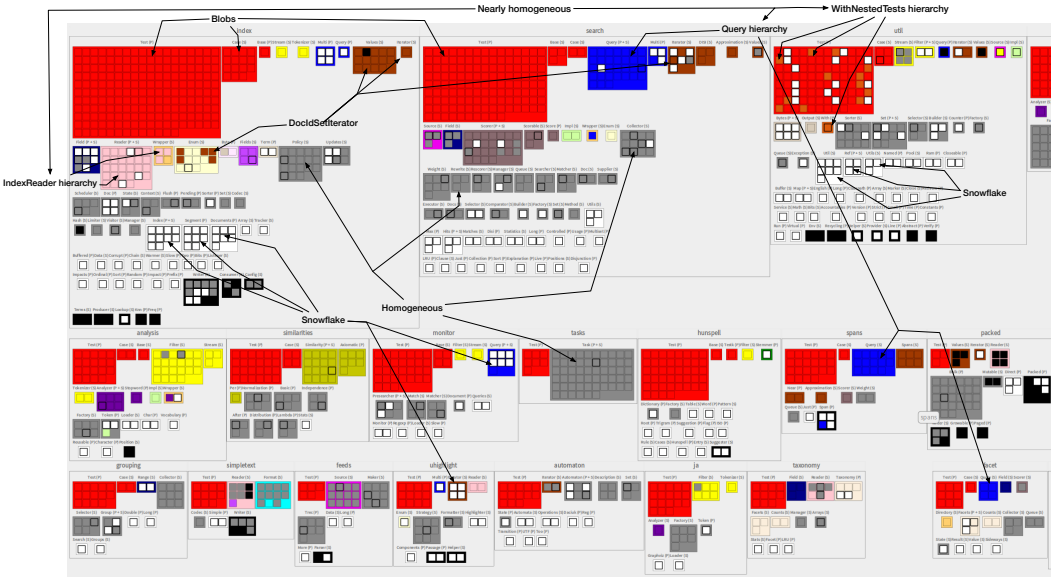

Figure 4.4: The ClassName Distribution of the Lucene project version of June 2021(Extract).

classes (184 interfaces). This project illustrates that our visualization scales for big projects and that it can be used for Java projects to identify class naming convention violations.

The ClassName Distribution shown in Figure 4.4 considers the distribution of both suffixes and prefixes. It corresponds only to an extract of the visualization. Indeed, although it is possible to zoom in with the tool, it is not on paper, so we display only a part of the project. A new screenshot of the whole project is available online[4]. Moreover, as there are 61 multi-spfix hierarchies in this version of Lucene, we colored only the 24 largest (totaling 2,458 classes, 54.52% of the project) due to distinguishable color number limitation and color aliasing.

The other 37 multi-suffix hierarchies are represented in black (175 classes, 3.88% of the project). Finally, there are 149 mono-spfix hierarchies colored in gray (908 classes, 20.14% of the project) and 967 mono-classes 21.45% of the project.

Homogeneous hierarchies. Several hierarchies are *homogeneous* which indicates that they follow a naming convention. Such patterns are exemplified by gray spfix boxes such as Policy (1st package), Collector, and Rewrite (2nd package). Classes of these spfix boxes follow the spfix naming convention and the location of the spfix, as they only use the suffix, however, other hierarchies such as Task (2nd

[4]https://github.com/NourDjihan/ClassNamesDistribution-PaperData/\blob/master/Lucene2021/Lucene2021.png

row 4th package, marked with P+S) respect the use of only one spfix throughout the inheritance tree, but do not fix its position. Some classes have the concept as a prefix, others as a suffix.

Blobs. The biggest *Blobs* in the Lucene project belong to the red hierarchy (whose root is the LuceneTestCase class). This hierarchy holds 1,504 classes, including 1,430 classes with Test as prefix (marked with the letter P above the spfix box). Hence, classes of the LuceneTestCase hierarchy indeed follow a particular naming pattern which is predominantly using the Test prefix. However, several classes of the hierarchy use both Base and Case as prefix and suffix, respectively. Others use Case only as suffix and Base only as suffix. Additional spfixes have very low occurrences such as Function, Abstract and Unicode.

Nearly homogeneous. Several hierarchies are close to being homogeneous, where the exception resides in one or a few classes using a second spfix.

Some of these hierarchies use only one suffix for almost all classes of the hierarchy but violate the spfix convention and introduce Wrapper as a second suffix. This may indicate a kind of decorator pattern within the hierarchy that the developer wants to make explicit.

For example, the IndexReader hierarchy (light pink in the 1st package) uses Reader as suffix except for three classes such as SlowMultiReaderWrapper which puts it in the middle and Wrapper as suffix.

A similar case is the Query hierarchy colored in blue, distributed over many packages (1st row 2nd package, 2nd row 3rd package,...). Most classes of this hierarchy use Query as a suffix which indicates a naming pattern. The Query spfix is marked as it is being used as both a prefix and a suffix (P+S) but this is due to the mono-classes using Query as the prefix. Inside the Query hierarchy, only three classes are considered as introducing irregularities: MultiTermQueryConstantScore-Wrapper, BlockScoreQueryWrapper and SpanMultiTermQueryWrapper. The two first classes belong to the second package. They are not in the same spfix box once again since the spfix detection is automatically performed by the tool.

In the util package, the Test spfix box contains in addition to the classes of the LuceneTestCase hierarchy in red, some mono-classes (represented in white) but also several classes in brown. The root class of the brown hierarchy WithNested-Tests appears in the With spfix box. It is the only class with With as prefix and Tests (plural) as suffix. The tool arbitrarily chooses the prefix. However, all the subclasses of this root class have Test (singular) as a prefix making this hierarchy nearly homogeneous.

Snowflake. Several *Snowflake* spfixes such as Segment, Index, Documents (1st row, 1st package), Util, Utils and Ref (3rd package) may indicate classes having the same behavior. This behavior is described by the name of spfix box. An interesting case is that of *Snowflake* classes belonging to a colored spfix box, which is the case for the Query spfix (2nd row 3rd package) including a query class. Moreover, the Iterator suffix box (3rd row 4th package) is dominated by the dark brown hierarchy (DocIdSetIterator) and contains mono-classes such as CustomSeparatorBreakIterator, WholeBreakIterator, LengthGoalBreakIterator, and SplittingBreakIterator. A look at these class names raises the question of whether there was a missed opportunity to group these classes in a new hierarchy or if they should belong to the dark brown hierarchy dominating the Iterator suffix.

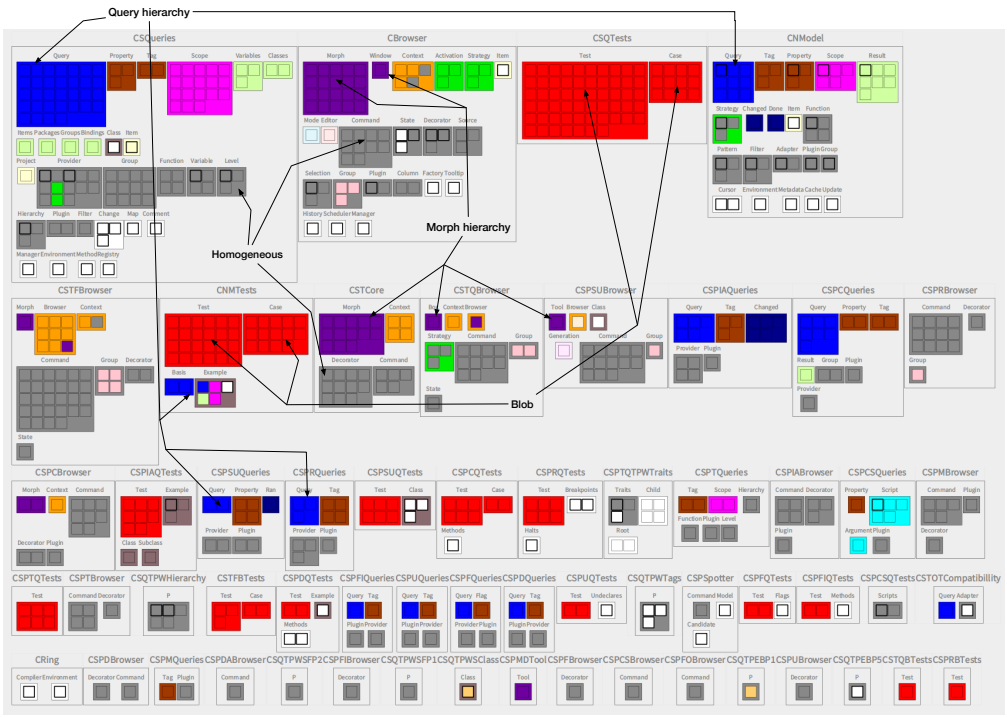

Figure 4.5: The ClassName Distribution of the Calypso project (v8) main packages.

About DocIdSetIterator. The dark brown DocIdSetIterator hierarchy presents an interesting case of the system architecture. This hierarchy is distributed over multiple packages. Its vocabulary consists of using the Enum and the Values suffixes (1st packages of the first row), the Iterator suffix (2nd and 3rd packages), Spans suffix (6th package 2nd row)... . A closer look at this hierarchy shows that classes using Spans as a suffix inherit from the Spans class which is a direct subclass of the root DocIdSetIterator. In contrast, classes using the Enum suffix also inherit from

PostingsEnum which is also a direct subclass of DocIdSetIterator Similarly, classes using the Values suffix inherit from DocValuesIterator, which is also a direct subclass of DocIdSetIterator. This leads us to think that the hierarchy is composed of several sub-hierarchies and needs to be decomposed.

4.5 Supporting Evolution

When the Calypso project was passed over to the community, maintainers found inconsistent class names that had been revealed using a preliminary version of our tool on Calypso v6 (July 2017) (Figure 4.3). To ensure consistency, many inconsistencies were corrected, leading to a new version of Calypso v8 (January 2020) presented in Section 4.5.1. This renaming work was huge because the maintainers did not know Calypso and had to understand the code in the presence of inconsistent class names. Moreover, this work was only tooled with a primitive version of our tool. Consequently, some irregularities remained. Finally, in Section 4.5.2, we show the latest version of Calypso v9 as of June 2021 after a final renaming effort. This last renaming phase was performed by the maintainers using the visualization proposed in this chapter as part of its evaluation as discussed in Chapter 5. This section shows that our approach supports also the understanding of class name evolution.

4.5.1 Calypso v8

When the community took over Calypso, some classes were renamed to improve the understandability of the project. The resulting project is shown in Figure 4.5. The new ClassName Distribution shows:

- More *Homogeneous suffixes* pattern (gray suffixes), for example the yellow Command suffix is now gray. This is positive and points to an improved naming quality.
- Less *Scattered Vocabulary*, in particular the blue hierarchy (1st package). Globally, this hierarchy now has only three classes outside the Query suffix, meaning that the hierarchy became more consistent, however three classes are left to be studied. Similarly, the purple hierarchy (3rd package) saw the number of different suffixes largely reduced to focus on the Morph suffix. This hierarchy now has only six classes without the Morph suffix. For example, the Morph hierarchy in package CBrowser went from eleven to two suffixes. A new suffix Window emerges which did not exist in v6, due to the definition of a new window class. Similarly in the v6, the ClyQuery hierarchy

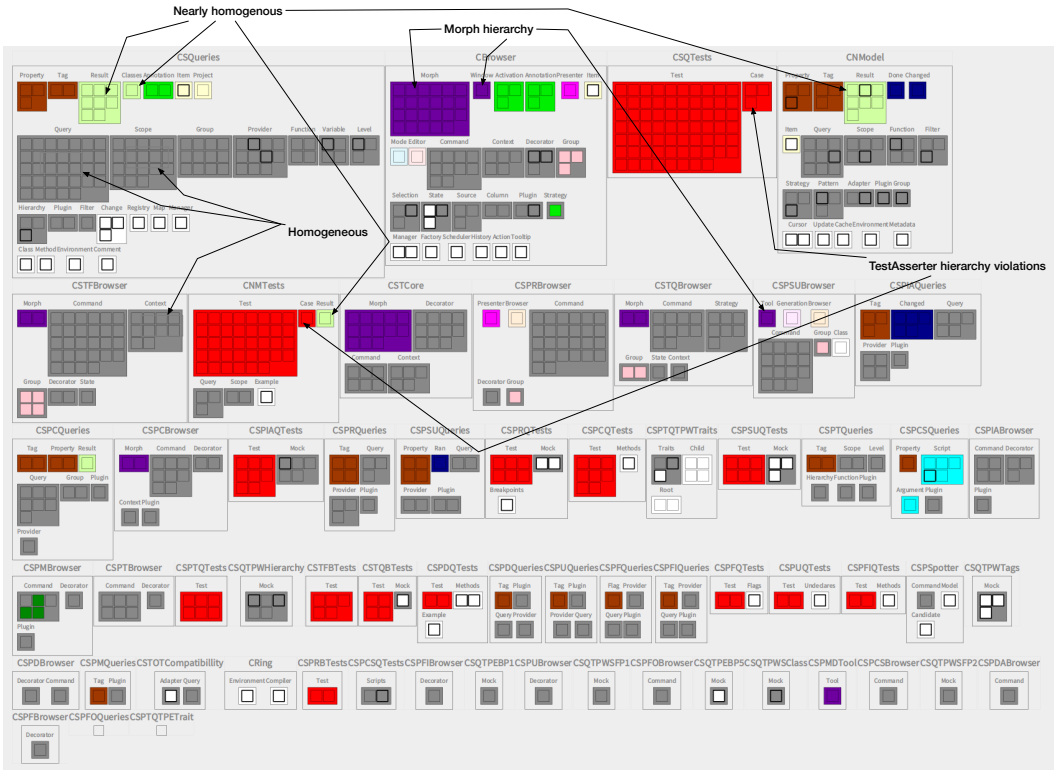

Figure 4.6: The ClassName Distribution of the Calypso project (v9) main packages.

exhibited several suffixes (32 classes in the Calypso-SystemQueries package with 14 suffixes). In v8, the 32 classes of the hierarchy share the same suffix Query.

- The *Blobs* of the red hierarchy TestAsserter went from using the Tests suffix, which is a violation of the testing naming convention, to using the Test suffix. However, some testing classes still violate this convention by using the Case suffix.

- Fewer *Intruders*, for example the light green *intruders* in the blue suffix Variables (1st package) kept their suffix, therefore the suffix Variables is now light green as blue classes were moved to the Query suffix.

4.5.2 Calypso v9

As explained above, while Calypso v8 underwent major class renaming, applying the ClassName Distribution visualization revealed some remaining class name irregularities. Figure 4.6 depicts the current version of Calypso v9 (2021) as a result of an evaluation with the maintainers (see Chapter 5). Indeed, a glimpse at the

visualization of v9 shows more gray classes and suffixes than in v8 which reveals the will of its maintainers to continue ensuring the consistency of their system. We notice:

- More *Homogeneous* hierarchies, in the first package both of Query and Scope classes: blue and magenta respectively in the previous visualization became fully consistent by using one suffix throughout each inheritance tree. Moreover, the orange hierarchy CmdToolContext also became *Homogeneous* by using only one suffix Context.
- Less *Scattered vocabulary*: the Morph hierarchy colored in purple went from having six classes using different suffixes other than Morph to three classes using the Window and Tool suffixes. The light green ClyQueryResult hierarchy grouped most of its classes under one suffix distributed over its packages, which is in fact the suffix Result used by the root class itself, this hierarchy becoming *nearly homogeneous* – one class away from being fully *homogeneous*, where the exception resides in the ClyExtensionLastSortedClasses class using the Classes suffix instead. It is an oversight, and this class will be renamed in v10alpha.
- Most of the Case *Blobs* of the red TestAsserter hierarchy are no longer present. However, four classes using this suffix are clearly still violating the test naming convention. This is one point the maintainers are considering correcting.
- Fewer *Intruders*: the two green *Intruders* in Provider suffix have disappeared, and are currently renamed to use the Annotation suffix instead of Provider. The purple *Intruder* in the Browser suffix has also disappeared because of two factors: the Morph hierarchy no longer uses this suffix and the orange hierarchy which dominated the Browser suffix is now homogeneous and uses the Context suffix instead.
- The *Confetti* case had vanished since both of ClyQuery and Scope hierarchies became *Homogeneous* and the Result hierarchy no longer makes use of the Example suffix.

The experiment in numbers. In total 91 classes were renamed between v8 and v9, over 10% of the system classes and as such, it is a large renaming effort. We cannot assess exactly the impact of the tool use but the maintainers reported that it helped them to be more systematic and get a better overview of the naming problems.

The visualization did not show any performance problems to render large projects. For example, a visualization is displayed in under 2 seconds for the Calypso project (around 700 classes).

4.6 The ClassName Distribution Tool

Figure 4.7: Tool interface with TestAsserter suffix classes highlighted: the tool surrounds with a white border irregularly named classes and darkens the rest of the visualisation.

The visualization presented in this chapter is proposed to the users via a tool. Besides selecting the projects and the packages, the tool supports the visualization configuration whether it should use prefix, suffix or both, the color palette to be used and it also proposes specific actions to highlight certain aspects of the visualization. After describing the basic architecture, we describe the features shown in Figure 4.7.

A language-independent metamodel. The tool is implemented on top of the Moose analysis platform developed in the Pharo language [Anquetil et al., 2020]. Therefore the tool is independent of the language used in the analyzed project. For the moment, it was used for Java and Smalltalk projects. Figure 4.7 depicts the ClassName Distribution tool user interface on a real project, Moose itself.

Importing models. The tool currently provides the possibility to visualize both Pharo and Java projects. It relies on a model of the project. The import of the models is performed differently for Java (1) and Pharo (2-5 in Figure 4.7) projects. Everything that follows is the same for projects of both programming languages.

Configuration. The tool builds three ClassName Distributions at once: with suffix, with prefix, and with both, following the algorithms explained in Section 4.7.2.

The user has the choice to render the desired visualization according to the selected radio button (6). By default, the suffix is selected.

By default the project root is Object, therefore root hierarchies are direct subclasses of the class Object. However, in some projects, many classes may inherit from the same subclass of Object (for instance, Widget for a GUI project). The user can define a new root class (7). If some classes do not inherit from the defined root, then their root remains Object. Nevertheless, the visualization will be based on two root classes – Object and the defined class. Changing the root class when a hierarchy contains multiple other sub-hierarchies is very helpful for a better overview of these sub-hierarchies and the distribution of their spfix in the project. A click on the "Visualize" button (8) renders the visualization (9).

Highlighting points of interest. To manipulate the visualization, a left-click on a class box highlights the whole hierarchy of the class, and since a hierarchy is represented by a color, then this highlights class boxes with the same color, red in Figure 4.7. Classes with potential violation spfixes are also highlighted but with white and thicker borders to attract the user's attention. Potential class name violations are: (i) in contrast to the other classes of the hierarchy, a class does not use the spfix of its root, or (ii) the spfix of the root is not used in the hierarchy and the classes have a different spfix than most classes of the hierarchy. To unhighlight the visualization, the user needs to left-click on a package box, a suffix box or a class box. Moreover, a right-click on a class box shows the class definition. A mouse-hover over a class box shows superclasses and subclasses of the class represented by the box and its root in bold (10).

Help and utils. Different kinds of help are available to the user (11-13). Last but not least, the list of visual patterns explained in Section 4.2.4 is found at the bottom left of the tool (14) with their explanation (15). These patterns help in guiding users to detect inconsistencies. When selecting a pattern, spfix boxes following that particular pattern are highlighted for the user to check. Finally, the user can export the visualization data such as the number of classes, packages, mono-classes, and mono-spfix hierarchies . . . as a CSV file using the export to "CSV" button (16).

The tool works for both Pharo and Java projects. When it comes to the performance of the visualization it purely depends on the size of the project. It is almost instantaneous for almost all the studied projects. Evidently, if the project contains more classes the tool takes more time to process the information before rendering. This also applies to the identification of the patterns: the bigger the project is the more time the tool needs to compute the selected pattern and highlight it visually. For illustration purposes, the rendering of the Lucene project (4,508) takes only a few seconds.

4.7 Visualization Algorithm Description

In this section, we describe the way the visualization is built. In particular, we describe in pseudo-code the different algorithms.

The approach follows the process below to build the visualization:

1. Cleaning and tokenizing the class names,
2. Identifying the spfix,
3. Identifying the color of the class, and
4. Ordering the packages and spfix.

In the following subsections, we explain each step of the process. The following section presents the tool as the users use it.

4.7.1 Cleaning and Tokenizing the Class Names

This step removes any digit or special character from the class name. The name is then split into a list of class name words according to the camel case convention. In a future version, we want to consider a list of exceptions, specified by the domain expert that will be taken into account to avoid false positives. Thus, it would, for example, be possible to make the distinction between 2D and 3D words and between Model and ListModel.

4.7.2 Identifying the spfix

In the case where only the suffixes (respectively the prefixes) are taken into account, this step is reduced to a simple activity, associating to each class the last (respectively the first) word composing its name as its spfix.

In the case where the project mixes prefixes and suffixes, we automated the detection of the spfix for each class to avoid the user manually specifying it.

First, we compute the number of occurrences of the suffix of the studied class in the whole hierarchy (line 2) and the same for the prefix (line 3). Inside a hierarchy, in case the number of suffix occurrences is equal to the one of prefixes (Line 4), the class is attributed to the spfix box in which it has more siblings. By siblings, we mean classes of the same package which belong to the same hierarchy and use the same spfix. Line 5 returns the spfix in which the associated box contains more siblings of `class` than the other. Line 7 returns the spfix with the highest occurrences in the `hierarchy`. If the occurrences are equal then the choice is arbitrary.

```
1 function chooseSPFix(hierarchy,class,suffix,prefix)
2   occurOfS = occurrences of suffix in hierarchy
3   occurOfP = occurrences of prefix in hierarchy
4   if occurOfS == occurOfP then
5     return class.maxSiblings(suffix,prefix).
6   else
7     return hierarchy.maxOccurrences(suffix,prefix)
8   end
9 end
```

Listing 4.1: Choosing the representing SP-fix

4.7.3 Color Assignment

Color assignment is decomposed into two parts: first the identification of the color per hierarchy and second the identification of the color per spfix box.

4.7.3.1 Identifying the Colors per Hierarchy.

As explained in Section 4.2.3, all classes belonging to the same hierarchy have the same color. In contrast, except for the trait classes and classes with no hierarchies that are colored in white, and mono-spfix hierarchies using only one spfix in gray, each multi-spfix hierarchy that uses more than one spfix is assigned to a different color (*e.g.,* red, green, blue, ...). Hence a color represents a hierarchy in the visualization.

Technically, each hierarchy is represented as an object whose attributes are its root class, the collection of the subclasses, the type of the hierarchy (*e.g.,* mono-class, or multi-spfix) and the color. Considering that we know for each class its spfix (as computed by algorithm 4.1) we attribute to each hierarchy object a type (trait, mono-class, mono-spfix, multi-spfix) (Line 3, Algo 4.3). Then, a color is assigned to the hierarchies (Line 4). For the mono-class and the mono-spfix hierarchies, classes are respectively colored in white and gray. Finally, we have to assign color to the multi-spfix hierarchies. We have selected 24 main recognizable colors for the palette of the visualization – it is possible to add more colors but then it becomes hard for the human eye to distinguish between hierarchy colors. Consequently, we sort the multi-spfix hierarchies according to the number of classes they contain. The first 24 largest multi-spfix hierarchies are assigned a color from the palette. Starting from the 25th multi-spfix hierarchy, complete hierarchies are colored in black.

```
1 procedure coloring(hierarchies)
2   for i = 1 to hierarchies size do:
```

```
3   attributeHierarchyType(hierarchy[i])
4   assignColorTo(hierarchy[i])
5   end
6 end
```

Listing 4.2: Attributing colors to hierarchies function

```
1 procedure attributeHierarchyType(hierarchy)
2 if hierarchy.subclasses size == 1 then
3   if hierarchy.subclasses[1] isTrait() then
4     hierarchy.type = traitType
5   else
6     hierarchyType = monoClassType
7   end
8 else
9   if all hierarchy.subclasses and hierarchy.root have
     the same spfix then
10    hierarchyType = monoSPFixType
11  else
12    hierarchyType = multiSPFixType
13  end
14 end
```

Listing 4.3: Attributing a type to a hierarchy

As detailed in algorithm 4.3, the type of the hierarchy depends on both the size and spfixes of its classes.

In case the collection of subclasses of the hierarchy has only one element (Line 2, algorithm 4.3), meaning that there is only one class in the hierarchy, the class itself is the hierarchy root. We check whether the class is a trait class (Line 3), in which case we attribute the trait type to the hierarchy type property (Line 4). If not, then the class is considered a mono-class therefore the hierarchy type is attributed the mono-class type value (Line 6). To distinguish between traits and mono-classes in the visualization, mono-classes have thicker borders.

In case the collection of subclasses has more than one element (Line 7), we first check if all the classes in the hierarchy have the same spfix or not. If all classes of the hierarchy including the root class have the same spfix then the hierarchy is attributed the mono-spfix type (Line 9). In contrast, if one of the classes including the root class of the hierarchy has a different spfix then the type of the hierarchy is attributed the multi-spfix type value (Line 11).

4.7.3.2 Identification of Colors per SP-fix box

The color of an spfix box depends on the biggest hierarchy using this spfix. In other words, the color of a specific spfix box follows the color of the hierarchy that uses it the most in the whole project. The size of the hierarchy does not matter, however, the number of classes using the spfix in each hierarchy does. For example, if we have two hierarchies, *H1* with 50 subclasses and *H2* with 30 subclasses, the two hierarchies use the same spfix with the occurrences of 10 and 15 classes respectively. The shared spfix box follows the color of the *H2* hierarchy since it has more classes using it (15 > 10). In this case, we say *H2 dominates* the spfix or the spfix *is dominated by* the *H2* hierarchy.

4.7.4 Ordering Packages and SPFixes

We order packages by the number of classes they contain. Then for each package, we create its package box and its name. For practical and display reasons, the name of the package may be shortened using a contracting algorithm [5] that keeps the starting letters of the package name in upper case and appends them to the last word of the package name to easily identify the package. Inside the package boxes, spfix boxes are always ordered in the same way. Thus for example, if an spfix *is dominated by* a hierarchy colored in red, it is the first to be rendered in the package– the absence of red at the very beginning of the package box means the absence of the red hierarchy in the package. This makes it easy for users to detect hierarchies and memorize the information of the visualization after a few interactions with the tool. Consequently, for this purpose, spfix boxes are ordered by color and by the number of classes in case they share the same spfix. Each box corresponding to an spfix or a class is created.

4.8 Conclusion

Understanding whether classes are consistently named within a project is important for developers. We presented one simple visualization that helps maintainers or developers to understand the regularities and irregularities of class names in hierarchies in the context of their packages. The CLASSNAME DISTRIBUTION aids maintainers and developers in comprehending the patterns and deviations present in class name hierarchies within their packages. We applied the visualization on two projects – one written in Pharo and the other in Java. We also showed that the visualization supports the evolution of projects: it helped the evolution of a large project over several years. In the upcoming chapter, we get into the details of the

[5] https://github.com/NourDjihan/NameAbbreviator

evaluations of the visualization on a larger set of projects of distinct domains under contrasting setups.

CHAPTER 5

Qualitative & Quantitative Evaluations of the ClassName Distribution Visualization

Contents

In the previous chapter, we introduced the CLASSNAME DISTRIBUTION visualization tool, which facilitates the identification of incorrect class names in Pharo and Java projects. So far, we have exclusively applied this visualization to two projects in each programming language: Calypso 4.3 and Lucene 4.4, encompassing various versions of the Calypso project 4.5. In this chapter, we extend the application of the visualization to a broader range of projects. Specifically, we have carefully chosen six significant projects from the Pharo community, and additionally, we have selected 50 open-source Java projects from GitHub. Through this expanded scope, we aim to provide a more comprehensive evaluation and analysis. Both qualitative 5.1 and quantitative 5.2 evaluation protocols and results are explained in the following sections.

5.1 Qualitative Evaluation

In this section, we explain the qualitative evaluation which aims to delve in depth into the quality of the class names of the selected projects. To evaluate our visualization, we used two different setups:

- **Domain Expert / Visualization Learners.** The idea of this first setup is to evaluate how experts of the code/domain who are also learners of the visual-

ization use the tool to identify inconsistencies in the class naming hierarchy. We presented the tool to Calypso as well as Roassal and Stargate experts.

- **Non-Domain Expert / Visualization Experts.** In this setup, we evaluate if non-experts of the code/domain but experts of the visualization can identify inconsistencies in class naming hierarchies that are then validated by experts. Non-experts also used the tool on Spec and Morphic projects.

Due to the size of the community and the proximity to experts, we chose only projects written in Pharo. In this section, we explain both protocols and present the feedback from participants.

5.1.1 Protocol for Domain Expert / Visualization Learners

Protocol. For this setup, we first prepared a 10-minutes Powerpoint presentation of the tool which includes (i) a summary of the approach principles (described in Section 4.2) and (ii) instructions on how to use the tool (described in Section 4.6). The presentation serves as a support guide for the tool.

For each project, we asked its practitioners (1 to 3 per project)[1] to do the experiment separately, to take notes of the changes each one would make, to record their screen, and to freely express their thoughts aloud during the whole experiment knowing that we will analyze their videos and that they will stay private. After receiving the screen records of each project, we collected the changes proposed by each participant. Collected data are thus twofold: first, a video showing the practitioner using our tool, and second a list of changes to correct inconsistencies. Depending on the cases, the changes may have been sent separately by email or we identified them in analyzing the video. Due to the nature of the collected data as well as the purpose of the evaluation, (*i.e.,* showing the ability of domain experts/visualization learners that our visualization can help them detect class name inconsistencies), it was not necessary to clean it. Then we set up one meeting per project gathering all the experts participating in the experiment, never more than two weeks after the experiment. To discuss the findings and to ask them if they can agree on the changes to make. Meeting all the experts of the project enabled us to discuss changes identified by only one expert and see if collectively they accept them or not. As explained later in the experiment, this meeting was also the opportunity for us to understand why they refused some changes. According to the final list of renamings, we made pull requests in each project GitHub repository and checked if the changes were integrated into the projects.

[1]Each practitioner was selected according to his expertise in the project and his availability during the experiment. Each of them has at least 10 years of experience in Pharo and more than 5 years of experience in the project.

The time spent by participants using the tool independently varied from 20 minutes to 30 minutes.

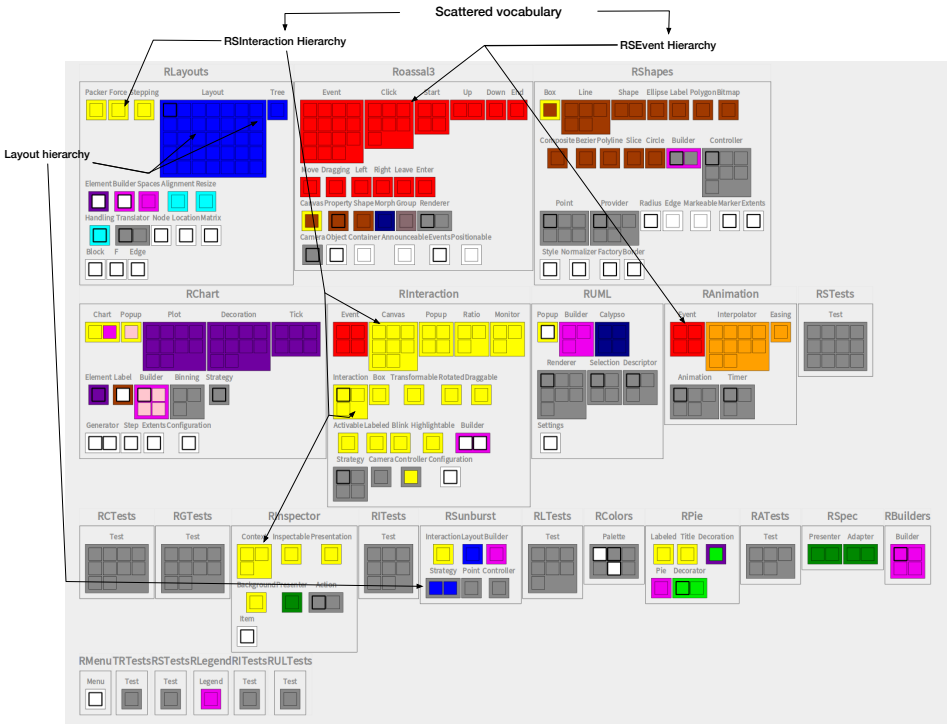

Figure 5.1: The ClassName Distribution of the Roassal-3 project.

Choice of the projects. We chose four projects from the Pharo community with the following criteria: (i) access to the developer or maintainers, (ii) diversity of the projects in terms of domain and size, and (iii) different development teams. This led to the choice of Calypso v8, Roassal-3, Stargate, and Willow. These projects consist of 150 to over 700 classes packaged in two to 57 packages. They are all in production and are respectively developed in France, Chile, and Argentina. Since Stargate and Willow are being developed by the same team and the validation was performed by the same expert, we describe the experiments of these two projects together.

5.1.1.1 Calypso v8 Experience Feedback

As discussed in Sections 4.3 and 4.5, Calypso underwent major changes in class names from v6 to v8. The experts were interested to see if there remain inconsistencies in the naming conventions. We asked three of them to use the tool and do the experiment on Calypso v8. In its v8 version, Calypso contains 57 packages and 716 classes.

Proposed renaming. Some test classes (red hierarchy) were still using the Case suffix, so they decided to rename them to remove Case but missed some as shown in Figure 4.6. The ClyQueryResult (light green) had several other suffixes which were changed to Result in the v9, as well as some classes of the ClassAnnotation hierarchy which eventually used the Annotation suffix (green).

They also intended to use the visualization to drive another pass such as strengthening further the purple hierarchy (now mostly Morph suffix but still with a Window and Tool suffixes). Thus, the goal is not to have all classes in gray but to ensure the correctness and consistency of class hierarchies. Remember that the gray color is an indicator of class names following the vocabulary pattern of their hierarchies, so such a case is considered consistent. In contrast, the use of another color means by definition that there is at least one inconsistency. However, this violation can be considered a false positive by the expert. For example to express the presence of several concepts in the same hierarchy (See Section 4.1.6).

Lessons drawn. There was no consensus on the identification of the classes to rename even if the majority of classes to rename were identified by at least two experts. However, (1) during the meeting, the experts agreed to rename almost all the identified classes, and (2) they systematically proposed the same name in case of renaming. Indeed, participants had the same logic when proposing new class names, following the suffix vocabulary used in the class hierarchy.

It was also interesting to see during the experiment that some experts identified not only inconsistencies in class naming but also errors in design. For example, currently the Tag and Property hierarchies are mixed. One expert proposed to rename all properties to tags whereas another considered that it should not be an inheritance, but a composition between Tag and Property. Such errors in design are more difficult to repair. We did not anticipate them, but we are pleased if the tool can also help in that.

5.1.1.2 Roassal-3 Experience Feedback

Roassal is an open-source visualization engine developed in Pharo. It forms part of the Moose project to script interactive visualizations. Roassal focuses on physically shaping digital data for further analysis [Araya et al., 2013]. The Roassal project consists of 326 classes organized in 24 packages. Figure 5.1 presents its visualization.

In this experiment, the three practitioners changed the root class from Object to RSObject. Indeed, almost all classes of the Roassal project inherit from RSObject, which plays the role of the root class for the whole project. Hierarchies in this project are built from RSObject and not directly from Object. Consequently, naming conventions are adopted from there.

Proposed renamings. There were a total of 39 renamings. Looking at the screen records from this experience, participants were all intuitively interested in classes of the RSInteraction hierarchy (in yellow in Figure 5.1) which had scattered vocabulary and was proposed to be renamed eventually to use only one suffix Interaction. This hierarchy consists of 45 subclasses, including five that already had the Interaction suffix, 27 classes that were renamed to use this suffix, and 13 classes that remained the same – without the Interaction suffix (see below for an explanation).

Another hierarchy that had very scattered vocabulary was the RSEvent one (in red in Figure 5.1): neither mouse nor key events use the Event suffix contrary to the other classes of the same hierarchy. These classes have not been renamed but only moved to a new Roassal3-Events package.

In the Layout hierarchy (in blue in Figure 5.1), three classes make the hierarchy inconsistent: RSAbstractCompactTree (in the first package of the first row), RSSunburstExtentStrategy and RSSunburstConstantWidthStrategy (fifth package of the 3rd row) have been renamed to adopt Layout, the suffix of the root class of the hierarchy. Here, it is not an addition of the suffix or a change in the order of the words composing the name of the classes that have been performed, but a change to use the suffix of the root class. Conceptually, the classes were strategies and become layouts, illustrating a real issue in their naming.

Furthermore, the hierarchy root class RSAbstractChartElement (in purple) was not only badly named and should use the suffix Plot instead, but needed to be decomposed since it contained a sub-hierarchy using the suffix Tick. Tick and Plot are two different concepts and need to be in two separate hierarchies. RSAbstractChartElement has been indeed renamed to RSAbstractChartPlot and its decomposition has been discussed and taken under consideration for future versions of the software.

From an architectural point of view, they have also moved the mono-classes in the first RLayout package to a new package called Roassal3-Layout-Utils, because these classes are not used alone but were created to serve other layout classes. In addition, the RSAbstractTick class should not inherit from RSAbstractChartElement.

Lessons drawn. The experiment was also the occasion to see that obsolete classes of another version of Roassal were still present in the code. The tool helps the developers to identify these errors based on class name inconsistencies but these errors can only be identified by experts of the projects. Indeed, our visualization is not focusing on the identification of obsolete classes and without prior knowledge, it is uncertain that a non-expert would identify them.

The brown hierarchy in the third package of the first row follows the scattered pattern. All the classes inherit from the RSObjectWithProperty class that plays a bit the role of a root class, (*i.e.,* classes of different concepts inherit from RSObjectWithProperty). However, for the moment, our tool enables the user to declare only one root class (besides from Object). It is part of our future work to enable the

user to declare several of them. Nevertheless, in that case, some experts were not sure whether the brown classes should inherit from RSObjectWithProperty or if these properties should be added through stateful traits Tesone et al. [2020].

As mentioned before, some renamings of the RSInteraction or the RSEvent hierarchies were not finally adopted by project maintainers. The reasons were different according to the cases. First, there is a lot of documentation for some of these classes. Consequently, renaming these classes would have a consequence on the documentation, which is not directly taken into account by the refactoring tool, and would have required more work to keep the documentation up to date. Second, the experts of the project wanted to keep the class names simple and short. We could not confirm if it is really simpler for a non-expert of the library when the suffix representing the concept embedded in the class is omitted. The experts were more familiar with the old names; they were reluctant to adopt some changes. Finally, they did not want to take the risk of changing these class names when many other projects depend on them, even if Pharo supports class deprecation.

An important point reported by the experts was that our tool allowed them to discuss their software, assessing some of their design decisions. This triggered points such as the use of old classes that they were not aware of anymore. They liked the idea to get an overview of the class names from a packaging point of view.

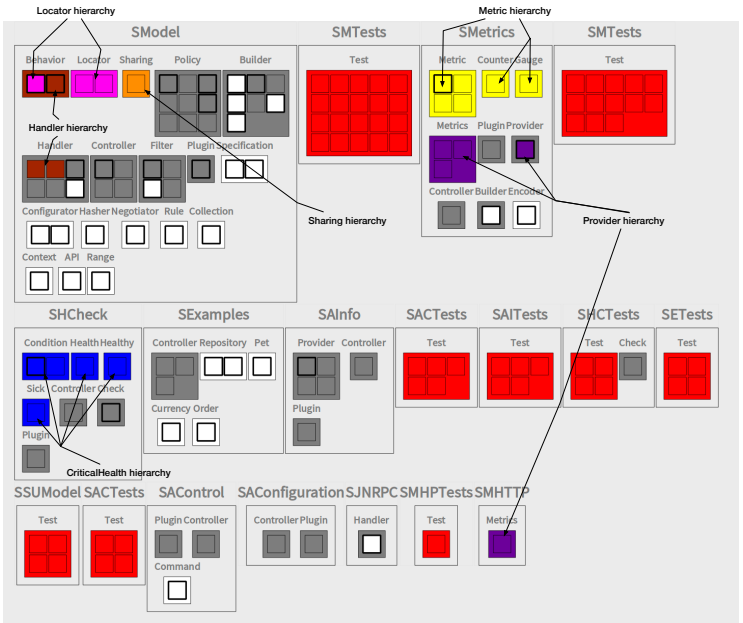

Figure 5.2: The ClassName Distribution of the Stargate project.

5.1.1.3 Stargate and Willow

Stargate is a library supporting the creation of HTTP-based RESTful APIs. It is composed of 18 packages and 151 classes. Willow provides a simple interface to develop web applications, no matter the chosen front-end framework. It consists of 234 classes and two packages. These two projects are developed by Buenos Aires Smalltalk under the MIT license. One expert accepted to use our tool on these two projects.

Proposed renamings in Stargate. There was a total of 12 renamings for Stargate and 12 for Willow. Globally, classes in Stargate were initially pretty well-named as shown in Figure 5.2. Indeed, there are seven multi-spfix hierarchies. Among them, there is the Test hierarchy (in red), which is fully consistent but as for the other Pharo projects appears as a multi-spfix hierarchy since the root class suffix is Asserter.

The Sharing hierarchy (in orange) has only one class that inherits from a class with another suffix. However, this root class exists outside the project. Consequently, no renaming has been proposed here.

The Provider hierarchy (in purple) has as root class MetricProvider, which has been renamed to MetricsProvider, and four subclasses had Metrics as suffix. These four classes have been renamed to add Provider as suffix. For example MemoryMetrics became MemoryMetricsProvider. In parallel, the associated test classes have been renamed: MemoryMetricsTest became MemoryMetricsProviderTest. Consequently, the Provider hierarchy is consistent, after these renamings.

The two classes of the first package with the suffix Behavior (ResourceLocatorBehavior and RESTfulRequestHandlerBehavior) have been renamed respectively to AbstractResourceLocator and AbstractRESTfulRequestHandler. Consequently, the two hierarchies Locator and Handler became consistent.

Finally, CriticalHealth (in blue) has been renamed to Critical. This renaming does not make the hierarchy consistent in terms of the used suffix. However, it is deliberate from the expert of the domain to keep the names as such since the first word is the most relevant.

When investigating why the yellow Metric hierarchy has not been touched, the maintainer explained that classes Gauge and Counter had exact proper names in the modeling context. However, the other classes that use the Metric suffix (LabeledMetric and TimestampedMetric) either add metadata over the others or act as a composition of other metrics (CompositeMetric), so in that sense are generic and hence justify the Metric suffix.

Proposed renamings in Willow. In the Willow project, the GRObject class serves as a root class for several sub-hierarchies. It ensures consistent initialization be-

havior on all platforms and provides error methods that signal an instance of WAPlatformError. It has been added as a root class for the project.

Three hierarchies used the Behavior suffix but only a few classes of these hierarchies share this suffix making the hierarchies inconsistent.

One of them inherits from GRObject and has three sub-classes. The maintainer of Willow chose to delete in the name of three of them the Behavior suffix. For instance, SingleSelectionWebViewBehavior became SingleSelectionWebView. However, for the WebInteractionInterpreterBehavior class, he deleted the Behavior suffix and added the Abstract prefix to identify this class as an abstract class. In addition, the test class associated to SingleSelectionWebViewBehavior has also been renamed to SingleSelectionWebViewTest.

WebTableColumnRendererBehavior is the root class of another hierarchy containing a unique subclass WebTableColumnRenderer. Once again the suffix Behavior was deleted from the root class and the prefix Abstract has been added. The hierarchy of this class is now fully consistent.

The third hierarchy is the one of the EventInterpreterDispatcherBehavior root class. None of its subclasses use this suffix. Consequently, the expert decided to remove the Behavior suffix of this root class. Nevertheless, when renaming this root class, the maintainer also had to rename its subclass EventInterpreterDispatcher to avoid a name clash between two classes of the same package. The maintainer renamed the subclass into SingleEventInterpreterDispatcher, adding the Single prefix.

Finally, another hierarchy that had inconsistent naming was the TriggeringPolicy hierarchy. This five-class hierarchy had two classes with the Policy suffix, while the rest used the Trigger suffix. The maintainer followed the Policy naming convention therefore the three remaining classes were renamed to use the Policy suffix instead of Trigger. Consequently, this contributed to the full consistency of the hierarchy.

Lessons drawn. In contrast to the two previous projects, this experiment highlighted only inconsistencies in class naming and no errors in the design.

If in Pharo, the suffix defines the concept, it appears that in some cases, the names inside a hierarchy have to stay inconsistent in the sense that their spfixes are not unique within the hierarchy. In those cases, the different values of the last word of the name are more important than keeping consistencies inside the hierarchy. This is the case for the purple hierarchy in the Stargate project.

Whereas for the other projects the inconsistencies were mostly resolved by adding a new spfix (suffix in those cases), in Stargate and Willow the inconsistencies have been mostly solved by deleting the suffix.

Some renamings may lead to other renamings. Indeed, if there is a kind of naming consistency inside the hierarchies, there is another one between a class and its associate test class. Consequently, when renaming the class, the test class is

also renamed even if it already has the suffix Test and is consistent inside its own hierarchy.

5.1.2 Protocol for experiment with Non-Domain Expert / Visualization Experts

Protocol. The purpose of this experiment is to check if users of our tool can easily identify inconsistencies in class names without having any knowledge about the system architecture or the system naming convention. For this setup, no explanation of the tool is useful because the users are tool experts (authors of the paper). Each expert used the tool separately on each project. Then a discussion followed to lead to a consensus. A pull request was made by a single tool expert for each retained renaming proposal. The developers or maintainers of the project then accepted or did not the renaming as any other pull request.

Choice of the project. We chose two Pharo projects in production with the following criteria: (i) none of the two tool experts should have worked on the project before; (ii) the projects needed to be diverse in terms of size, domain, and development team. This led to the choice of Spec, a UI Builder framework[2] and Morphic, a graphics and widget library that are part of Pharo[3] projects.

Two of the authors applied the visualization to both of these projects, then simply made pull requests on their GitHub repositories. Most of the pull requests were accepted by the domain experts who found these changes relevant.

5.1.2.1 Spec Project

Spec is a framework in Pharo for describing user interfaces. It allows the construction of a wide variety of UIs from small windows with a few buttons up to complex tools such as an advanced debugger Fabry and Ducasse [2017]. The Spec2.0 project consists of 795 classes.

We applied the ClassName Distribution to the Spec v2 beta project and proposed an overall of 34 class renamings. All but two were accepted: one class was SpTestApplicationWithLocale which inherits from SpApplication. This class implements a method locale returning the current locale of the underlying platform running Pharo. Spec developers considered the proposed name (*i.e.,* SpWithLocaleTestApplication) inadequate and after further discussions preferred to keep the old name because it stresses the use of the locale variable which they consider more important. The second refused pull request was about the naming of the class

[2] hosted at http://github.com/pharo-spec/spec
[3] http://github.com/pharo-project/pharo

Announcement. The Announcement hierarchy renaming was not accepted because Announcement followed a different naming pattern convention based on the past tense of the last word *e.g.,*FocusChanged and not FocusChangedAnnouncement.

Our analysis was made on the Spec 2.0 beta version of the project. The main developer was more inclined to clean his code and offer its users better and more consistent names. No real documentation such as books and tutorials was already widely written and distributed. This probably eased the adoption of our suggested changes.

5.1.2.2 Morphic Project

Morphic is the name given to the Pharo graphical interface. Morphic was initially developed by Maloney and Smith for the Self programming language, starting around 1993. Maloney later wrote a new version of Morphic for Squeak. Even if the basic ideas behind the Self version are still alive and well in Pharo Morphic, the project has evolved over the years.

In its current version, the project consists of 405 classes spread over 20 packages. Concerning the hierarchies, there are relatively few. Some small hierarchies are homogeneous. There are three big hierarchies that are not homogeneous. The Morph hierarchy is spread over several packages and suffixes. Ten classes were proposed to be renamed to remove inconsistencies and eight unnecessary suffixes.

Inside this hierarchy, the class WindowMorph was shortened to Window, thus not following the superclass suffix. Our proposal to rename it to WindowMorph was refused because the expert considered that because it belongs to the Morph hierarchy it is obviously a morph and that he wanted to keep the class name short. Consequently, four classes, subclasses of Window, were not renamed.

The Announcement hierarchy adopts a scattered vocabulary pattern exactly as in the Spec project. A further 29 classes are concerned but are not proposed to be renamed for the same reason the classes were not renamed in Spec; the naming pattern is different. There are some classes with the Wrapper suffix. The mix between Model and Wrapper inside the hierarchy is not clear for a non-expert of the domain. However, as already seen in other projects the notion of wrapper introduces a kind of decorator and seems to be an accepted naming inconsistency.

Finally, the class CalendarDayMorph is a mono-class, with the Morph suffix. It is certainly a design mistake that it does not inherit from the Morph class since it shares some instance variables with it. It was easy for a non-expert of the domain to discover such an issue since it is manifested by a white box inside a colored spfix box.

In total, 15 renamings have been proposed and accepted.

5.1.3 Effective Class Renaming

Table 5.1 sums up these two experiments by gathering for all projects the number of classes that have been renamed using the ClassName Distribution tool.

Project	Classes	Renamings	Percentage
Calypso v8	716	91	∼13%
Roassal	326	68	∼21%
Spec	795	34	∼4%
Morphic	403	15	∼4%
Stargate	151	12	∼8%
Willow	234	12	∼5%

Table 5.1: Number of classes and renamed classes per project in addition to the percentage of the renamed classes per project with some approximate values.

5.2 Quantitative Evaluation

We applied the ClassName Distribution visualization to a set of 50 Java projects. We discuss in this section how we chose these projects. We then discuss the occurrences of the visual patterns with regard to the global Java naming patterns. Next, we discuss correlations between the metrics of the visualization.

5.2.1 Choice of the projects

For the quantitative evaluation of the tool, we wanted to use the tool on representative Java projects. For this purpose, we have set up the GitHub advanced search to select projects with more than (i) 1000 stars, (ii) 50 forks and 5,000 KB. The number of stars and forks ensure us that the project is used or accepted by the community. The size in KB gives us an indication of the size of the project. Some of the projects are currently still being maintained. The number of Java classes per project is in a range of 179 - 13,653, with a median of 1,711.5 classes and 206.5 packages. Table 5.2 summarizes the data extracted from the selected Java projects.

5.2.2 Protocol

After selecting the projects, we clone each project from its GitHub repository. We then create their ClassName Distributions and export all metrics such as the number of classes, spfixes, types of hierarchies, hierarchy patterns as well as spfix pattern occurrences into a CSV file. Since the sizes of the projects are very different, some metrics are scaled to a percentage in the following way:

Table 5.2: Quantitative analysis of Java projects. **MC** refers to Mono-classes, **H** to Homogeneous hierarchies, **NH** to Nearly Homogeneous and **SV** to Scattered Vocabulary. (Projects are ordered by their number of classes)

name	#packages	#classes	avgChildren	avgDepth	#hierarchies	%MC	%H	%NH	%SV
elasticsearch	1,416	13,653	67	5	467	59.8	64.4	17.3	18.2
flink	1,193	9,154	59	6	404	71.7	81.6	8.9	9.4
hadoop	799	8,183	19	3	715	66.1	88.8	4.8	6.2
openSearch	594	6,627	41	4	266	60.5	64.6	16.9	18.4
sonarqube	645	6,075	37	2	129	82.0	82.1	8.5	9.3
geoserver	655	5,791	26	3	321	64.9	83.4	7.1	9.3
springframework	613	4,794	11	2	223	82.9	92.8	2.2	4.9
druid	321	4,784	121	3	43	44.7	53.4	13.9	32.5
keycloak	776	4,744	38	4	230	61.7	83.4	6.9	9.5
springboot	864	4,325	9	2	91	90.5	96.7	2.1	1
orientdb	439	3,966	40	3	144	57.9	75.6	7.6	16.6
skywalking	1,038	3,267	17	2	112	77.1	84.8	6.2	8.9
jobjc	414	3,014	19	2	111	77.7	81.9	8.1	9.9
cassandra	154	2,533	14	3	129	74.2	78.2	5.4	16.2
pmd	241	2,472	48	3	81	47.2	61.7	13.5	24.6
spotbugs	200	2,347	3	2	70	91.2	70	15.7	14.28
gobblin	403	2,328	4	3	172	76.5	87.2	6.3	6.3
plantuml	228	2,315	32	3	82	49.3	58.5	10.9	30.4
Activiti	354	2,308	23	3	92	66.5	90.2	5.4	4.3
pulsar	343	2,289	23	3	105	71.9	84.7	5.7	9.5
storm	368	1,908	14	2	93	79.8	80.6	9.6	9.6
optaplanner	732	1,883	24	3	65	61.5	95.3	0	4.6
dubbo	467	1,863	8	3	95	74.3	86.3	6.3	7.3
jpexsdecompiler	213	1,811	44	3	86	40.5	65.1	18.6	16.2
mapstruct	534	1,746	5	2	77	86.6	79.2	7.7	12.9
languagetool	189	1,677	30	3	53	58.3	88.6	3.7	7.5
jenkins	123	1,437	10	3	71	75.8	73.2	9.8	16.9
javaparser	126	1294	26	5	34	65.4	67.6	20.5	11.7
jstorm	263	1,289	13	2	70	73.3	72.8	11.4	15.7
nacos	279	1,178	9	2	56	74.7	87.5	0	12.5
exoPlayer	93	967	4	2	26	89.8	96.1	3.8	0
jadx	119	932	53	3	24	37.2	79.1	4.1	16.6
bytebuddy	53	896	11	2	24	79.4	95.8	0	4.1
yacysearchserver	105	858	7	2	41	81.8	65.8	19.5	14.63
lettucecore	85	755	17	2	37	65.6	78.3	10.8	10.8
maven	145	727	6	2	47	73.5	89.3	4.2	6.3

name	#packages	#classes	avgChildren	avgDepth	#hierarchies	%MC	%H	%NH	%SV
micrometer	93	566	12	2	11	78.9	72.7	9.1	18.1
osmdroid	93	477	19	2	17	57.4	76.4	17.6	5.8
arthas	93	474	12	2	18	70.8	77.7	16.6	5.5
dataX	164	460	10	2	25	70.4	96.0	0	4
Servicecomb	116	446	4	2	18	83.63	44.44	50	5.55
guice	42	419	2	2	8	91.8	87.5	12.5	0
javassist	42	415	7	2	14	79.5	42.8	28.5	28.5
conductor	121	401	5	2	18	75.5	72.2	5.5	22.2
halo	82	400	8	2	24	71.0	83.3	4.1	12.5
wechat	27	234	6	2	4	83.7	75.0	0	25
baritone	71	213	7	2	14	53.5	78.5	21.4	0
jsonschemapojo	29	192	1	1	4	94.7	100	0	0
processing	43	190	2	2	18	74.7	88.8	11.1	0
cryptomator	31	179	1	1	2	97.2	100	0	0
median	206.5	1,711.5	12.5	2	67.5	73.9	81.1	7.7	9.5

- Number of mono-spfix hierarchies (*homogeneous*), multi-spfix hierarchies, hierarchies with a *scattered vocabulary* and *nearly homogeneous* hierarchies are scaled to the global number of hierarchies,
- *Blobs*, *Confetti*, *Intruders*, *Snowflakes* and the suspicious spfixes are scaled to the global number of spfixes in the project
- Multi-spfix classes, mono-spfix classes and mono-classes are scaled to the number of classes in the whole project

Finally, we generate a heatmap using Pearson correlation between the metrics to gain a better insight on the relation between the visualization and software metrics.

5.2.3 Java Projects Visual Patterns Analysis

Table 5.2 gathers some metrics concerning the projects. In addition to the names of the projects, we provide information about their size (with the number of packages and the number of classes), the hierarchies (with the average number of children per class, the average depth in the inheritance tree and the number of hierarchies) and the patterns (with the percentage of mono-classes MC, the percentage of homogeneous hierarchies H, the percentage of nearly homogeneous hierarchies NH, and the percentage of the scattered vocabulary hierarchies SV).

Hierarchies. The analysis of the selected Java projects with regard to hierarchies shows:

- **Size of the hierarchies:** The average number of children goes from one to 121. In addition, the average number of inheritance levels ranges from one to six. These two columns show that the inheritance and thus the hierarchy notion is used differently according to the projects.

- **Mono-classes:** Only five projects have less than 50% of mono-classes. In contrast, five of the selected projects have more than 90% of mono-classes. These two observations show that many classes are mono-classes and thus that the number of classes inside hierarchies is often smaller. Our tool is relevant only for those classes.

- **Homogeneous**: Half of the projects in our dataset have more than 80% of hierarchies using the same spfix, including six projects with more than 95% of homogeneous hierarchies. This means that indeed Java projects follow inheritance naming conventions, by dropping the description of the hierarchy behavior in the names of subclasses (either in the suffix or prefix position).

- **Multi-spfix hierarchies**: With regard to the two most recurrent patterns representing multi-spfix hierarchies, half of the projects have fewer than 8% of hierarchies introducing a new spfix (*nearly homogeneous*), which is close to the median of the *scattered vocabulary* with a value of 9.5%. Their presence indicates small violations of naming conventions that can easily be corrected.

Visual patterns. For the sake of space, Table 5.2 provides only general data about the projects without entering into the details of the visual patterns. However, complete data are available[4] and the analysis of the selected Java projects with regard to the visual patterns shows:

- **Use of visual patterns:** Even if all the visual patterns do not appear in each project, globally they appear for the Java projects as we already show it for the Lucene project in Section 4.4.

- **Intruders, Blobs and Confetti:** The percentages of *Intruders*, *Blobs* and *Confetti* are very small (between 0% and 1%).

- **Snowflakes**: The median value of snowflake spfixes from the dataset is more than 20%, with a maximum value of 40%. This is reasonable compared to the median percentage of mono-classes which is more than 70%. This means that only 30% of classes are defined in hierarchies. The fact that more than half of the classes do not belong to any hierarchy raises questions about the

[4] https://github.com/NourDjihan/ClassNamesDistribution-PaperData/\blob/master/JavaProjects/JavaProjects.csv

usage of inheritance and polymorphism in Java but this is outside the scope of this article Tempero et al. [2008].

- **Scattered vocabulary:** In the projects of the dataset, the percentage of the hierarchies with a *scattered vocabulary* ranges from a total absence to more than 30%, with a 9.5% median. It may not only indicate inconsistencies in class names but also presumably architectural inconsistencies, that can be spotted when the used spfixes do not make sense when put together in the same inheritance tree.

Details. Two projects (cryptomator and jsonschemapojo) have an average number of children and an average depth of inheritance tree equal to one. These two projects have respectively two and four hierarchies with a total number of classes of 179 and 192. Inheritance and polymorphism are perhaps underused and our tool obviously does not detect naming inconsistencies since it is based on respecting conventions within hierarchies.

One project (Javassist) has less than half of its hierarchies being homogeneous. It also has the biggest percentage of nearly homogeneous hierarchies. A deeper analysis would be useful for this project since the number of classes is reasonable (*i.e.,* 415) and on average the hierarchies are not large.

5.2.4 Correlations between metrics

The heatmap of the correlations[5] between the metrics showed some intuitive results. The simplest correlation can be found between packages, classes, and spfixes. Indeed, the correlation is positive, so the more packages a system has, the more classes it contains and the more spfixes are used. These spfixes are indicators of the system vocabulary and thus of the services that the system provides.

A positive correlation (0.84) exists between classes of multi-spfix hierarchies and *Blobs*. The more classes there are in multi-spfix hierarchies, the more *Blobs* the project contains. Knowing that *Blobs* are a group of seven or more classes of the same hierarchy using the same spfix, this is considered a good indicator that classes of multi-spfix hierarchies usually use the same spfix or extend the use of the main spfix, and continue following the naming convention.

A positive correlation exists between *suspicious spfixes* and average children. The suspicious spfixes refer to spfixes which are neither used by (i) the root class nor (ii) most classes of the hierarchy. The more children a hierarchy has, the higher the probability of introducing new vocabulary, and hence inconsistencies.

[5] Available online https://github.com/NourDjihan/ClassNamesDistribution-PaperData/blob/master/JavaProjects/JavaProjectsHeatmap.png

Another positive correlation between multi-spfix hierarchies and both *nearly homogeneous* and *scattered vocabulary* supports the previous correlation. Since spfixes of *nearly homogeneous* and *scattered vocabulary* are treated as suspicious spfixes. A new vocabulary has probably emerged when new classes were added. In some cases this is not a problem however, it may also indicate violations of the naming convention or false inheritance.

The last correlation is a negative correlation between mono-classes and the average of children per project. Indeed, when a project contains more mono-classes, the inheritance is not used often which decreases the number of children per hierarchy.

We expected a positive correlation between the number of mono-classes and the number of spfixes. The absence of this correlation strengthens our hypothesis of missed opportunities to group mono-classes in hierarchies.

5.3 Discussion

Here we discuss some aspects of the proposed visualization.

5.3.1 About colors and sizes

During the design of ClassName Distribution, we experimented with several features:

- To convey the depth of a class in its inheritance hierarchy we used its size (the smaller, the deeper). This added more information but proved too cumbersome to interpret.
- At first, every hierarchy had a color, even the homogeneous ones. The result was a flurry of colors, very distracting and drawing attention to the homogeneous hierarchies while the focus should be on consistency violations.
- There is a limited number of distinguishable colors on a screen. We chose a limit of 24 colors.

5.3.2 About Prefix and Suffix

Our analysis of several Pharo points to a general adherence to a suffix convention– see also the work of Butler *et al.,* Butler et al. [2011b]. Concerning the Java projects, we observed that box prefixes and suffixes can be used inside the same project. Our tool automatically selects the prefix or the suffix according to the algorithm detailed in Section 4.7.2.

The automatic identification of the spfix enables the user to gain time considering that she does not have to manually specify it. However, sometimes the numbers of occurrences of prefixes and suffixes are the same inside the hierarchy and the number of siblings is also the same. In addition, sometimes the same word can be used both as a suffix or as a prefix possibly inside the same package, such as Test. The tool arbitrarily chooses the prefix or the suffix, but the user can set it manually.

5.3.3 About Blurry Domains

Some domains seem to naturally lead to non-homogeneous hierarchies. The Morph concept is one of these. A Morph is its own model. While the MVC pattern clearly separates the responsibilities, Morphs tend to blur the distinction. Therefore, the class Browser in Calypso, which could be understood as a model of a code browser, can be implemented as a subclass of Morph. This makes the code more difficult to understand since the reader should always keep this in mind. Developers could change the class suffix (*i.e.,* Browser in BrowserMorph). Our visualization highlights such issues and lets developers consider their naming conventions.

5.3.4 About Changing the Root Class

In some projects, designers decided to introduce a root class for a part or the whole project. Such cases occur when the hierarchies are large or deep, describing several concepts but with a common ancestor. Our tool considers that option. Specifying a new root class reduces the noise in the visualization and removes from the analysis very abstract classes like Object or RSObject.

However, the tool allows users to specify only one additional root class. The users need to choose the one that introduces the most noise to remove it. In future work, we plan to introduce the possibility for the user to add several root classes.

5.3.5 Renaming and Inconsistencies

Our tool enables the users to identify inconsistencies in class naming inside hierarchies. The experiments performed with experts of the domain or of the tool have shown that this is helpful. However, even a consistent hierarchy can be renamed. The hierarchy is consistent but the chosen spfix is not adapted or expressive enough. This is for example the case of classes that, in Calypso until v8, end with P1, P2... as suffixes (classes in the two last rows of packages) in Figures 4.3 and 4.5. In version v9, Figure 4.6, these classes have been renamed with Mock, a more evocative suffix. The name of the spfix on top of the corresponding box in the visualization can help to identify such cases.

5.4 Threats to Validity

There are several validity threats to the design of our experiments.

Internal Validity: Is there something inherent to how we collect and analyze the data that could skew our findings? Regarding the tool, we used it both on Pharo and Java projects. Both ecosystems have different cultures regarding inheritance use and naming conventions. Pharo projects tend to have classes more structured in hierarchies. Our experience with Java projects showed a lot more mono-classes. Our tool supports different naming cultures by enabling the user to choose a visualization taking into account prefix, suffix, or both spfixes. The presence of interface implementation (Java), trait usage (Pharo), or multiple inheritances (C++) should also be assessed.

Regarding the experiments with users (Domain Expert / Tool Learners and Non-Domain Expert / Tool Experts), *in fine*, they both required access to experts because, in the end, the proposed renaming should be accepted. Due to the size of the community and our access to Pharo experts, we performed these experiments only on Pharo projects. They have illustrated how our tool can help even experts of the domain to identify inconsistencies in the class naming. With the quantitative experiment done on Java projects, we show that the visual patterns also occur in projects of this language.

External Validity: Are our results generalizable for practice modernization? Concerning other object-oriented languages than Pharo and Java such as Python or C++, we did not apply our tool because we do not have yet a parser to have the abstract model of the project. Therefore, the external validity is limited in terms of generalizing the results to software written in languages other than Pharo and Java. The approach itself however can be applied to software written in any object-oriented programming language as long as there is a naming convention supporting the identification of words composing the class names: the uppercase letters in our case or a separator between words in a snake format.

Regarding the number of experiments, we are aware of the fact that we experimented on only a few projects hence the external validity is constrained by the small sample size. However, even if it is easier to access experts in the Pharo community, they have to be available. The presented projects are real-world, reasonably sized projects with a couple of hundreds of classes, many contributors, a long history, and very different domains. We tried to compensate for this threat by evaluating different setups with two qualitative and one quantitative analysis. Concerning the quantitative experiments, we clearly explained how the Java projects were chosen and provided results for 50 projects. Yet as all experiments on software systems, more cases should be considered and analyzed.

Reliability: To what extent can the results be reproduced when the research is repeated under the same conditions? We provided users the link to the tool GitHub

repository for further usage, as we also explained each and every configuration of the tool. Furthermore, we described the algorithms used to implement our approach in Section 4.7. For reproducibility purposes, we ordered packages, spfixes, and classes alphabetically, by size, and by color for the spfix boxes. The interaction of the tool may have an impact on its users.

5.5 Conclusion

We conducted a consequent assessment of the visualization with real developers and open-source software structured in two different setups: in the first one, we asked *domain experts* to use the visualization. Three groups of engineers in different countries and projects (France, Chile, and Argentina) applied our visualization to the software they develop or maintain. We recorded individual video sessions and assisted to final discussions within the team around class names. This resulted in respectively 91, 68, and 24 class renamings in their projects. In the second setup, as authors of the visualization and the tool we applied our tool to two projects we didn't know before. These two experiments led to 24 to 91 renamings per project showing that (i) the visualization can help experts of a project to identify irregularities in class naming and (ii) to use the visualization. It is not mandatory to be an expert in the domain to propose relevant renamings. Finally, we applied our visualization to 50 Java projects and identified the presence of the visual patterns in most of them. This experiment shows that our visualization can be used both on Pharo and Java while considering the specificities of these languages.

CHAPTER 6

A New Generation of ClassBlueprint

Contents

In object-oriented programming, classes are the primary abstraction mechanism used by and exposed to developers [Shalloway and Trott, 2005] In the preceding chapter, we underscored the paramount significance of well-chosen class names and endeavoured to address the issue of erroneous class names. In the present chapter, we delve into the intricacies of class content and explore their inherent qualities. Indeed, understanding classes is key for the development and evolution of object-oriented applications [Smith, 2020]. One challenge however faced by developers is that while classes are conceptually structured entities, in integrated development environments (IDEs), they are often represented as a blob of text. IDEs typically present classes as textual files, which may contain a large amount of code, making it difficult to grasp the structure and relationships within the class at a glance. This representation of classes as text can hinder the developer's ability to quickly understand and navigate the codebase. It becomes challenging to visualize the organization of the class, identify its methods and attributes, and comprehend how such components interact with each other.

The idea behind the original CLASS BLUEPRINT [Lanza and Ducasse, 2001] was to visualize the internal structure of classes in terms of fields, their accesses, and the method call flow. Additional information was depicted using colors. This visualization proved to be an effective means to support program comprehension. Subsequent observations however revealed certain limitations (Section 6.2).

We propose CLASS BLUEPRINT V2 (in short BLUEPRINTV2), which in addition to the information depicted by CLASS BLUEPRINT also supports dead code

identification, methods under tests, and differentiation between class and instance level components (Section 6.3). In addition, BLUEPRINTV2 enhances the understanding of fields by showing how fields of super/subclasses are accessed. We present the enhanced visualization and detect some class conception patterns (Section 6.4). Finally, we present both qualitative and quantitative evaluations and report on a first validation with 26 developers and 18 projects (Section 6.5). This work has been published in the *IEEE Working Conference on Software Visualization (VISSOFT2022)* [Agouf et al., 2022b].

6.1 Classes in Object-Oriented Programming

In object-oriented programming (OOP), classes serve as the primary abstraction mechanism that developers use and interact with [Shalloway and Trott, 2005]. The concept of abstraction in object-oriented programming is fundamental as it helps developers in representing real-world entities as objects characterized by certain attributes and having a behaviour on their own [Gamma et al., 1995]. The significance of well-structured class code cannot be overstated as it enhances the readability therefore the understandability of the class in its contextual domain and its interdependencies with the other components. A well-structured class promotes its reusability and extensibility because of its clear responsibility described in its cohesive methods which can be reused in other parts of the software and/or extended to create new class behavior [Martin, 2009].

Conversely, intricate classes not only impose frustration upon software maintainers due to the arduous nature of comprehending their content but also result in more profound ramifications that significantly amplify the time and effort required for maintenance tasks [Feathers, 2005]. These intricacies hinder class readability and complicate its maintenance, impeding effective communication among team members in collaborative maintenance scenarios. Moreover, complex code exhibits heightened sensitivity to risks and the emergence of new issues during maintenance, increasing the likelihood of unexpected bugs when modifications are made to the codebase. Furthermore, in the pursuit of enhancing the trustworthiness and reliability of software which is of paramount importance for its longevity, it becomes harder to impose unit tests of big and complex classes.

6.2 Limits of CLASS BLUEPRINT

The CLASS BLUEPRINT visualization was created to help developers understand class structures Ducasse and Lanza [2005], Lanza and Ducasse [2001]. It decomposes classes into layers representing the call-graph going through external, internal, and accessor methods. This decomposition into layers organizes the method

call-graph and allows one to see which attributes are accessed by which methods, directly or through their accessors (see Figure 6.1).

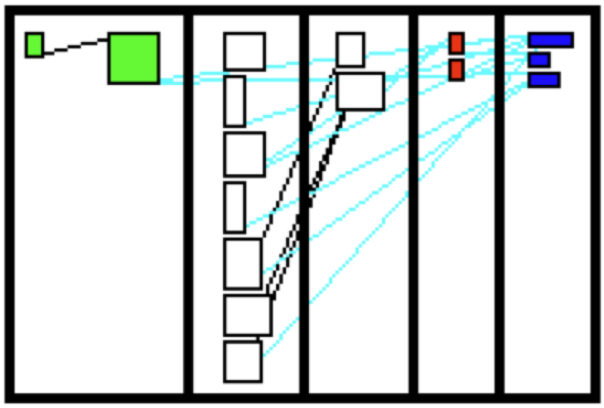

Figure 6.1: A class blueprint from Lanza and Ducasse [2001] with 5 layers: initialization, interface, internal implementation, accessor, and attribute.

CLASS BLUEPRINT has a number of limitations:

a) *Tests*: does not show whether a method is covered by a test: When CLASS BLUEPRINT was initially developed, continuous integration, version control, and tests were not common development practices. Hence, CLASS BLUEPRINT did not take into consideration test classes or test methods. Today, tests are the gate towards a better and faster understanding of the software and ensure software robustness. Easily distinguishing tested and untested methods indicated the degree of reliability of the class.

b) *Dead Code*: It does not heed dead code, *i.e.,* if a method is used or not as such it misses an opportunity to provide information about effective class functionality: CLASS BLUEPRINT represents all methods of a class; even if these methods are dead. However, dead code hinders the understanding of classes Fard and Mesbah [2013], Mantyla et al. [2003]. Yamashita *et al.,* report that dead code detection is desired by software professionals Yamashita and Moonen [2013]. It avoids project practitioners to waste time on reading, understanding, and maintaining irrelevant code Eder et al. [2012].

c) *Instance vs Static*: The interplay between instance and class side (static) is not supported. This is because all methods are classified into four layers only, concealing the details about such a property in the class structure.

d) *Cyclomatic Complexity*: The method cyclomatic complexity is not revealed: Beside lines of code, CLASS BLUEPRINT gave limited information about the code quality such as method complexity Henderson-Sellers [1995], Shepperd

[1988]. The cyclomatic complexity measures the number of linearly independent paths of a method Watson and McCabe [1996]. It is of interest to practitioners responsible for maintenance activities since methods with high cyclomatic complexity tend to be more difficult to understand and thus to test and maintain.

e) *Layer*: The accessors layer takes an unnecessary amount of space for a poor return of information. This layer covers the direct accesses of methods to attributes since the connection from a method layer to an attribute layer passes by the accessors layer.

f) *Hooks*: The reader has no clue if a given method is a hook in the system (a method being already defined in the subclasses), and in general CLASS BLUEPRINT disregarded the display of the superclass attributes used in the class but focuses on the global inheritance relation between superclass and subclass blueprints. CLASS BLUEPRINT did provide an inheritance perspective to highlight the inheritance relationships of a given class with its ancestors and descendants: The class blueprints of the different classes of the hierarchy are gathered inside a single visualization. They are linked through inheritance, but also access, when methods of the subclasses use attributes defined in a superclass or invocation in case a method of a subclass, invokes one inherited method for example. In practice, this representation with several CLASS BLUEPRINT visualizations leads to complex visual elements and makes the understanding more challenging [Diehl, 2007b].

Moreover, on the one hand, CLASS BLUEPRINT succeeded in conveying information about method redefinition, using colors to distinguish between extending and overriding methods. On the other hand, the CLASS BLUEPRINT failed to stress out the methods redefined in the subclasses (overridden methods). Spotting overridden methods allows the user to detect interface classes defining a generic behavior. This makes it easy for developers to understand the hierarchy from the studied class without having to read the source code of each and every subclass.

Our goal was to make up for all the above limitations, proposing a revisited and modern class blueprint visualization.

6.3 The New Generation of CLASS BLUEPRINT Visualization

As its ancestor, BLUEPRINTV2[1] focuses on individual class structure visual representation. It supports the understanding of each class separately. The focus is put on methods call-graph and how methods access the studied class attributes in addition to the superclass attributes. Methods and attributes are represented as nodes whose color maps some semantic information. In addition, the node size conveys some properties: The height of a method node reflects the number of lines of code, and the width, the number of outgoing invocations. As for the attributes, the node height corresponds to the number of direct accesses from methods within the class, the width concerns the number of external accesses from methods of the same hierarchy of classes. To ease the understanding, methods are positioned in different columns depending on their interactions with methods of other classes, or their nature (such as initializer, internal, getter, etc). BLUEPRINTV2 adds the following features (see Figures 6.2 & 6.3):

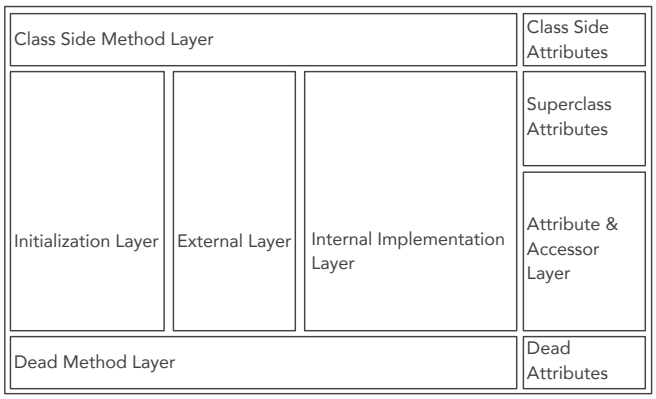

Figure 6.2: Layout of CLASS BLUEPRINT V2: class level is on the top, instance level in the middle, and dead entity on the bottom. The middle layer presents information via layers.

- *Static (or class side) entities* are placed in a dedicated area on top of the visualization and separated from the instance side methods. The calls between the class and instance sides are represented. Class attributes are also represented in a distinct area in the top right corner.

[1]The link to the GitHub repository of the visualization can be found here: https://github.com/NourDjihan/ClassBlueprint. All the instructions on how to use the visualization are explained in the Readme.

- *Dead branches* are identified and separated in a specific layer at the bottom of the visualization (the cemetery). Dead attributes as well as their accessors are also separated at the bottom right corner.
- *Getters and setters* are merged into annotations around attributes to gain space. In addition, lazy initializers are handled as a kind of accessor.
- *Superclass state usage*: access to superclass attributes is represented in a separated area on top of the one of instance side attributes layer. Accesses to superclass state from local methods are represented.

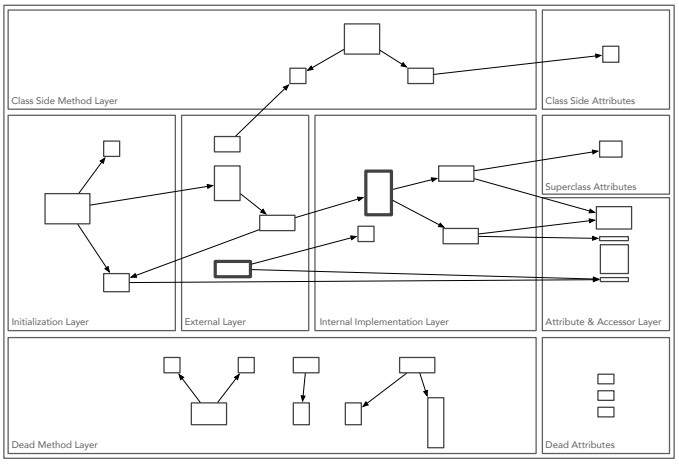

Figure 6.3: A colorless BLUEPRINTV2 with the graphical representations of methods, attributes, and accessors using height and width metrics. The arcs represent calls between methods and accesses to attributes.

These points are described in the following subsections.

6.3.1 Layers

As its ancestor, BLUEPRINTV2 classifies methods in different layers represented in columns.

Horizontal Layers. In addition, in BLUEPRINTV2 the visualization has been split into 3 horizontal layers.

The *top* layer corresponds to the *class* side (static in Java). It gathers on the left the methods connected within a call-graph and on the right the *class side attributes*. In Java, the class side is specified with the static keyword. In Pharo, these are entities of the metaclass.

The *bottom* layer corresponds to dead code. Once again, methods and attributes are separated (methods on the left, attributes on the right). It is possible to see a call-graph in the dead code layer. Indeed, a method is considered dead if it is not called in the project or if it is called only by dead methods. Consequently, dead branches can also be identified. This definition of dead code may lead to false positives, in particular in the case where API methods are called by external projects, not under analysis.

Vertical Layers. First, on the left, the *initialization* layer gathers the methods responsible for object creation and initialization (*e.g.* initialize methods in Pharo and constructors in Java). Then comes the *external* layer containing methods that compose the external interface of the class. Such methods are either invoked by methods of the initialization layer or declared public or protected in languages supporting modifiers, or invoked by methods outside the class. In third comes the *internal implementation* layer representing the core of the class, *i.e.,* methods that are not supposed to be exposed to the outside of the class. It contains for example private methods or methods invoked by other methods of the same class. We removed the *accessors layer*: the setter and getter methods are on top and below their attribute, respectively (see Figure 6.4). It compacts the visualization without losing information: the developer can, at a glance, see if the attribute has a getter or a setter, is directly accessed, or is used through its accessors (if present).

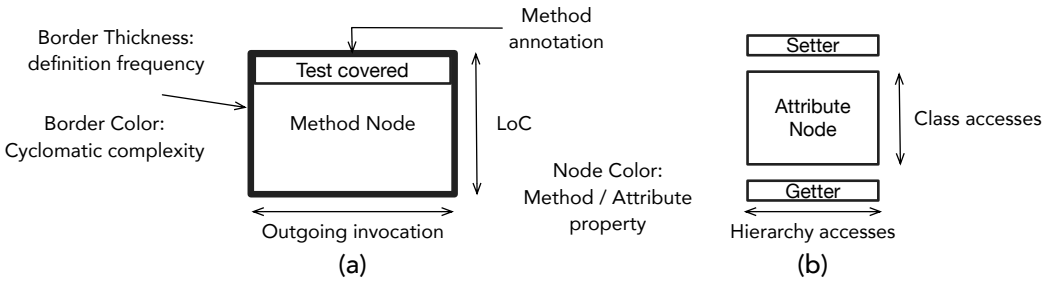

Figure 6.4: Sketch of nodes: (a) For methods, size, border thickness, and color convey information. (b) For attributes, size, and color.

The *middle* layer corresponds to instance side methods and not dead code. It follows the vertical layer decomposition.

6.3.2 A Class Inside a Hierarchy

BLUEPRINTV2 puts the studied class in the perspective of its hierarchy. For this purpose, first, the inherited attributes are separated from the ones defined in the studied class and put in a different layer than the local attributes layer. In addition,

attributes accessed by at least one method in the subclasses of the studied class are colored in dark green whereas the other ones are colored in dark blue. This allows developers to see how the class is using a superclass state and if their state is directly used in subclasses.

Second and as in the initial CLASS BLUEPRINT visualization, extending methods (performing a super invocation) are colored in orange and overriding methods (method redefinition without super invocation) are colored in brown. In addition, in BLUEPRINTV2 we introduced a color to spot *overridden* methods (*e.g.,* methods redefined in subclasses) are colored in pale orange. This information gives an idea of how the methods of the class are extended. In addition, to indicate if an abstract method is redefined in the subclasses, they are marked with a pale orange square (In Pharo, classes may have abstract methods not redefined in the subclasses, moreover even concrete ones).

6.3.3 Additional Indications

In addition to the extensions presented above, BLUEPRINTV2 introduces other new features:

- *Cyclomatic complexity.* To complement the Lines of Code and fan-out of the method, we indicate the method with high cyclomatic complexity by coloring their border in red. Indeed when a method has a cyclomatic complexity higher than a given threshold fixed to five in Pharo, its border color is red.

  The thresholds have been defined following JIT compiler practices Deutsch and Schiffman [1984], Miranda [1987, 2011]. They can be changed if needed as discussed in Section 5.3.

- *Tested methods*. To easily identify if a method is called from a test method, in BLUEPRINTV2, tested method nodes have their superior third in green. Note that we only use static information: it does not indicate whether the associated test passes or not.

- *Dead entities*. An *attribute* is dead if it has no incoming accesses, meaning that no external/internal method or accessor is accessing it. Because the visualization treats only direct accesses, then an attribute only accessed by its accessors it is not considered dead. A *method* is considered dead if *the method is not invoked by other methods*. Note that an abstract method is dead only if the method itself is not called and none of its reimplementations in the subsystem are invoked. In addition, initialization and test methods are not considered dead methods to limit false positives.

- *Monomorphic, Polymorphic, & Megamorphic entities*. A monomorphic entity refers to a method with a unique name within the entire project. Monomorphic methods are typically designed for specific and well-defined purposes, and each method serves a unique functionality without being shared by other methods. On the other hand, a polymorphic method is one that is moderately named across the project. Polymorphic methods are often used in multiple contexts or scenarios, providing a certain level of flexibility and adaptability to various situations. While megamorphic entities are methods frequently named in the system, meaning that several methods share the same name throughout the entire project. Megamorphic methods are often employed as general utility methods, serving common functionalities shared by multiple components within the project.

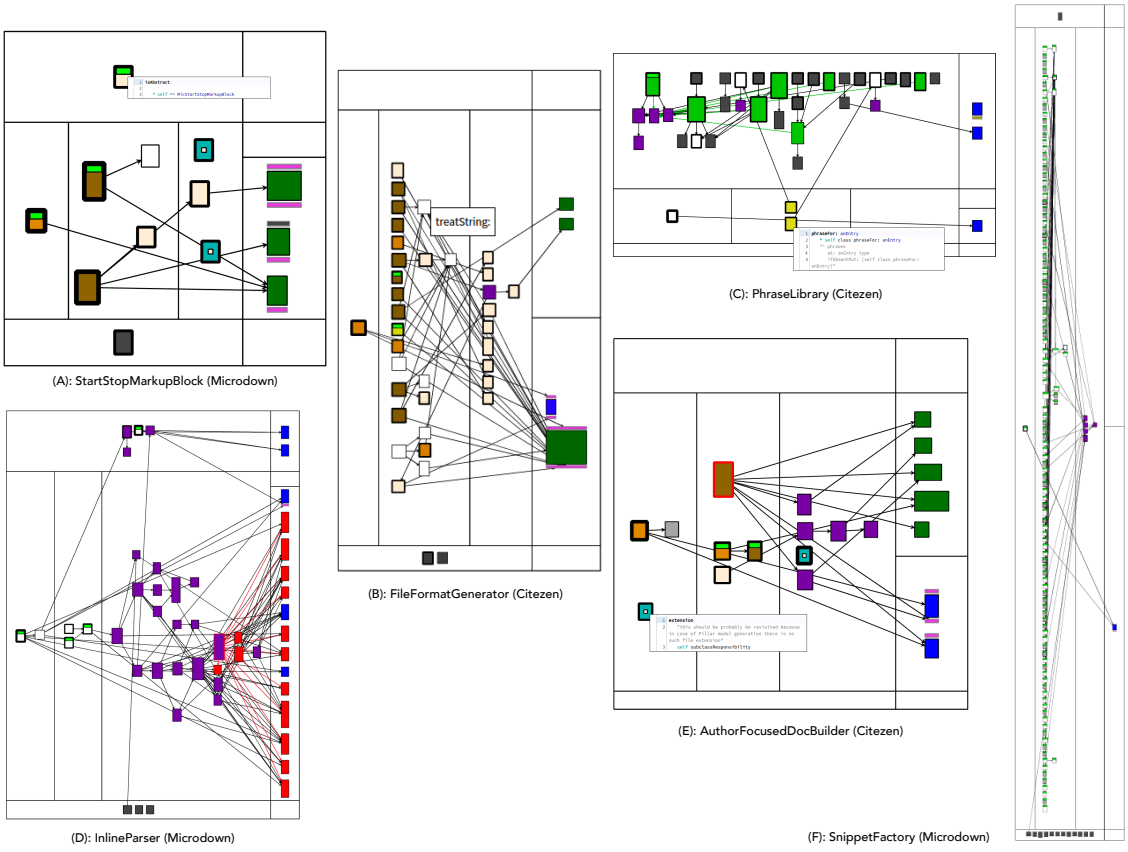

Figure 6.5: Some examples of BLUEPRINTV2 on Citezen (a bib library) and Microdown (a markup language).

Properties	Description
Constant	Gray
* Dead code (attributes or methods)	Black
* Invoked from a test	Green annotation
Extending	Orange
Overriding	Brown
* Overridden	Pale orange
Abstract	Cyan
* Abstract and Reimplemented	Inside pale orange square
Delegating	Yellow
Internal Implementation	Purple
* Test Method	Pink
Setter or Getter	Magenta
* Lazy Initializer	Olive
Accessed locally	Blue
* Accessed by subclass	Green

Table 6.1: Methods and attribute properties. * mark new compared to CLASS BLUEPRINT

6.3.4 Interactions

BLUEPRINTV2 is automatically computed and displayed for a class. It is an interactive visualization that is included in a tool to enable interactions.

Access to the code. A classic mouse hover displays the method name as depicted in Figure 6.5-(B). This example illustrates the event of a mouse hovering a method node colored in white, displayed in the externals layer, and named treatString. Additionally, a Shift pressed with a mouse hover displays the method corresponding code. Figure 6.5 illustrates three examples annotated with (A), (C), and (E) displaying the source code of methods contained in each class. The StartStopMarkupBlock-(A) visualization shows the source code of an overridden class side method named isAbstract. The PhraseLibrary-(C) class presents the source code of the phraseFor: delegating method colored in yellow. Finally, the AuthorFocusedDocBuilder-(E) class with the source code of an abstract initialization method colored in cyan named extension.

Call-graph. To better follow the call-graph, a left-click on a method displays in red its outgoing invocations as shown in Figure 6.5-(D). The InlineParser class visualization shows the outgoing calls from the linkOrFigureProcess: method (with red border) to three methods in the same internal implementation layer and its

access to most attributes of the instance side (eleven attributes). These remain red until clicking again on the selected method. Thus clicking on another method adds new invocations in red and so on. It enables the user to better see and understand the call-graph of the studied method in the scope of the class.

Similarly, it is possible to highlight the incoming invocations of a method with a right-click as shown for the locationMonthYear method in Figure 6.5-(D) (second purple method of the first branch on the left of the class side methods). The incoming invocations of a method and the calling methods are highlighted in green. Adding new methods in the call-graphs and deactivating is also possible.

6.4 BLUEPRINTV2 in Practice

This section presents some examples of the BLUEPRINTV2. Each example brings attention to the internal structure of the class visualizations depicted in Figure 6.5. The visualizations are grouped in one figure to gain space and to easily compare different class structures since the visualizations are put close to each other. Each visualization is annotated with a letter to ease reference hereafter.

6.4.1 A Simple Class

The visualization of the StartStopMarkupBlock class (Figure 6.5) from Microdown project) shows that it has few methods. An initialization method is colored in orange showing that it extends its super implementation. It has a thick border because it is a megamorphic method called initialize. Three overridden methods are colored in pale orange on the class side, external, and internal implementation layers. Two overriding methods are colored in brown each defining a new behavior in the externals layer. Moreover, two abstract methods colored in cyan with a pale orange square indicate the presence of reimplementations in the subclasses. The class contains a dead method colored in black at the bottom of the visualization. Only three methods are under test (as identified by the green node annotation).

The class also contains three attributes each with both accessors however, these attributes are all accessed directly from the class methods without using their accessors and are colored in green meaning that they are accessed directly in the subclasses. Furthermore, the second attribute has a setter on top, colored in black indicating that the setter is not used in the class nor by methods of other classes, thus is dead. The other accessors are colored in magenta meaning that they are used by methods of other classes.

6.4.2 A Class Defining and Redefining Hooks

The FileFormatGenerator class (from the Citezen project) specializes in several superclass methods: it includes four methods extending an inherited behavior colored in orange and nine methods in brown overriding their superclass methods. Moreover, the class provides 13 new methods colored in pale orange overridden in the subclasses. With such data, we see that the class integrates well with its superclass and defines some important methods for its subclasses. Only two methods are tested (as indicated by the green annotation). The class accesses directly two attributes of its superclasses. One of its attributes (colored in green) is heavily used by its subclasses.

6.4.3 A Static Class: Playing a kind of Factory/Builder

The PhraseLibrary class from the Citezen project has an unconventional design. Most of its methods are defined on the class side. It acts as a kind of object factory composing kind of sentences. The PhraseLibrary has two instance side methods in the externals layer colored in yellow delegating to the class side methods for creational purposes. An example of a lazy initializer is depicted in the PhraseLibrary class visualization: the setter of the first attribute of the class side (on top), colored in olive under the attribute. The lazy initializer has no incoming calls inside the class however, its olive color indicates its use in other classes.

6.4.4 A Large Internal Class with Dead Code

Classes such as the MicInlineParser class are referred to as *Single Entry* Lanza and Ducasse [2001], indicating that the class has one or few entry points from its externals layer and a wide internal implementation layer, in this case, composed of 22 methods and many invocations between these methods. Such classes are designed to deliver a one-complex functionality.

Yet, what draws attention to this class are the dead methods at the bottom layer of the visualization. Indeed, three methods inside this class are not invoked and not used in the project. Because dead code, when executed, is memory and time-consuming this leads to believe that the class needs refactoring to improve the overall performance of the program.

6.4.5 Accessing Superclass State

The AuthorFocusedDocBuilder class does not have class side nor dead nodes. Thereby, the call-graph of the methods inside the class appears clearly going from the left initializers, passing by the externals, the internals, and finally the attributes. This class

contains many coloured nodes, for instance, in the initialization layer, a method coloured in orange extends its superclass method and has direct access to both attributes of the class.

The externals layer contains a complex big method buildBody coloured in brown with a red border, which directly accesses most attributes of the class except for the superclass attribute fieldOrder, and the instance side of attribute bodySpecification accessed through its getter. Such a case of direct access and through the accessor in the same class shows inconsistency in the design of the class. Nonetheless, the superclass attributes are all accessed by methods of the subsystem, hence their green colour. The width of the first four superclass attributes gradually changes forming a cascade shape, this indicates which attribute is more accessed by methods of the subsystem.

6.4.6 A Factory Class with Tested Methods

Finally, the SnippetFactory class has an externals layer with 120 methods. This class is a holder of ready-to-be-used document elements that are essentially used by tests.

The externals layer contains constant methods in gray returning each predefined value and methods colored in white, meaning that they do not belong to the classifications defined in Table 6.1. Such methods contain a small portion of source code, hence their small size, where each is performing an operation and returning its result.

Moreover, almost all methods of the externals layer are annotated with green on top indicating that each is called by at least one test method. This class also has dead methods (bottom layer). Such dead methods are methods returning document elements that are not used by the tests. They are effectively dead methods.

6.5 Evaluation

This section explains the process of the evaluation of the BLUEPRINTV2. We proceeded with both *qualitative* and *quantitative* evaluations. Such evaluations try to answer the following research questions:

- How does the visualization support the understanding of the class structure?
- How does the visualization help in assessing the quality of the class?
- How satisfied are the different profiles of participants with the proposed visualization?

6.5.1 Protocol

We invited developers from the Pharo community to participate in our research: We sent an email welcoming participants that would be interested in evaluating a new visualization. Twenty-six developers joined our evaluation. Then, to explain and guide the participants through the experiment, we scheduled both group and individual meetings according to the participants' availabilities. If several participants were available at the same time we proceeded with a group meeting. On the opposite, we scheduled individual meetings. During those meetings, we first explained the visualization and its motivations, then described to the participants the process of the experiment:

a) Select a project they wish to analyze

b) Use the visualization on the selected project

c) Record their screen during the experiment

d) Write a report summarizing their findings

e) Fill in the post-experiment survey

The meetings took from 10 minutes when individually, to 25 minutes in groups. This difference is principally due to the adaptation of the tool to the chosen project and the possible interactions between participants while in groups. Nonetheless, the content was the same for each participant.

Projects. The selected projects vary in number of packages, classes, and methods. Table 6.2 summarizes this variety of projects.

The same project could have been analyzed by more than one participant. For example, Iceberg was analyzed by two participants, MooseIDE by three, and OpalCompiler by two; but separately.

Participants. The participants of the experiment come from diverse backgrounds and have different statuses: interns, PhD students, developers, and researchers. Consequently, their years of experience in programming also vary. Six participants have between [1-5] years of experience. Nine participants between [6-10] years of experience. Two participants have 14 and 15 years of experience each, and nine participants have over 20 years of experience.

In regards to the projects, the participants either used these projects before, developed them or are responsible for their maintenance. The expertise degree of the participants on the selected projects is let to their own appreciation. The survey showed that only 10% of the participants consider themselves newbies to

Project	Classes	Renamings	Percentage	
Avatar	2	18	6	Proxy
Sindarin	3	18	14	Debugging
MoTion	2	35	5	PM
Clap	5	47	8	Parsing
Slang	2	73	29	VM
Polyphemus	3	79	9	VM
AST-Core	3	101	21	DSL
Reflectivity	5	114	13	DSL
OpalCompiler	3	156	15	Compiling
Druid	1	170	12	VM
Seeker	2	236	9	Debugging
MooseIDE	16	250	8	Analysis
Polymath	60	309	11	Computing
Refactoring	12	378	6	Refactorings
AIPharo	85	424	6	AI
Roassal	39	445	12	Visualization
Iceberg	11	488	10	VCS
fylgja	73	941	15	Migration
microdown	29	268	11	Parsing

Table 6.2: The number of packages and classes in each project, and of median methods in each class, in addition to project domains. In the table, the order is descending according to the number of classes in each project. The abbreviations refer to VM (Virtual Machine), VCS (Version Control), PM (Pattern Machine), DSL (Domain-Specific Language).

the selected project, while 35% had some basic knowledge about the source code. Another 10% of the participants with advanced knowledge of the project and finally 45% of the participants are project experts.

6.5.2 Data Collection

Once we received the data (screen records, reports, and survey filled by all participants), we watched the screen records looking for participants' behavior. Particularly, a focus has been put on (i) the participant's first impression in regards to the visualization, (ii) the most used features of the tool, and (iii) if other tools have been used besides the BLUEPRINTV2 and what they are. We then, read the reports to have a better insight into their actual impressions and findings. Finally, we collected both data from the reports and the survey as explained in the next section.

6.5.3 Data Analysis

As mentioned prior, the evaluation of the BLUEPRINTV2 consists of two parts: qualitative and quantitative. The qualitative evaluation is based on the screen records and the reports sent by the participants. The quantitative evaluation is built upon the post-experiment survey.

Qualitative evaluation. When observing the screen records of the participants, it was brought to our attention that participants instinctively followed two complementary approaches: *Flight over* and *Plunge in*.

Flight over: The flight over consists of quickly navigating the visualizations of the classes one by one. It allows the user to visually detect the important classes and the less important ones based on the amount of information held by the visualizations (nodes and colors). It also allowed some users to detect:

- Empty classes: without methods nor attributes;
- Big classes: with lots of methods;
- Complex classes: with methods with high cyclomatic complexity *i.e.,* method nodes with red border;
- *Class-side* methods: with a considerable amount of method nodes placed in the class side layer;
- *Hook* classes: with noticeable hook nodes colored in pale orange;
- *Dying* classes: with several dead methods, or attributes, positioned at the bottom of the visualization;

- *Tested classes*: with tested methods *i.e.,* the green annotation in method nodes. More specifically, here, it is the proportion of tested methods that have been appreciated by the participants.

Plunge in: The plunge in consists of focusing on some classes for further investigations. This provides the user with a more in-depth analysis of the class internal structure, and the possibility to use more helpful interactions with the visualization such as the shift + mouse hover to quickly view the source code of the methods.

- *Code duplication*: some duplicated code was detected by several participants when method nodes have the same size and color, inside the same class or in other neighbor classes.

- *Complex classes*: most participants reported finding complex methods inside classes due to the size of the nodes or the red border color. They intend to divide them into small blocks of source code.

- *Dead method analysis*: some participants were surprised to find many dead methods. Such an outcome motivated them to inspect the senders of the "dead methods" using other tools than the visualization, (e.g., system browser and cross-referencer).

- *Dead code detection*: the detection of dead code was also surprisingly possible due to the length of the method nodes. One of the participants investigated long methods relatively used in his studied system and found a few which contained internal dead code which was not removed.

- *Commented methods*: some participants identified peculiarly long methods but not necessarily complex ones. Then, when navigating the source code of the method nodes (through a mouse hover), participants found that the length of the method node in fact reflects the long comment inside the method definition. One of the participants suggested adding an annotation at the bottom of the method node (since the top might be full with the test annotation), to demonstrate the presence of a comment.

Quantitative evaluation. Participants were asked to answer a post-experiment survey to give feedback about the visualization using a five-Likert scale. We asked the participants if:

- *The visualization helps in understanding the code/state of a class is reused:* Most participants (46%) answered that they agree on this point, while others (15%) strongly agreed. However, over 23% said that they disagree, and 15% were undecided.

Some participants in their reports mentioned that they appreciated the feature of adding the superclass attributes in the visualizations. This helped them understand which attributes of the superclass are used in the class under analysis, and which methods are accessing them. Such a feature also reduces the time spent on understanding the relation with the superclass.

- *The visualization helped in understanding the reused code from the superclasses:* Over half of the participants (53%) were undecided, where 19% disagreed, 23% agreed, and 3% strongly agreed. Because some projects did not use inheritance, hence the visualization was not answering this question which justifies the amount of indecision in this case.

- *Did the visualization help in understanding class/instance side communication*: Over 53% answered that they agree, and 19% strongly agreed. Other participants of 19% answered that they were undecided about this point, and a percentage of 7% answered that they disagree.

  We believe that the classification of the class side methods on top and the instance side methods in the middle reduces the cumbersome links between method nodes in the middle layers. Moreover, the participant who analyzed the *Microdown* project mentioned: "The MicAbstractDelimiter class shows the nice interplay between class and instance side methods".

- *Is the design of the class well summarized in the visualization:* Over 7% and 38% of the participants strongly agreed, and agreed, respectively. Among the participants over 30% were undecided and 23% disagreed.

  This is also understandable because the visualization summarizes the structural relationships and not the design of the class itself. In some cases, indeed the design is well shown for instance visitor classes, factory classes, and builder classes, etc. But not all design is reflected by the class blueprint. Nonetheless, one participant mentioned: "I was able to follow for each method clearly what are the methods of the same class that are injected inside of it", referring to the outgoing invocations between methods.

- *Does the visualization help in detecting dead code?* Over 26% and 46% of the participants answered that they strongly agree and agree, respectively. However, 11% answered that they strongly disagree, 3,8% just disagree, and another 11% were undecided.

  All participants reported finding dead code in their projects. The participants who analyzed the same projects on the one hand found the same dead methods and on the other hand different dead methods. This is one of the reasons which motivated us to agree on analyzing the same project by two

or three participants. Including one participant who eventually refactored his code and justified in his report: “The visualization also helped me quickly identify dead code and eliminate it. As this is a new project (early stage of development) I didn’t remove all dead methods or classes, but in other kinds of projects I would do it”. Another participant expressed: “Dead methods correspond mostly to unused code that I forgot to remove”.

- *The visualization helps in detecting complex methods:* Almost all participants appreciated this feature, including 46% who strongly agreed and 38% who agreed to this assertion. The other participants were 11% undecided, 3% did not agree. The participant responsible for maintaining the Roassal project mentioned and we cite: “With the height representing the lines of code and with the red border. It was easy to find the complex methods in this class. This is an anomaly because they are long examples that maybe should be split into classes”. He found long complex methods that represent examples of how to use Roassal.

  When reading the reports not all participants found complex methods in their projects, which might explain such a decision. However, several others mentioned the presence of complex methods in their projects, including some who consider investigating the complexity of such methods for correction purposes.

- *The visualization helps in identifying tested/untested methods:* Over half of the participants also appreciated this feature, including 34% who agreed and 34% who strongly agree. The other 19% were undecided and only 7% disagreed and very few of 3% strongly disagreed.

  Some participants in their reports mentioned that the visualization helped them identify the weak spots (not tested methods) in their source code, which they intend to reinforce. The *Microdown* participant found this feature useful and we quote: “In the MicHTMLDoc class we could exclusively see the tested and untested methods”.

- *The visualization is scalable for large classes:* Among the participants, 3% strongly agreed, and 42% agreed that the visualization was scalable for their classes. The other 30% were undecided, 15% disagreed, and 7% strongly disagreed.

  This question is also relative to the project (in case the project contains big classes), hence the diversity of answers. As with any visualization including nodes and connections between those nodes, the bigger the class is the harder it becomes to display all the pieces of information at once with a clear classification of the layers and connections between the nodes. Nonetheless,

other participants found that the classification of the methods on the right and attributes on the left in big classes helped them to (i) better see the attributes and (ii) more clearly identify the access to these attributes, and (iii) better distinguish the class side methods.

- *The visualization is easy to use:* Three-quarters of the participants (76%) agreed that the visualization was easy to navigate and 15% strongly agreed. For the disagreed participants of 7%, the difficulties mostly come from the first interaction with the visualization and how to start using it.

- *They would like to use the visualization in the future:* Half of the participants (50%) wish to re-use the visualization in their future work and over 15% strongly agreed. Other answers include indecision with 26%, disagreeing, and strongly disagreeing with the same percentage (3.8%).

  When looking for clues in the reports to understand the reasons behind indecision and disagreement, it was mostly because of big large classes. One of the participants who analyzed Iceberg reported that some classes contained more than 120 methods with hundreds of connections between method nodes.

  Nonetheless, the most appreciated feature (mentioned in all the reports) is dead code detection. Others also found very useful the interactions with the visualization (mouse hover to see the name of the method, shift + mouse hover to see the source code of the method, the double click to open the method in the system browser, the right-click to highlight the outgoing invocations and left-click to highlight the incoming invocations). While others found the colors very useful to have an understanding of the methods inside its system and their relation with the super/sub implementations.

  Very few participants reported finding dead attributes in their projects. Furthermore, some participants reported finding some false positives in dead methods. Such false positives are often due to methods that belong to an API and that are not called in the system under analysis. Other cases were in extension[2] methods from a package that itself is not part of the analysis.

  Finally, the absence of the green test annotation in method nodes allowed users to consider reinforcing tests in these parts of the source code. Especially since tests measure the confidence that certain features are adequately implemented. Some participants reported missing tests meaning that their projects were not as expectedly well covered by tests and others added tests to cover such methods.

[2]In Pharo, a class can be extended by methods that are packaged in another package than the one of the class.

6.6 Threats to Validity

In order to ensure the credibility and reliability of the study's findings, this section addresses potential threats to validity.

Internal Validity: To what extent we can draw a causal link between the treatment in the experiment and the response? Regarding the projects, they were selected by the participants according to their previous experiences. The participants either used or developed these projects before. They considered their level of knowledge of the projects as debutants, intermediaries, advanced, and experts. The subjective nature of these classifications however introduces a potential threat to internal validity.

The level of expertise in software programming has been collected for each participant. Concretely, the expertise varies from two to thirty years. Our experiment was not dedicated to a specific audience but to diverse profiles. We want to report that we performed a first attempt to evaluate the use of the visualization to reverse engineer unknown software with internship students (3rd year). Such an attempt was unsuccessful since most of the students did not have enough concerns about quality and good object-oriented design.

The project sizes vary from 18 to 941 classes. As explained in Section 6.5.3, most participants flight over the whole or lot of classes and only plunged into the classes they found worthy of the analysis (important classes, big classes, etc).

Obviously, the answers depend on the project the participants choose and their expertise in the project or in software programming. However, the diversity of the projects and the diversity of experiences of the participants limit the threat of internal validity.

External Validity: Are our results generalizable for practice modernization? Even though the evaluation was only based on Pharo projects, the visualization also supports Java project analysis. However, for this work, we limited our choice to only Pharo projects since we have easy access to experts. We plan in future works to apply the visualization on Java projects when we have the possibility to interact with Java maintainers to better adapt the visualization or add new features according to their feedback if needed. Nonetheless, the approach itself can be applied to any object-oriented program. For other programming languages such as C++ and Python, the Pharo community does not have a parser yet.

Construct Validity: Are we asking the right questions? The answers to the questions, presented in the quantitative evaluation (Section 6.5), may depend on the projects analyzed by the participants and their expertise. Some projects do not necessarily provide elements to answer all the questions. For instance, some projects do not commonly use inheritance, consequently, methods overriding, extending their super implementations, and methods overridden in the subclasses may be absent. Additionally, some classes do not have lots of class side methods. Thus,

it may be difficult to observe the communication instance/class sides. However, we believe that the number of participants as well as the diversity of the projects and participants reduce the threats.

Reliability: To what extent can the results be reproduced when the research is repeated under the same conditions? The qualitative evaluation relies on the feedback of users according to their projects. The anonymity does not enable the reproduction of the research under the same conditions. Furthermore, we also suggest that the results of these evaluations vary with the projects and the experiences of the users even if we did our best to reduce this point by increasing the number and the diversity of users and projects.

Nevertheless, for reproducibility purposes, the tool and all the artefacts used in the evaluations are available online. With the tool, we provide full instructions on how to start using the visualization in the Readme. The legend of the visualization can also help the user through her navigation.

6.7 Conclusion

Understanding classes is important since they are the key abstractions in object-oriented programming. Object-oriented programming late binding makes understanding more difficult than procedural. In particular, there is no specific reading order that IDE could use to present information to developers.

CLASS BLUEPRINT Lanza and Ducasse [2001] proposed a compact view of class call-graph based on layers. In this chapter, we identified the limits of CLASS BLUEPRINT and proposed a new version. BLUEPRINTV2 supports dead code identification, methods under tests, and call flow between instance and class (static) methods. It enhances field understanding by showing how fields of super/subclasses are accessed, as well as lazy initialization in a compact form. It also supports hook understanding from a superclass point of view.

We presented a first validation with developers. The evaluation is twofold qualitative and quantitative. The qualitative part enabled us to highlight two complementary approaches to use the proposed visualization. The *Flight over* consists of quickly navigating the visualizations of the classes one by one, with a generalized purpose in mind for the user. This approach leads to the detection of some issues in existing classes or highlights the need for deeper analysis when needed. The *Plunge in* corresponds to this deeper analysis, like complex class or dead code detection. From a quantitative point of view, the feedbacks are mostly positive from the uses of the visualization.

For future works, we believe that new features such as annotating commented methods similar to the tested methods annotation feature would be beneficial. In addition to highlighting if the method test passes or fails. Secondly, evaluating our

visualization on projects written in Java and other object-oriented languages gives valuable insights into the potential improvements and optimizations needed for our visualization approach to be more universally adaptable.

CHAPTER 7

Conclusion

Contents

7.1 Summary

Any software is susceptible to becoming legacy software. As software ages over time, legacies are characterized by a complex structure, degraded architecture, and poor code quality. Such characteristics make the maintenance of these software systems even more difficult and time-consuming according to the literature. The task of merely reading the source code and looking for clues to make the right decisions without compromising the overall software becomes arduous.

Many tools and approaches have been proposed by researchers to deal with different maintenance challenges. A significant focus has been put on enhancing the comprehensibility of software programs by identifying specific maintenance tasks and providing appropriate solutions. A community of researchers successfully believes that visualizations can be of great help when maintaining software since they provide a visual approach to understanding software instead of sequentially reading the source code. Furthermore, such a visual approach relies on representing software artefacts using simple and familiar shapes and colours to distinguish between these artefacts whether they are entities or entities' properties , etc.

To mitigate the challenges faced by software maintainers of our industrial partners as well as the challenges already existing in the literature, this thesis provides a collection of panoramic views: three Birds Eye Views, each aiming to solve a specific maintenance challenge successfully. Each Bird Eye View is targeted to solve an instance of software violation: architectural (CLISERVO), coding standards (CLASSNAME DISTRIBUTION), and coding quality (BLUEPRINTV2).

In summary, this thesis contributes to the field of software maintenance by bridging the gap between academia and industry, addressing real-world challenges,

and providing visualizations to assist software maintainers in understanding the codebase and identifying specific instances of architectural, naming and code-based violations. The research conducted in this thesis aims to improve the effectiveness of software maintenance processes and contribute to the development of more efficient software systems.

Chapter 2. presents an overview of the literature portraying works, methodologies, and studies related to our field of topic: software maintenance and visualizations. This chapter specifically focuses on software architecture, and assessing the quality of software architecture through its code, encompassing identifier names quality. While also presenting the most influential visualizations provided by the visualizations community.

Chapter 3. presents a novel visualization dedicated to recovering client-server architecture from layers perspective, distinct viewpoints, and configurations. The main motivation underlying the conception of the CLISERVO visualization is the identification of architectural violations which consist of suboptimal package structure and unjustifiable dependencies between components of the software layers. The visualization was validated on industrial projects with the presence of their maintainers to accept/reject the collected results by the visualization experts. The validation presented satisfying results when applied to industrial projects and detected several violations which were validated by the software maintainers.

Chapter 4. presents the second visualization of our Bird-Eye Views which consist of the CLASSNAME DISTRIBUTION. A visualization which is dedicated to the detection of violations of coding standards of software naming conventions. The visualization depicts the whole project at a glimpse and uses boxes to identify the different packages, suffixes/prefixes, classes, and colours to identify each of the hierarchies implemented in the project. Coloured hierarchies display potential violations and bring the maintainers' attention to further investigations. In this chapter, we also use the visualization to detect naming patterns and anti-patterns to effectively highlight and understand naming practices in object-oriented software, particularly Pharo and Java.

Chapter 5. validating software visualizations is not an easy task. According to the literature, most software visualizations have not been effectively and efficiently validated. This chapter is dedicated to the validation of the CLASSNAME DISTRIBUTION visualization which was applied to 6 important Pharo projects with their maintainers from different countries (France, Chile, and Argentina) and a broader

range of 50 Java projects carefully selected from GitHub repositories. The validation was divided into both qualitative and quantitative evaluations from which we reported the analysis result of the selected projects, the detection of important naming violations, and finally the presence of naming patterns in almost all projects.

Chapter 6. presents our third Bird Eye View which consists of an enhancement of an important and influential visualization: the CLASS BLUEPRINT. This chapter first presents the limitations of the first version underlying the motivation behind this work. With the contribution of the main authors of the first version, we successfully enhanced the initial visualization and provided an up-to-date version (BLUEPRINTV2). BLUEPRINTV2 provides a better classification of the class methods (static/instance side) while ensuring to display the interplay between these methods. It also considers the attributes of the superclass and their relations with the class components. BLUEPRINTV2 goes further to detecting dead code and methods with high cyclomatic complexity revealing violations with regard to code practices. The visualization was also validated using both qualitative and quantitative evaluations providing a broader analysis and an in-depth analysis of the visualization application. Both evaluations were successful in detecting violations of code practices. Furthermore, the use of the visualization by the participants surprisingly revealed two instinct modes: the *Fly over* where the user continuously displays one visualization after another looking for important class visualization such as God or complex classes, and the *Plunge-in* when finally the user selects a class visualization to interact with on a deeper level.

7.2 Contributions

The contributions of this thesis can be summarized as follows:

- *A visualization for Client-Server Architecture* — a novel visualization for the detection of architectural violations validated on real industrial projects;

- A visualization for naming convention assessment, the *ClassNames Distribution* also helps in identifying naming (anti-)patterns. The visualization is validated using both qualitative and quantitative evaluations on important Pharo projects and a large set of Java projects;

- An enhancement of a prominent visualization, the *Class Blueprint* (*Best Conference Paper for VISSOFT22*);

7.3 Future Work

Here we discuss the future improvements of the work already presented in this thesis as well as future ideas as a means of extension to provide more visualizations to software maintainers.

7.3.1 Future Enhancements of the Bird-Eye Views

This section addresses some enhancements to the work presented in this thesis.

Extension of the CLISERVO validation. the validation of the CLISERVO visualization can be extended and applied to a larger set of projects when access to more maintainers is possible. This will help in enhancing the detection of the architectural violations and extend it to identifying more of the architectural violations rather than the ones detected in this thesis. For instance, identifying architectural violations in microservices applications. But first, a clear definition of what are microservices architecture violation needs to be defined.

The validation of the BLUEPRINTV2 on Java projects. applying the visualization on Java projects when the possibility to interact with Java maintainers is possible to better adapt the visualization or add new features according to their feedback if needed. For other programming languages such as C++ and Python, the Pharo community does not have a parser yet.

New features to the BLUEPRINTV2. introducing new features such as annotating commented methods similar to the tested methods annotation feature. In addition to highlighting if the method test passes or fails.

Improving ClassName Distribution in a future version of the CLASSNAME DISTRIBUTION, considering a list of exceptions specified by the domain expert that will be taken into account to avoid false positives. Additionally, the current visualization of the CLASSNAME DISTRIBUTION is based on inheritance. Therefore, for each hierarchy, we detect its root class and build the visualization accordingly. Currently, the user has the possibility to change the root class according to her needs. However, it would also be interesting to be able to provide more than one root class. As it allows the users to have more control of the depicted visualization.

7.3.2 Future Explorations of Visualization tools

Many ideas could be exploited to build visualizations. A studied and well-defined set of visualizations is already present in the Moose environment and probably

extended in the future would benefit maintainers in understanding different aspects of the software. To do so, the right questions must be asked depending on the needs of the users and their goals.

Creating a metamodel for visualization tools. and yet the process of using a metamodel for software visualization encompasses a few systematic steps such as defining the aspects of the software to visualize, identifying both software components and their relations that are taking part of the visualization. Besides defining an abstract syntax that describes each of these elements. The main key in the visualization metamodel is to be as extensible and flexible as possible to cover as many visualization needs.

A what-if visualization. the big majority of the proposed visualizations display software as they are. If a change happens to the software the user has to regenerate a new visualization of the current state of the software. A what-if visualization allows the user to simulate the change from the visualization itself which computes the different changes that happen to the software and displays it to the user directly without making the actual change to the software. This allows the user to comprehend the consequences of her change, analyze it and make decisions based on the results. A further idea would predict the potential future states of the software based on the practices and changes that happened to the software since its existence.

Real-Time collaboration. visualizations serve as visual aids for software maintainers with the responsibility of interpreting these visualizations. The interpretation is usually made in the form of a text report because most visualizations cannot be annotated. Introducing user annotations into the visualization amongst team members helps in better interpreting and studying the visualization effectively.

Visualizing microservices. just as we visualized client-server architecture in chapter 3, visualizations can be applied to different kinds of software architecture. As long as the architecture features and software rules are detectable. A visualization can show the clusters of the microservices in the software and the global architecture, meaning how they interact with the other components. The visualization can also depict information about each microservice; about the quality, coupling and cohesion, the complexity and highlight refactoring opportunities , etc.

Detecting code smells. after the collection of metrics from the literature that describe the distinct code smells, it could be interesting to have a full visualization

that depicts all the code smells and bug opportunities in large-scale software from a class code perspective.

Dynamic analysis. in this thesis, we only focused on the static analysis of software, meaning visualizing the static state of software. Few works however focus on visualizing the dynamic structure of software and computer artefacts. For instance, assessing the memory leak in the RAM by highlighting parts of the software where memory is allocated but not released. Or, the use of dashboards representing various metrics to monitor the usage of CPU. Even so, dashboards are widely used to monitor the state of software. Another example would be to visualize the data circulation in the software, meaning a visualization that could answer the questions: How does the data circulate in the program, where does it go and where does it land at last.

Visualizing hexagonal architecture. such as visualizing the set of hexagones in the software as hexagone shapes and in each shape a list of properties and their corresponding percentage represented by a colored bar (coupling, cohesion, test , etc). The color can also highlight whether there is a need to maintain the hexagone.

Combining AI. with the growing interest and new technology related to AI such as ChatGPT that generates code, a platform that generates program visualizations from a simple description.

IDE plug-in. using our visualizations requires the user to obtain a Moose image to visualize her software, create a Moose model, and import the model into the image , etc. A more easy solution for our tools to be available for external usage is to work on a plug-in with IDEs such as IntelliJ and Eclipse that can offer the same visualization of the artefact directly on the IDE.

Survey. the text discusses the enthusiasm among researchers for developing new software visualization methods. It highlights the importance of merging academic and industrial approaches to fully comprehend the broader requirements of developers and software maintainers. The current issue is that academic tools are often developed without considering real-world needs. One proposed solution to bridge this gap is conducting surveys among participants from various backgrounds and roles in labs and companies. These surveys can shed light on the actual needs and challenges faced by developers and researchers in understanding and maintaining large software programs. Despite the presence of empirical studies and surveys in existing literature, there is a need for updated and more extensive research, as current studies are either outdated or have limited participant diversity.

In conclusion, this section discussed various ideas for future improvements concerning the work presented in this thesis and proposed ideas on the broader aspect of software visualization tools. By addressing the current limitations and continuously striving to develop more user-friendly, comprehensive, and effective tools, we can contribute to enhancing the understanding and maintenance of software, ultimately benefiting both the academic and industrial communities.

Bibliography

Hani Abdeen, Stéphane Ducasse, Houari A. Sahraoui, and Ilham Alloui. Automatic package coupling and cycle minimization. In *Proceedings of the 16th International Working Conference on Reverse Engineering (WCRE'09)*, pages 103–112, Washington, DC, USA, 2009. IEEE Computer Society Press. doi: 10.1109/WCRE.2009.13.

Hani Abdeen, Stéphane Ducasse, Damien Pollet, Ilham Alloui, and Jean-Rémy Falleri. The package blueprint: Visually analyzing and quantifying packages dependencies. *Science of Computer Programming*, 89:298–319, February 2014. doi: 10.1016/j.scico.2014.02.016.

S.L. Abebe, S. Haiduc, P. Tonella, and A. Marcus. Lexicon bad smells in software. In *Working Conference on Reverse Engineering (WCRE '09)*, pages 95–99, Little, France, October 2009. doi: 10.1109/{WCRE}.2009.26.

Nour Jihene Agouf, Stéphane Ducasse, Anne Etien, Abdelghani Alidra, and Arnaud Thiefaine. Understanding Class Name Regularity: A Simple Heuristic and Supportive Visualization. *Journal of Object Technology*, 21:1312–1330, 2022a. doi: 10.5381/jot.2022.21.1.a2.

Nour Jihene Agouf, Stéphane Ducasse, Anne Etien, and Michele Lanza. A New Generation of Class Blueprint (best paper award). In *IEEE Working Conference on Software Visualization (VISSOFT)*, 2022b.

Nour Jihene Agouf, Soufyane Labsari, Stéphane Ducasse, Anne Etien, and Nicolas Anquetil. A visualization for client-server software assessement. In *IEEE Working Conference on Software Visualization (VISSOFT)*, 2023.

Sazzadul Alam and Philippe Dugerdil. Evospaces: 3d visualization of software architecture. In *SEKE*, volume 7, page 500, 2007.

Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. Suggesting accurate method and class names. In *Proceedings of the Joint Meeting on Foundations of Software Engineering*, pages 38–49. ACM, 2015.

Simon Allier, Salah Sadou, Houari Sahraoui, and Régis Fleurquin. From object-oriented applications to component-oriented applications via component-oriented architecture. In *2011 Ninth Working IEEE/IFIP Conference on Software Architecture*, pages 214–223. IEEE, 2011.

Reem S. Alsuhaibani, Christian D. Newman, Michael J. Decker, Michael L. Collard, and Jonathan I. Maletic. On the naming of methods: A survey of professional developers. In *International Conference on Software Engineering*, 2021.

Nicolas Anquetil and Timothy C. Lethbridge. Assessing the relevance of identifier names in a legacy software system. In *Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative research*, CASCON'98, pages 213–222. IBM Press, 1998. URL http://portal.acm.org/citation.cfm?id=783160.783164.

Nicolas Anquetil, Anne Etien, Mahugnon Honoré Houekpetodji, Benoît Verhaeghe, Stéphane Ducasse, Clotilde Toullec, Fatija Djareddir, Jèrôme Sudich, and Mustapha Derras. Modular moose: A new generation of software reengineering platform. In *International Conference on Software and Systems Reuse (ICSR'20)*, number 12541 in LNCS, December 2020. doi: 10.1007/978-3-030-64694-3_8.

Craig Anslow, James Noble, Stuart Marshall, and Ewan Tempero. Visualizing the word structure of java class names. In *Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems, languages and applications*, pages 777–778. ACM, 2008.

Craig Anslow, Stuart Marshall, James Noble, and Robert Biddle. Sourcevis: Collaborative software visualization for co-located environments. In *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*, pages 1–10. IEEE, 2013.

Giuliano Antoniol, Yann-Gael Gueheneuc, Ettore Merlo, and Paolo Tonella. Mining the lexicon used by programmers during sofware evolution. In *ICSM 2007: IEEE International Conference on Software Maintenance*, pages 14–23, October 2007. ISBN 978-1-4244-1256-3. doi: 10.1109/ICSM.2007.4362614.

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. ISBN 978-3-9523341-6-4.

Daniel Atzberger, Tim Cech, Merlin de La Haye, Maximilian Söchting, Willy Scheibel, Daniel Limberger, and Jürgen Döllner. Software forest: A visualization of semantic similarities in source code using a tree metaphor. In *VISIGRAPP (3: IVAPP)*, pages 112–122, 2021.

Michael Balzer, Andreas Noack, Oliver Deussen, and Claus Lewerentz. Software landscapes: Visualizing the structure of large software systems. In *IEEE TCVG*, 2004.

Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison Wesley, 1998.

Fabian Beck. Software feathers figurative visualization of software metrics. In *2014 International Conference on Information Visualization Theory and Applications (IVAPP)*, pages 5–16. IEEE, 2014.

Dirk Beyer. Co-change visualization. In *Proceedings of the 21st IEEE International Conference on Software Maintenance, Industrial and Tool volume*, ICSM'05, pages 89–92, 2005. URL http://citeseer.ist.psu.edu/beyer05cochange.html.

Dave Binkley, Dawn Lawrie, Steve Maex, and Christopher Morrell. Identifier length and limited programmer memory. *Science of Computer Programming*, 74(7):430–445, 2009.

Sandro Boccuzzo and Harald Gall. CocoViz: Towards cognitive software visualizations. *VISSOFT 2007. 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, 0:72–79, 2007. doi: 10.1109/VISSOF.2007.4290703.

Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. Relating identifier naming flaws and code quality: An empirical study. In *European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE Press, 2009.

Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. Exploring the influence of identifier names on code quality: An empirical study. In *European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE Press, 2010.

Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. Improving the tokenisation of identifier names. In *European Conference on Object-Oriented Programming (ECOOP)*. Springer, 2011a.

Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. Mining java class identifier naming conventions. In *International Conference on Software Maintenance (ICSM)*, pages 1641–1643. IEEE Press, 2011b. URL https://ieeexplore.ieee.org/document/6080776.

Pierre Caserta and Olivier Zendra. Visualization of the static aspects of software: a survey. *IEEE Transactions on Visualization and Computer Graphics*, 17(7): 913–933, 2011.

P Clements, R Kazman, and M Klein. Evaluating software architectures: methods and case studies. 2002.

Donny T Daniel, Egon Wuchner, Konstantin Sokolov, Michael Stal, and Peter Liggesmeyer. Polyptychon: A hierarchically-constrained classified dependencies visualization. In *2014 Second IEEE Working Conference on Software Visualization*, pages 83–86. IEEE, 2014.

Florian Deissenboeck and Markus Pizka. Concise and consistent naming. *Software Quality Journal*, 14(3):261–282, 2006.

L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Proceedings POPL '84*, Salt Lake City, Utah, January 1984. doi: 10.1145/800017.800542. URL http://webpages.charter.net/allanms/popl84.pdf.

Karim Dhambri, Houari Sahraoui, and Pierre Poulin. Visual detection of design anomalies. In *2008 12th European Conference on Software Maintenance and Reengineering*, pages 279–283. IEEE, 2008.

Martin Dias, Diego Orellana, Santiago Vidal, Leonel Merino, and Alexandre Bergel. Evaluating a visual approach for understanding javascript source code. In *2020 IEEE/ACM 28th International Conference on Program Comprehension (ICPC)*, 2020.

Stephan Diehl, editor. *Software Visualization*. Springer, 2002.

Stephan Diehl. *Software Visualization*. Springer-Verlag, Berlin Heidelberg, 2007a. ISBN 978-3-540-46504-1.

Stephan Diehl. *Software visualization: visualizing the structure, behaviour, and evolution of software*. Springer Science & Business Media, 2007b.

Stéphane Ducasse and Michele Lanza. The Class Blueprint: Visually supporting the understanding of classes. *Transactions on Software Engineering (TSE)*, 31 (1):75–90, January 2005. doi: 10.1109/TSE.2005.14.

Stéphane Ducasse and Damien Pollet. Software architecture reconstruction: A process-oriented taxonomy. *IEEE Transactions on Software Engineering*, 35(4): 573–591, July 2009a. doi: 10.1109/TSE.2009.19.

Stéphane Ducasse and Damien Pollet. Software architecture reconstruction: A process-oriented taxonomy. *IEEE Transactions on Software Engineering*, pages 573–591, 2009b.

Stéphane Ducasse, Tudor Gîrba, and Adrian Kuhn. Distribution map. In *Proceedings of 22nd IEEE International Conference on Software Maintenance*, ICSM'06, pages 203–212, Los Alamitos CA, 2006. IEEE Computer Society. doi: 10.1109/ICSM.2006.22.

Stéphane Ducasse, Nicolas Anquetil, Usman Bhatti, Andre Cavalcante Hora, Jannik Laval, and Tudor Girba. MSE and FAMIX 3.0: an Interexchange Format and Source Code Model Family. Technical report, RMod – INRIA Lille-Nord Europe, 2011.

Sebastian Eder, Maximilian Junker, Elmar Jürgens, Benedikt Hauptmann, Rudolf Vaas, and Karl-Heinz Prommer. How much does unused code matter for maintenance? In *2012 34th International Conference on Software Engineering (ICSE)*, pages 1102–1111. IEEE, 2012.

Alexander Egyed and Phillipe B. Kruchten. Rose/architect: a tool to visualize architecture. In *Proc. 32nd Annual Hawaii Conference on Systems Sciences*, 1999.

Stephen Eick, Todd Graves, Alan Karr, Audris Mockus, and Paul Schuster. Visualizing software changes. *IEEE Transactions on Software Engineering*, 28(4): 396–412, 2002.

Ural Erdemir, Umut Tekin, and Feza Buzluca. E-quality: A graph based object oriented software quality visualization tool. In *2011 6th International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*, pages 1–8. IEEE, 2011.

Johan Fabry and Stéphane Ducasse. *The Spec UI Framework*. Square Bracket Associates, 2017. URL http://books.pharo.org.

Amin Milani Fard and Ali Mesbah. Jsnose: Detecting javascript code smells. In *2013 IEEE 13th international working conference on Source Code Analysis and Manipulation (SCAM)*, pages 116–125. IEEE, 2013.

Michael C. Feathers. *Working Effectively with Legacy Code*. Prentice Hall, 2005. ISBN 0-13-117705-2.

I. Fernandez, A. Bergel andJ. P. S. Alcocer, A. Infante, and T. Gîrba. Glyph-based software component identification. In *International Conference on Program Comprehension (ICPC)*, pages 1–10, 2016a.

Ignacio Fernandez, Alexandre Bergel, Juan Pablo Sandoval Alcocer, Alejandro Infante, and Tudor Gîrba. Glyph-based software component identification. In *Proceedings of the 24th IEEE International Conference on Program Comprehension (ICPC '16)*, 2016b.

P. Finnigan, R. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. Mueller, J. Mylopoulos, S. Perelgut, M. Stanley, and K. Wong. The software bookshelf. *IBM Systems*

Journal, 36(4):564–593, November 1997. URL http://researchweb.watson.ibm.com/journal/sj/364/finnigan.htmlhttp://researchweb.watson.ibm.com/journal/sj/364/finnigan.pdf.

Florian Fittkau, Alexander Krause, and Wilhelm Hasselbring. Exploring software cities in virtual reality. In *2015 ieee 3rd working conference on software visualization (vissoft)*, pages 130–134. IEEE, 2015.

Francesca Arcelli Fontana, Ilaria Pigazzini, Riccardo Roveda, Damian Tamburri, Marco Zanoni, and Elisabetta Di Nitto. Arcan: A tool for architectural smells detection. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 282–285. IEEE, 2017.

Jon Froehlich and Paul Dourish. Unifying artifacts and activities in a visual tool for distributed software development teams. In *Proceedings of the 26th International Conference on Software Engineering*, pages 387–396, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2163-0.

Keith Gallagher, Andrew Hatch, and Malcolm Munro. Software architecture visualization: An evaluation framework and its application. *IEEE Transactions on Software Engineering*, 34(2):260–270, 2008.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

Yaser Ghanam and Sheelagh Carpendale. A survey paper on software architecture visualization. *University of Calgary, Tech. Rep*, page 17, 2008.

Adele Goldberg. *Smalltalk 80: the Interactive Programming Environment*. Addison Wesley, Reading, Mass., 1984. ISBN 0-201-11372-4.

Giona Granchelli, Mario Cardarelli, Paolo Di Francesco, Ivano Malavolta, Ludovico Iovino, and Amleto Di Salle. Microart: A software architecture recovery tool for maintaining microservice-based systems. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 298–302. IEEE, 2017.

M David Hanson. The client/server architecture. In *Server Management*, pages 17–28. Auerbach Publications, 2000.

C. G. Healey, K. S. Booth, and J. T. Enns. Harnessing preattentive processes for multivariate data visualization. In *GI '93: Proceedings of Graphics Interface*, 1993.

Brian Henderson-Sellers. *Object-oriented metrics: measures of complexity*. Prentice-Hall, Inc., 1995.

Yosuke Isobe and Haruaki Tamada. Are identifier renaming methods secure? In *2018 19th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, pages 322–328. IEEE, 2018.

Rick Kazman and S Jeromy Carriere. View extraction and view fusion in architectural understanding. In *Proceedings. Fifth International Conference on Software Reuse (Cat. No. 98TB100203)*, pages 290–299. IEEE, 1998.

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

Claire Knight and Malcolm Munro. Visualising software-a key research area. In *Proceedings of the IEEE International Conference on Software Maintenance*, page 437, 1999.

Kenichi Kobayashi, Manabu Kamimura, Keisuke Yano, Koki Kato, and Akihiko Matsuo. Sarf map: Visualizing software architecture from feature and layer viewpoints. In *2013 21st International Conference on Program Comprehension (ICPC)*, pages 43–52. IEEE, 2013.

Rainer Koschke. Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey. *Journal of Software Maintenance and Evolution: Research and Practice*, 15(2):87–109, 2003. ISSN 1040-550X. doi: 10.1002/smr.270.

Rainer Koschke and Daniel Simon. Hierarchical reflexion models. In *Working Conference on Reverse Engineering*, page 36. IEEE Computer Society, 2003.

Philippe B. Kruchten. The 4+1 view model of architecture. *IEEE Software*, pages 42–50, 1995.

Arun Lakhotia. Understanding someone else's code: Analysis of experiences. *J. Syst. Softw.*, 23(3):269–275, 1993.

Wilf LaLonde and John Pugh. Subclassing $\neq$ Subtyping $\neq$ Is-a. *Journal of Object-Oriented Programming*, 3(5):57–62, January 1991. URL http://scgresources.unibe.ch/~scg/Literature/PL/LaLo91a-JOOP0305.pdf.

Guillaume Langelier, Houari A. Sahraoui, and Pierre Poulin. Visualization-based analysis of quality for large-scale software systems. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 214–223, New York, NY, USA, 2005. ACM. ISBN 1-59593-993-4. doi: 10.1145/1101908.1101941. URL http://dx.doi.org/10.1145/1101908.1101941.

Michele Lanza. The evolution matrix: Recovering software evolution using software visualization techniques. In *Proceedings of the International Workshop on Principles of Software Evolution*, IWPSE'01, pages 37–42, 2001. doi: 10.1145/602461.602467. URL http://scg.unibe.ch/archive/papers/Lanz01cEvolutionMatrix.pdf.

Michele Lanza. *Object-Oriented Reverse Engineering — Coarse-grained, Fine-grained, and Evolutionary Software Visualization*. PhD thesis, University of Bern, May 2003. URL http://scg.unibe.ch/archive/phd/lanza-phd.pdf.

Michele Lanza and Stéphane Ducasse. A Categorization of Classes based on the Visualization of their Internal Structure: the Class Blueprint. In *Proceedings of 16th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '01)*, pages 300–311. ACM Press, 2001. doi: 10.1145/504282.504304.

Michele Lanza and Stéphane Ducasse. Polymetric views—a lightweight visual approach to reverse engineering. *Transactions on Software Engineering (TSE)*, 29(9):782–795, September 2003. doi: 10.1109/TSE.2003.1232284.

Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice*. Springer-Verlag, 2006. ISBN 3-540-24429-8. URL http://www.springer.com/alert/urltracking.do?id=5907042.

Thomas D LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th international conference on Software engineering*, pages 492–501, 2006.

Manny Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, September 1980.

Manny Lehman. Laws of software evolution revisited. In *European Workshop on Software Process Technology*, pages 108–124, Berlin, 1996. Springer.

Meir M Lehman and Juan F Ramil. Rules and tools for software evolution planning and management. *Annals of software engineering*, 11:15–44, 2001.

Yi Li, Shaohua Wang, and Tien N Nguyen. A context-based automated approach for method name consistency checking and suggestion. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 574–586. IEEE, 2021.

B. Liblit, A.Begel, and E. Sweetser. Cognitive perspectives on the role of naming in computer programs. In *Annual Psychology of Programming Workshop*, 2006.

Martin Lippert and Stephen Roock. *Refactoring in large software projects: performing complex restructurings successfully*. John Wiley & Sons, 2006.

David C Littman, Jeannine Pinto, Stanley Letovsky, and Elliot Soloway. Mental models and software maintenance. *Journal of Systems and Software*, 7(4):341–355, 1987.

Kui Liu, Dongsun Kim, Tegawende F. Bissyande andTaeyoung Kim, Kisub Kim, Anil Koyuncu andSuntae Kim, and Yves Le Traon. Learning to spot and refactor inconsistent method names. In *Proceedings of ICSE'19*, 2019.

Chung-Horng Lung and Kalai Kalaichelvan. An approach to quantitative software architecture sensitivity analysis. *International Journal of Software Engineering and Knowledge Engineering*, 10(01):97–114, 2000a.

Chung-Horng Lung and Kalai Kalaichelvan. An approach to quantitative software architecture sensitivity analysis. *International Journal of SE and Knowledge Engineering*, pages 97–114, 2000b.

Walid Maalej, Rebecca Tiarks, Tobias Roehm, and Rainer Koschke. On the comprehension of program comprehension. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(4):1–37, 2014.

Isela Macia, Joshua Garcia, Daniel Popescu, Alessandro Garcia, Nenad Medvidovic, and Arndt von Staa. Are automatically-detected code anomalies relevant to architectural modularity? an exploratory analysis of evolving systems. In *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*, pages 167–178, 2012.

Mika Mantyla, Jari Vanhanen, and Casper Lassenius. A taxonomy and an initial empirical study of bad smells in code. In *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.*, pages 381–384. IEEE, 2003.

Andrian Marcus, Louis Feng, and Jonathan I. Maletic. 3D representations for software visualization. In *Proceedings of the ACM Symposium on Software Visualization*, pages 27–ff. IEEE, 2003.

Robert C. Martin. Design principles and design patterns, 2000. www.objectmentor.com.

Robert C. Martin. *Agile Software Development: principles, patterns and practices*. Prentice-Hall, 2003.

Robert C Martin. *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

Nenad Medvidovic, Alexander Egyed, and Paul Gruenbacher. Stemming architectural erosion by architectural discovery and recovery. In *Proceedings of the 2nd Second International Workshop from Software Requirements to Architectures (STRAW)*, 2003a.

Nenad Medvidovic, Alexander Egyed, and Paul Gruenbacher. Stemming architectural erosion by architectural discovery and recovery. In *Proceedings of the 2nd Second International Workshop from Software Requirements to Architectures*, 2003b.

Leonel Merino, Mohammad Ghafari, Craig Anslow, and Oscar Nierstrasz. Cityvr: Gameful software visualization. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 633–637. IEEE, 2017.

Leonel Merino, Mohammad Ghafari, Craig Anslow, and Oscar Nierstrasz. A systematic literature review of software visualization evaluation. *Journal of systems and software*, 144:165–180, 2018.

Leonel Merino, Ekaterina Kozlova, Oscar Nierstrasz, and Daniel Weiskopf. VISON: An ontology-based approach for software visualization tool discoverability. In *VISSOFT'19: Proceedings of the 7th IEEE Working Conference on Software Visualization*. IEEE, 2019. doi: 10.1109/VISSOFT.2019.00014. URL http://scg.unibe.ch/archive/papers/Meri19b-vison.pdf.

Eliot Miranda. Brouhaha — A portable Smalltalk interpreter. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 354–365, December 1987. doi: 10.1145/38765.38839.

Eliot Miranda. The cog smalltalk virtual machine. In *Proceedings of VMIL 2011*, 2011.

Ran Mo, Yuanfang Cai, Rick Kazman, and Lu Xiao. Hotspot patterns: The formal definition and automatic detection of architecture smells. In *2015 12th Working IEEE/IFIP Conference on Software Architecture*, pages 51–60. IEEE, 2015.

Son Nguyen, Hung Phan, Trinh Le, and Tien N Nguyen. Suggesting natural method names to check name consistencies. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 1372–1384, 2020.

David Lorge Parnas. Software aging. In *Proceedings 16th International Conference on Software Engineering (ICSE '94)*, pages 279–287, Los Alamitos CA, 1994. IEEE Computer Society.

Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992a. URL http://www.bell-labs.com/user/dep/work/papers/swa-sen.ps.

Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17:40–52, 1992b. URL http://www.bell-labs.com/user/dep/work/papers/swa-sen.ps.

Dwight J Peterson and Marian E Berryhill. The gestalt principle of similarity benefits visual working memory. *Psychonomic bulletin & review*, 20(6):1282–1289, 2013.

Federico Pfahler, Roberto Minelli, Csaba Nagy, and Michele Lanza. Visualizing evolving software cities. In *2020 Working Conference on Software Visualization (VISSOFT)*, pages 22–26. IEEE, 2020.

T. Pigoski. *Practical Software Maintenance. Best Practices Managing your Software Investment.* John Wiley and Sons, 1997.

Marcus Randevik and Patrik Olson. Secarchunit extending archunit to support validation of security architectural constraints. Master thesis, University OF Gothenburg, 2020.

Dennie Reniers, Lucian Voinea, and Alexandru Telea. Visual exploration of program structure, dependencies and metrics with solidsx. In *2011 6th International workshop on visualizing software for understanding and analysis (VISSOFT)*, pages 1–4. IEEE, 2011.

rigi design-recovery. Rigi home page. URL http://www.rigi.csc.uvic.ca/. http://www.rigi.csc.uvic.ca/.

Tobias Roehm, Rebecca Tiarks, Rainer Koschke, and Walid Maalej. How do professional developers comprehend software? In *2012 34th International Conference on Software Engineering (ICSE)*, pages 255–265. IEEE, 2012.

Ganesh Samarthyam, Girish Suryanarayana, and Tushar Sharma. Refactoring for software architecture smells. In *Proceedings of the 1st International Workshop on Software Refactoring*, pages 1–4, 2016.

Marcelo Schmitt Laser, Nenad Medvidovic, Duc Minh Le, and Joshua Garcia. Arcade: an extensible workbench for architecture recovery, change, and decay evaluation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1546–1550, 2020.

Alan Shalloway and James R Trott. *Design patterns explained: A new perspective on object-oriented design, 2/E*. Pearson Education India, 2005.

L Phleeger Shari et al. Software engineering: Theory and practice. *International Edition, Prentice Hall, Inc*, 1998.

Martin Shepperd. A critique of cyclomatic complexity as a software metric. *Software Engineering Journal*, 3(2):30–36, 1988.

Jeremy Singer and Chris Kirkham. Exploiting the correspondence between micro patterns and class names. In *International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2008.

J. Smith. *Object-Oriented Programming: Understanding Classes for Software Development*. ABC Publications, 2020.

Harry M Sneed. Object-oriented cobol recycling. In *Proceedings of WCRE'96: 4rd Working Conference on Reverse Engineering*, pages 169–178. IEEE, 1996.

Iain Sommerville and Peter Sawyer. *Requirements engineering: a good practice guide*. John Wiley and Sons, Inc., 1997.

Ian Sommerville. *Software Engineering*. Addison Wesley, sixth edition, 2000.

Robert Spence. *Information Visualization*. Adisson-Wesley, 2001.

John T. Stasko, John Domingue, Marc H. Brown, and Blaine A. Price. *Software Visualization — Programming as a Multimedia Experience*. The MIT Press, 1998.

Margaret-Anne D. Storey, F. David Fracchia, and Hausi A. Müller. Cognitive design elements to support the construction of a mental model during software exploration. *Journal of Software Systems*, 44:171–185, 1999.

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 *SoftVis'05: Proceedings of the 2005 ACM symposium on software visualization*, pages 193–202. ACM Press, 2005. ISBN 1595930736. doi: 10.1145/1056018.1056045. URL http://portal.acm.org/citation.cfm?id=1056018.1056045.

SNiFF+. TakeFive Software GmbH, 1996.

Hagen Tamer, Daniel van den Bongard, and Fabian Beck. Visually analyzing the structure and code quality of component-based web applications. In *2021 Working Conference on Software Visualization (VISSOFT)*, pages 160–164. IEEE, 2021.

Ewan Tempero, James Noble, and Hayden Melton. How do java programs use inheritance? an empirical study of inheritance in java software. In *ECOOP '08: Proceedings of the 22nd European conference on Object-Oriented Programming*, pages 667–691, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-70591-8.

Pablo Tesone, Stéphane Ducasse, Guillermo Polito, Luc Fabresse, and Noury Bouraqadi. A new modular implementation for stateful traits. *Science of Computer Programming*, 195:1–37, 2020. doi: 10.1016/j.scico.2020.102470.

Anne Treisman. Preattentive processing in vision. *Computer Vision, Graphics, and Image Processing*, 31(2):156–177, 1985. doi: 10.1016/S0734-189X(85)80004-9.

Benoît Verhaeghe, Anas Shatnawi, Abderrahmane Seriai, Nicolas Anquetil, Anne Etien, Stéphane Ducasse, and Mustapha Derras. Migrating GUI behavior: from GWT to Angular. In *International Conference on Software Maintenance and Evolution*, Luxembourg, 2021. URL https://hal.archives-ouvertes.fr/hal-03341866.

Lucian Voinea, Alex Telea, and Jarke J. van Wijk. Cvsscan: visualization of code evolution. In *SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization*, pages 47–56, New York, NY, USA, 2005. ACM. ISBN 1-59593-073-6. doi: 10.1145/1056018.1056025.

Tatiana von Landesberger, Arjan Kuijper, Tobias Schreck, Jörn Kohlhammer, Jarke J. van Wijk, Jean-Daniel Fekete, and Dieter W. Fellner. Visual analysis of large graphs: State-of-the-art and future research challenges. *Comput. Graph. Forum*, 30(6):1719–1749, 2011.

Anneliese von Mayrhauser and A. Marie Vans. Program comprehension during software maintenance and evolution. *IEEE Computer*, 28(8):44–55, 1995.

Daniel Votipka, Seth M Rabin, Kristopher Micinski, Jeffrey S Foster, and Michelle M Mazurek. An observational investigation of reverse engineers' processes. In *Proceedings of the 29th USENIX Conference on Security Symposium*, pages 1875–1892, 2020.

Colin Ware. *Information visualization: perception for design*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000. ISBN 1-55860-511-8.

Colin Ware. *Information Visualisation*. Elsevier, Sansome Street, San Fransico, 2004. ISBN 1-55860-819-2.

A. Watson and T. McCabe. Structured testing: A testing methodology using the cyclomatic complexity metric. Technical report, National Institute of Standards and Technology, Washington, D.C., 1996.

Richard Wettel and Michele Lanza. Visualizing software systems as cities. In *Proceedings of VISSOFT 2007 (4th IEEE International Workshop on Visualizing Software For Understanding and Analysis)*, pages 92–99, 2007. ISBN 1-4244-0599-8. doi: 10.1109/VISSOF.2007.4290706. URL http://dx.doi.org/10.1109/VISSOF.2007.4290706.

Richard Wettel and Michele Lanza. Codecity: 3d visualization of large-scale software. In *Companion of the 30th international conference on Software engineering*, pages 921–922, 2008.

Rebecca Wirfs-Brock and Alan McKean. *Object Design — Roles, Responsibilities and Collaborations*. Addison-Wesley, 2003. ISBN 0-201-37943-0.

Xinrong Xie, Denys Poshyvanyk, and Andrian Marcus. Visualization of CVS repository information. In *WCRE'06: Proceedings of the 13th Working Conference on Reverse Engineering*, pages 231–242, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2719-1. doi: 10.1109/{WCRE}.2006.55.

Aiko Yamashita and Leon Moonen. Do developers care about code smells? an exploratory survey. In *2013 20th working conference on reverse engineering (WCRE)*, pages 242–251. IEEE, 2013.

Keisuke Yano and Akihiko Matsuo. Labeling feature-oriented software clusters for software visualization application. In *2015 Asia-Pacific Software Engineering Conference (APSEC)*, pages 354–361. IEEE, 2015.

A. Zaidman, T. Calders, S. Demeyer, and J. Paredaens. Applying webmining techniques to execution traces to support the program comprehension process. In *Proceedings IEEE European Conference on Software Maintenance and Reengineering (CSMR'05)*, pages 134–142, Los Alamitos CA, 2005. IEEE Computer Society Press.

Huaxi Zhang, Christelle Urtado, and Sylvain Vauttier. Architecture-centric component-based development needs a three-level adl. In *Software Architecture: 4th European Conference, ECSA 2010, Copenhagen, Denmark, August 23-26, 2010. Proceedings 4*, pages 295–310. Springer, 2010.