JOT
JOURNAL OF
OBJECT TECHNOLOGY

Understanding Class Name Regularity: A Simple Heuristic and Supportive Visualization

Nour Jihene Agouf[‡*], Stéphane Ducasse[*], Anne Etien[†], Abdelghani Alidra , and Arnaud Thiefaine[‡]
[†]Univ. Lille, CNRS, Inria, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France
[*]Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France
[‡]Arolla, France

ABSTRACT Studies have shown that more than 50% of software maintenance time is spent reading code to understand it. This puts a strong emphasis on the understandability of source code. Class names constitute one of the first pieces of information developers have access to. **Proposal:** To assist developers in understanding the logic and regularity of class names, we present a new and simple 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. **Validation:** We show (1) how this visualization helps capture recurrent patterns relative to concept reference in class names and (2) that this visualization supports the evolution of software systems by monitoring and guiding class renamings over multiple versions. To evaluate our approach we did a consequent assessment with real practitioners 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 applied our tool to the the software they develop or maintain. They proposed and performed respectively 91, 68, and 24 class renamings. In the second setup, as authors of the visualization and the tool (*visualization experts*), we applied our tool to a new UI framework for Pharo. We sent 34 pull requests for renaming classes and 32 were accepted. Finally, we applied our visualizations to 50 Java projects and identified visual patterns in most of them. Consequently, it shows that the proposed visualization is effective for spotting class name inconsistencies, and this by both developers of the system and external persons.

The visualization presented in this article has been designed with colors, therefore the paper should be printed using an adequate medium or be read digitally.

KEYWORDS class name, visualization, program comprehension

1. Introduction

Quickly grasping the overall purpose of a source code abstraction such as a class is a key concern for plain forward development but also very important during maintenance tasks (Demeyer et al. 2002; Antoniol et al. 2007; Newman et al. 2017; Butler et al. 2009; Osman et al. 2012). The name of a class is one of the first pieces of information developers have access to.

Many researchers have analyzed program identifiers (Liblit et al. 2006; Abebe et al. 2009; Falleri et al. 2010; Alsuhaibani et al. 2021) and some have focused on the analysis of class names (Singer & Kirkham 2008; Butler et al. 2011b). For instance, Osman *et al.* show through a survey that involved 32 developers, that programmers estimate that good class names are among the most important elements in class diagrams and good class names improve comprehension (Osman et al. 2012).

In contrast, when a class is badly named, a developer may have to check carefully its definition and analyze how it is

JOT reference format:
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. Vol. 21, No. 1, 2022.
http://dx.doi.org/10.5381/jot.2022.21.1.a2

related to its superclass. This often happens when using subclassing instead of subtyping (LaLonde & Pugh 1991). For example, quickly understanding whether an abstraction is a model or a view in an MVC triad is key to avoiding mistakes or misinterpretations. Butler (Butler et al. 2009) shows that bad identifiers affect code quality and are correlated to bugs.

Assessing whether a class is correctly named is a challenge because class names may have been chosen for various reasons such as refining a general concept, inclusion in a different project, or adhering to a different naming convention. Ideally, a class name should convey the concept and the role that the class implements. In practice, however, classes get named inconsistently because of language or cultural issues, lack of conventions, violations of conventions, lack of tools to spot irregularities, concept refinement, and overly long names (see Section 2).

Helping understanding names in software is not new. Lawrie *et al.* (Lawrie et al. 2007, 2006) carried out an empirical study to assess the quality of source code identifiers. Their study involved 100 programmers and indicated that full words, as well as recognizable abbreviations, lead to better comprehension. More recently Liu *et al.* (Liu et al. 2019) proposed to use machine learning to spot and refactor inconsistent method names. Butler *et al.* (Butler et al. 2011b) worked on identifying class name conventions and as such is close to our work. Yano *et al.* (Yano & Matsuo 2015) presented a hierarchical solution to label clusters in visualizations, using lemmatization of words to describe each cluster. Their approach does not support the understanding of class name inconsistencies.

Some simple tools already exist (*e.g.,* Sonar[1] or PMD[2]) that check conventions such as: camel case convention, utility class names not ending with "Util", etc. but they are limited to checking generic regular expressions, considering only a given class and not its context (inheritance tree, package).

It may be possible to implement new rules such as the ones defined in Section 2.6 to check class name inconsistencies. However, adopting the right level of local/global perspective and thus determining which force between inheritance or package dominates may be difficult in practice.

Consequently, we tackle this problem by offering developers a new visualization, named *ClassName Distribution*, that helps them identify suspicious patterns and class name irregularities. A visualization can offer both a local view inside a package, while providing a global view, all along the hierarchies inside the whole project. Moreover, a visualization helps one to detect inconsistencies but without posing a definitive judgment as a rule violation. The proposed visualization helps to answer the following questions:

- What are the irregularities in terms of class naming over a hierarchy?
- What are the irregularities in terms of class naming inside a package?
- How can the study of class naming help to identify a new concept emerging?
- Why do two classes in separate hierarchies have similar names?

The article's contributions are (1) the identification of challenges to support the understanding of class names, (2) the definition of a simple visualization to capture regularities of class names, (3) the identification of visual patterns, and (4) a large validation on real cases with software developers of multiple projects.

The outline of the article is as follows: Section 2 analyzes the challenges of understanding class name regularity. Section 3 presents the ClassName Distribution visualization as well as (Section 3.4) patterns to better identify class name irregularities. Sections 4 and 5 present our case studies, *i.e.,* real-world projects in Pharo and Java with several hundreds of classes. Section 6 presents how the visualization helps developers to understand and steer the renaming of Calypso, a large system that has evolved over several releases. Section 7 presents a double evaluation: *Domain Expert / visualization Learners* and *Non-Domain Expert / visualization Experts*, while Section 8 reports the occurrences of visual patterns in several Java open-source projects. Section 9 discusses the possible design variations and lists the threats to validity. Section 10 stresses the comparison with selected related work. In addition, in the appendix we added two sections: Section A describes the tool supporting the approach (this is the tool user interface that the developers used during the evaluation), while Section B describes the algorithms used to precisely produce the visualization.

2. 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 (Liblit et al. 2006; Singer & Kirkham 2008; Butler et al. 2011b). 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's naming convention (Deissenboeck & 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.

2.1. 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

[1] rules.sonarsource.com
[2] pmd.github.io

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 Tool-DependencyAnalyser-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 TreeNodePresenter 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 DAPackageTreePresenter 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.

2.2. About Correct Class Names

In object-oriented languages, classes should have one responsibility (Wirfs-Brock & 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.

2.3. 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 (Martin 2000; Abdeen et al. 2009; Lanza & Marinescu 2006). 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 & 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 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 paper, we focus on the code written in English and consequently adopting such a naming convention.

2.4. 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 Section 8).

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.

2.5. Our Definition of Class Name Consistency

In this article, 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 & 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. We *exclusively* focus on class names (not method identifier (Anquetil & Lethbridge 1998), not method body vocabulary (Antoniol et al. 2007)). 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.

2.6. 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 paper, 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.

2.7. A Schematic Project

Before introducing concepts useful to the detection of class name inconsistency, let's consider the hypothetical project depicted in Figure 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.

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.

2.8. 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 1.

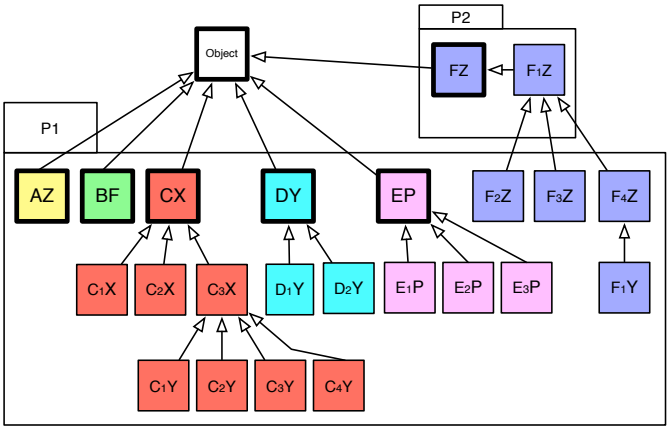

Figure 1 A schematic mini project composed of the A, B, C, D, E, and F hierarchies (thick borders denote hierarchy roots).

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.

3. The ClassName Distribution

The ClassName Distribution[3] 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.

3.1. 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 the design principles of Lanza's visualizations such as system complexity and evolution matrix (Lanza 2001; Lanza & Ducasse 2003; Lanza 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 & 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 3.2) then its color assignment (see Section 3.3).

3.2. 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 2 represents the ClassName Distribution for our hypothetical project shown in Figure 1 and Figure 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 3.3), there is

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

one color by inheritance hierarchy. Here, the colors match the ones in Figure 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 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 B).

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 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 3), or focus on a particular package. Package boxes are displayed in decreasing order of size (in number of classes).

3.3. 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:

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 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.

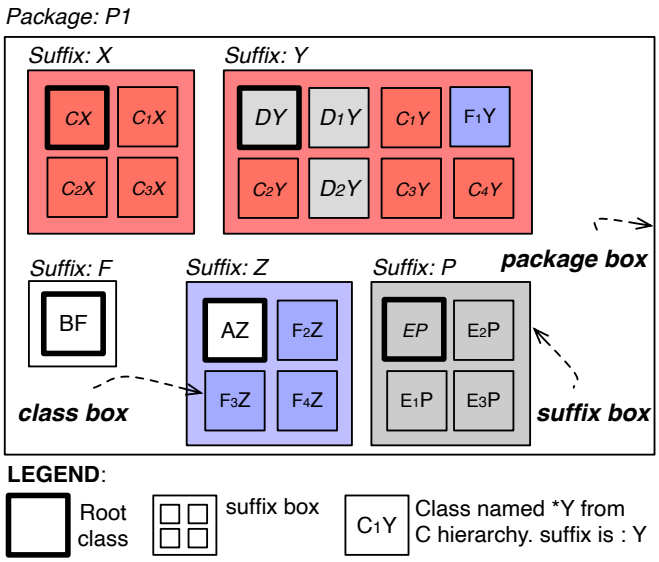

Figure 2 ClassName Distribution for package P1 of Figure 1: 1 package box, 5 suffix boxes, and 21 class boxes.

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 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 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 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.

3.4. 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 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 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.

4. 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 6.

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.

4.1. Calypso Hierarchies Analysis

This section analyzes Calypso hierarchies. Figure 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.

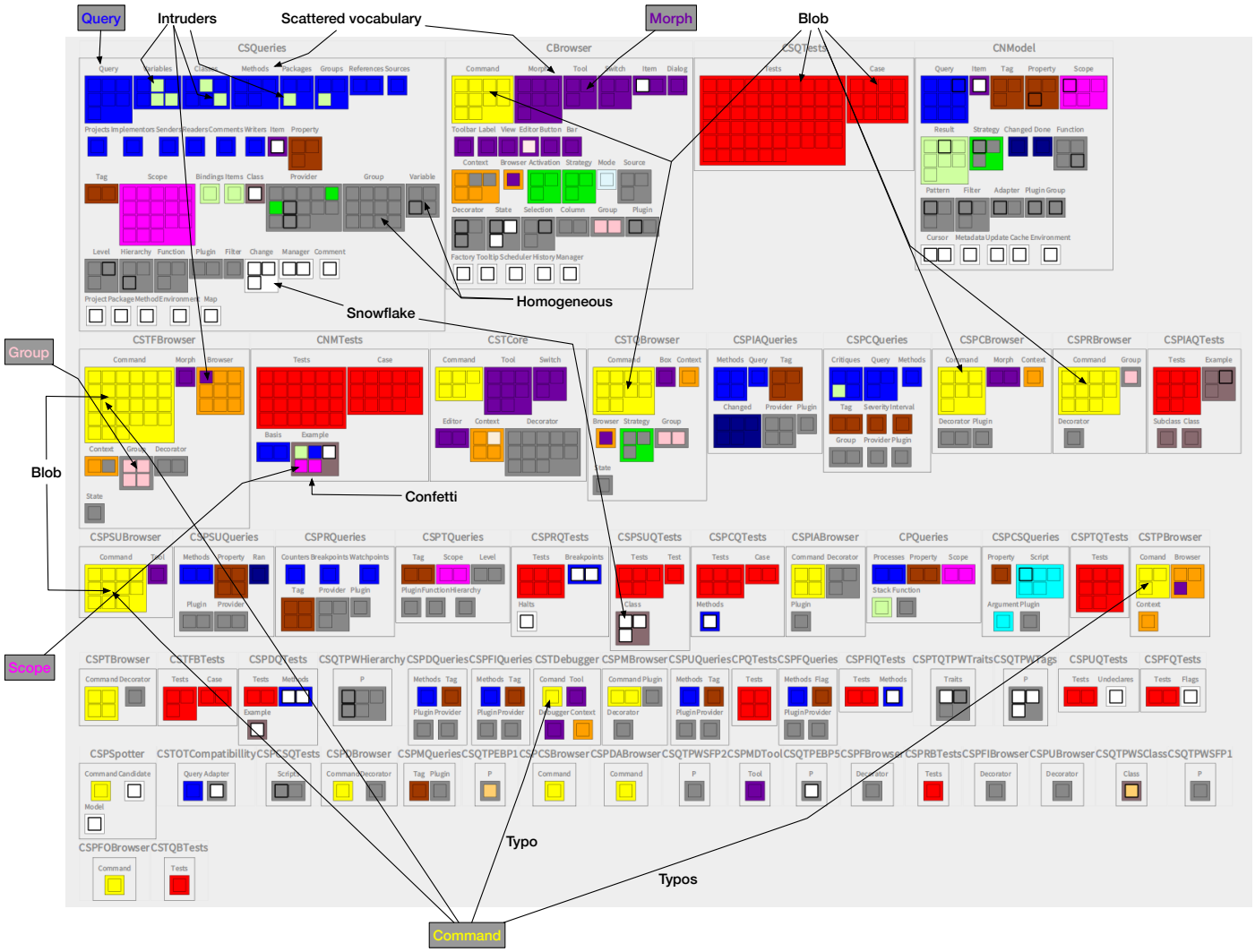

Figure 3 Visual patterns & hierarchies in ClassName Distribution of the Calypso project (v6) main packages.

Large hierarchies. A color identifies a hierarchy with inconsistent naming (remember that consistent hierarchies are in gray). Figure 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 A). In addition, the high quality of the screen resolution supports the crisp reading of names.

- The **red** hierarchy contains the SUnit[4] 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).
- 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[5]) 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 col-

[4] SUnit is the test framework in Pharo

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

ored 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 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.

4.2. Calypso Visual Pattern Analysis

We illustrate the visual patterns with Calypso. Figure 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 9 and puts the suspicious cases in a thicker white border. This is a way to detect misspellings.

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.

5. 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 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 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[6]. 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 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

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

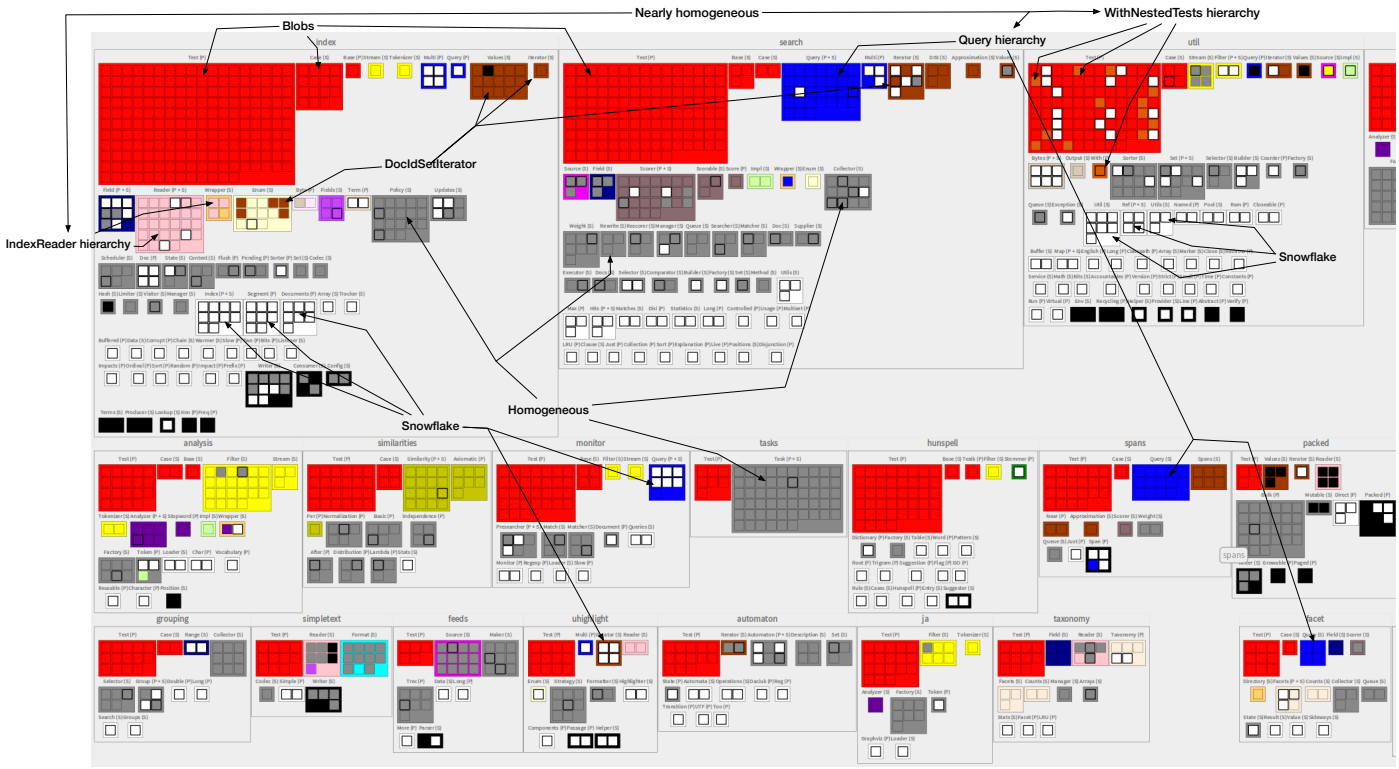

Figure 4 The ClassName Distribution of the Lucene project version of June 2021(Extract).

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: MultiTermQueryConstantScoreWrapper, 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 WithNestedTests 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.

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

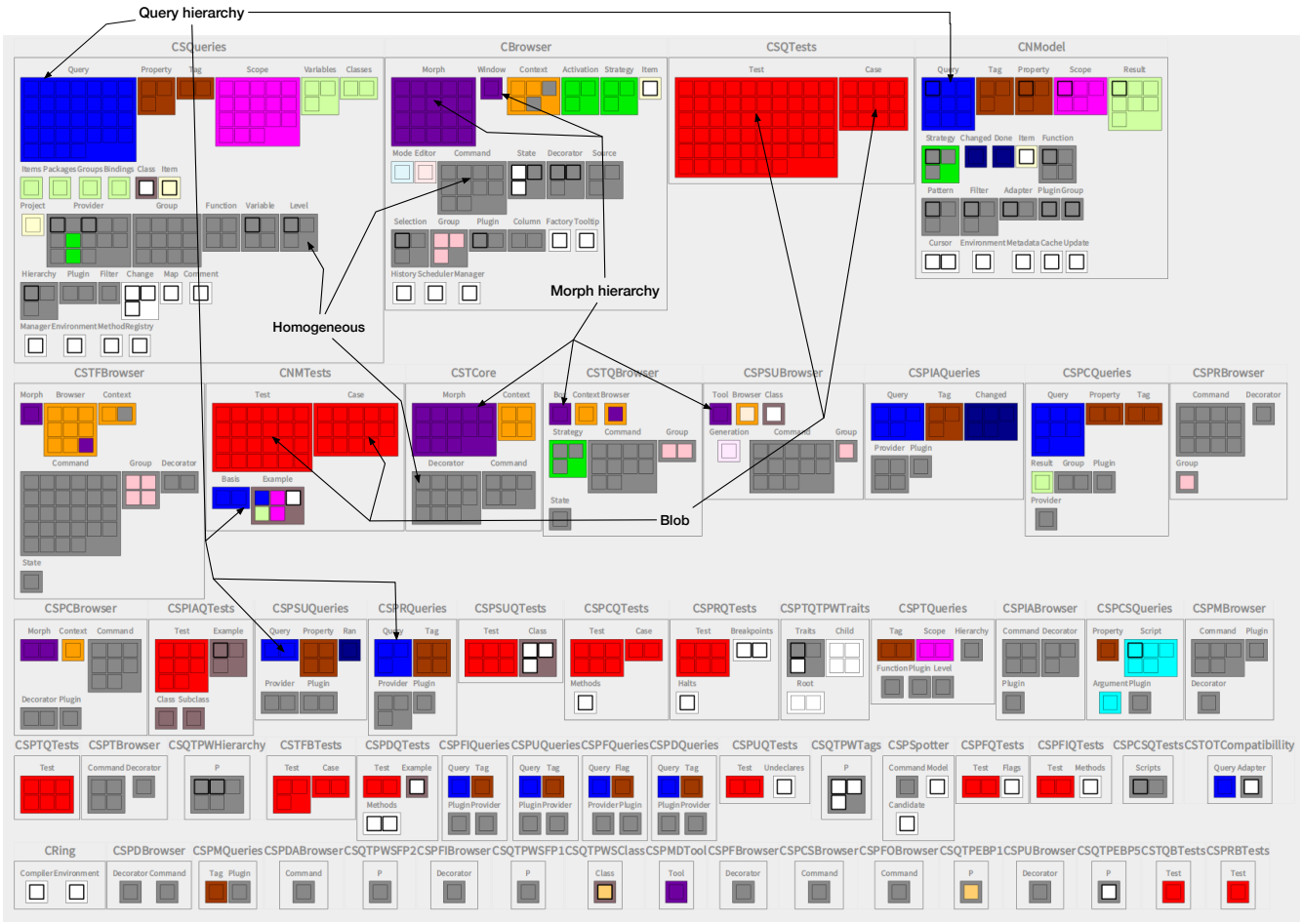

Figure 5 The ClassName Distribution of the Calypso project (v8) main packages.

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.

More could be said about this part of the project but a complete analysis of this part or the project is out of the scope of the paper.

6. 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 3). To ensure consistency, many inconsistencies were corrected, leading to a new version of Calypso v8 (January 2020) presented in Section 6.1. 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 6.2, 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 article as part of its evaluation as discussed in Section 7. This section shows that our approach supports also the understanding of class name evolution.

6.1. 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 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

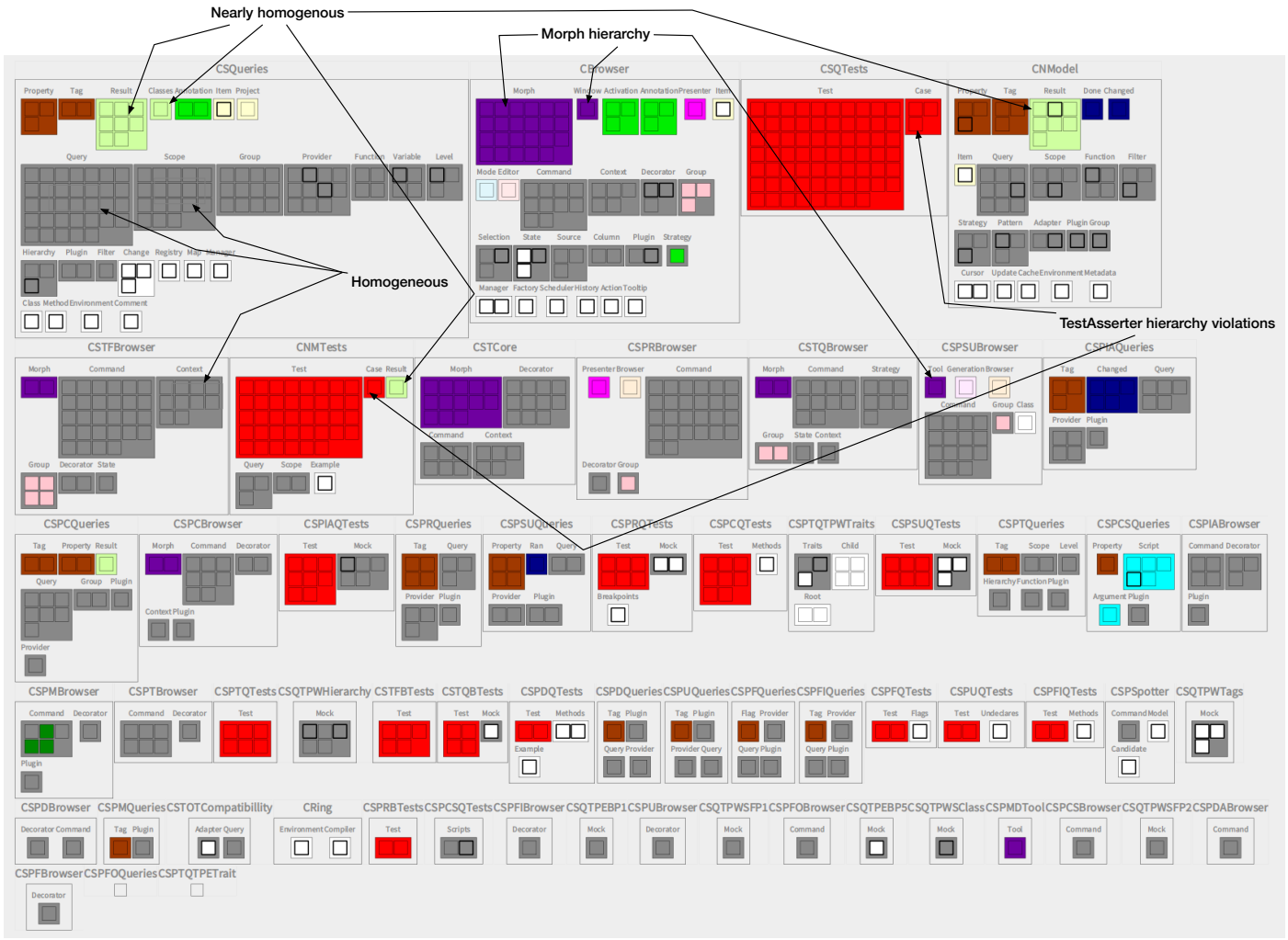

Figure 6 The ClassName Distribution of the Calypso project (v9) main packages.

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 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.

6.2. Calypso v9

As explained above, while Calypso v8 underwent major class renaming, applying the ClassName Distribution visualization revealed some remaining class name irregularities. Figure 6 depicts the current version of Calypso v9 (2021) as a result of an evaluation with the maintainers (see Section 7). 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).

7. Qualitative Evaluation

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 visualization 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.

7.1. Protocol for Domain Expert / Visualization Learners

Protocol. For this setup, we first prepared a 10-minute Powerpoint presentation of the tool which includes (i) a summary of the approach principles (described in Section 3) and (ii) instructions on how to use the tool (described in Section A). The presentation serves as a support guide for the tool.

For each project, we asked its practitioners (1 to 3 per project)[7] 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's GitHub repository and checked if the changes were integrated into the projects.

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

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.

7.1.1. Calypso v8 Experience Feedback.

As discussed in Sections 4 and 6, 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 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).

[7] 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.

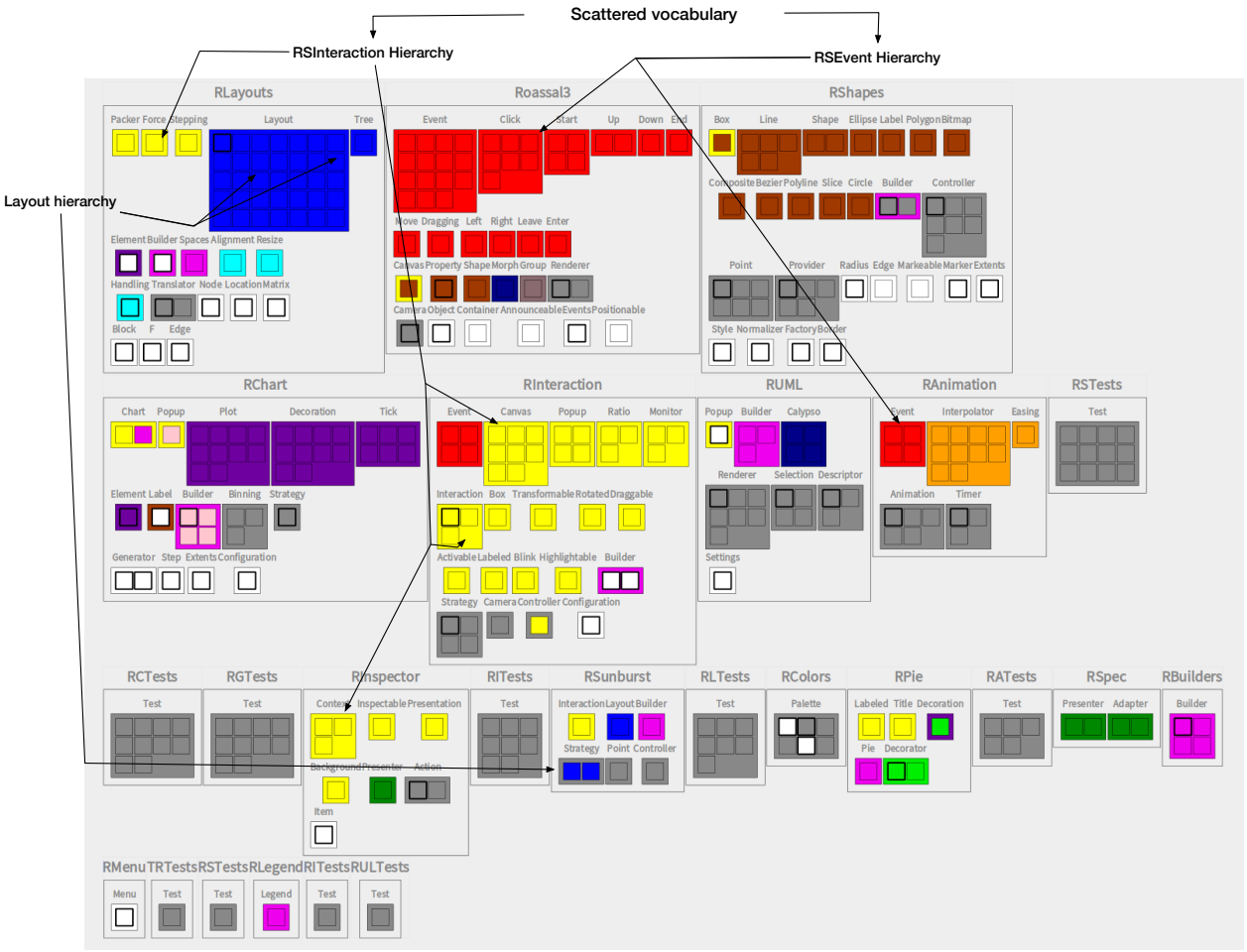

Figure 7 The ClassName Distribution of the Roassal-3 project.

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 2.6).

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.

7.1.2. 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 7 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 7) which had scattered vocab-

ulary 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 7): 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 7), 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.

7.1.3. 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 8. 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

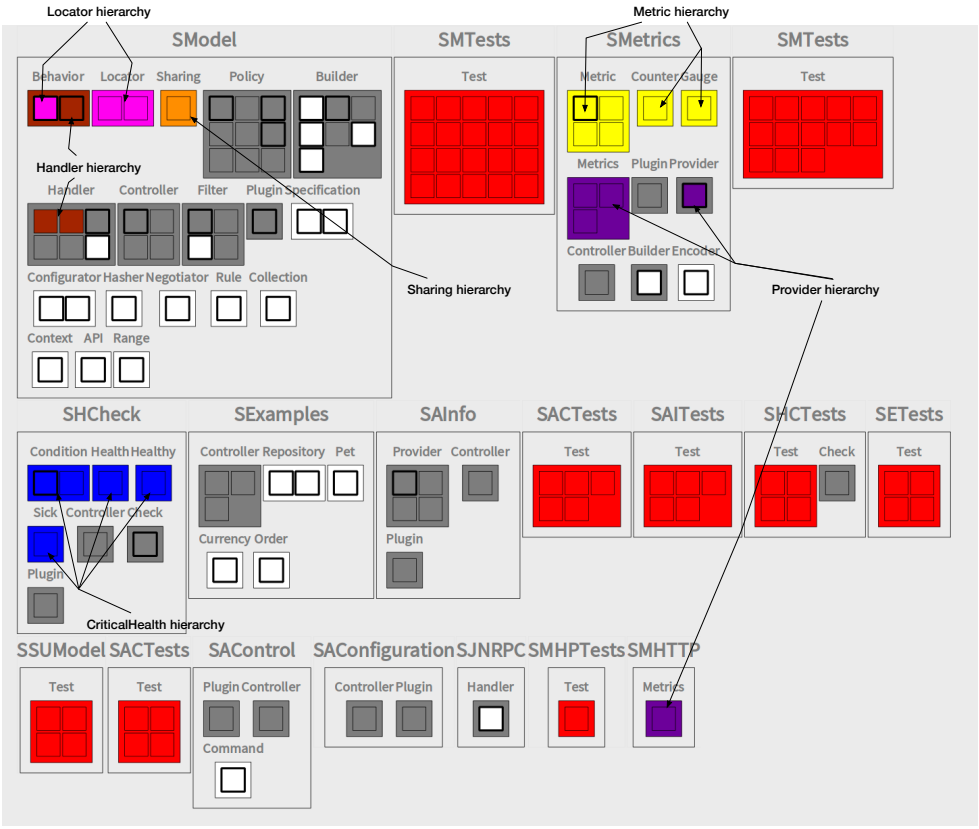

Figure 8 The ClassName Distribution of the Stargate project.

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 behavior 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 subclasses. 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.

7.2. 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[8] and Morphic, a graphics and widget library that are part of Pharo[9] 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.

7.2.1. 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 & 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 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.

7.2.2. 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.

7.3. Effective Class Renaming

Table 1 sums up these two experiments by gathering for all projects the number of classes that have been renamed using the ClassName Distribution tool.

[8] hosted at http://github.com/pharo-spec/spec
[9] http://github.com/pharo-project/pharo

Table 1 Number of classes and renamed classes per project.

Project	#classes	#renamings
Calypso v8	716	91
Roassal	326	68
Spec	795	34
Morphic	403	15
Stargate	151	12
Willow	234	12

8. 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.

8.1. 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 2 summarizes the data extracted from the selected Java projects.

8.2. 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:

- 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's correlation between the metrics to gain a better insight on the relation between the visualization and software metrics.

8.3. Java Projects Visual Patterns Analysis

Table 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 2 provides only general data about the projects without entering into the details of the visual patterns. However, complete data are available[10] 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 5.
- **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

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

Table 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
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

hierarchy raises questions about the 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.

8.4. Correlations between metrics

The heatmap of the correlations[11] 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.

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.

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

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.

9. Discussion

Here we discuss some aspects of the proposed visualization.

9.1. 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.

9.2. 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 B.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.

9.3. 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.

9.4. 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.

9.5. 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 3 and 5. In version v9, Figure 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.

9.6. 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. The approach itself 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. 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 11. 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. We tried to minimize such aspects by limiting the interactions to the essential and we described it in the Appendix.

10. Related Work

This paper addresses the problem of class name consistency over hierarchies and packages. It proposes a visualization called *ClassName Distribution* to gain an understanding of the class name consistency within their hierarchy and scoped by packages. While there is a large body of work on identifiers, our work focuses on providing compact visual maps about the consistency of class names but structured around packages. We report works around visualizations, identifiers, and concept location, even if the last ones are not directly connected to our work.

10.1. Visualizations

There is an extensive body of work related to the program visualization (Stasko et al. 1998; Ware 2000; Spence 2001; von Landesberger et al. 2011; Caserta & Zendra 2011; Merino et al. 2019). Kienle and Müller (Kienle & Müller 2010) present requirements for reverse engineering tools and their evaluation.

Class Visualization. Lanza's Polymetric views enrich simple program visualizations such as inheritance trees with metrics (Lanza & 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. Class Blueprint visualizes the 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 (Ducasse & Lanza 2005). Churcher *et al.* used 3D to visualize class cohesion (Churcher et al. 2003). 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. 2016). Glyph could be used to convey class identifiers but our proposal is more compact.

Package visualization. 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 (Beyer 2005; Eick et al. 2002; Froehlich & Dourish 2004; Storey et al. 2005; Voinea et al. 2005; Xie et al. 2006), whereas in this paper we focus on finding class name inconsistencies. Abdeen *et al.* (Abdeen et al. 2014) focus on understanding the fan in-out dependencies of packages to help split them.

Distribution Map (Ducasse et al. 2006) is a project level visualization showing how a given property (such as authors or commits) spreads into a system. The ClassName Distribution shares the scalability of the approach and the idea not to display relationships. Ducasse *et al.* present Butterfly (Ducasse et al. 2005), a radar-based visualization that can be used to present the values of several package metrics (*e.g.,* the number of classes, the number of incoming and outcoming dependencies for a package), but only gives a high-level abstract view of the package structure.

Evolution. Chuah and Eick (Chuah & Eick 1998) use rich glyphs to characterize software artefacts and their evolution (number of bugs, number of deleted lines...). D'Ambros *et al.* (D'Ambros & Lanza 2006) propose an evolution radar to depict the package coupling based on their evolution. The radar view is effective at identifying outliers but does not detail the structure. Pinzger *et al.* use Kiviat diagrams to present the evolution of package metrics (Pinzger et al. 2005).

Identifier clustering and visualization. Kuhn *et al.* (Kuhn et al. 2007) use information retrieval to exploit linguistic information found in source code, such as identifier names and comments. They introduce Semantic Clustering, a technique based on Latent Semantic Indexing and clustering to group source artefacts that use similar vocabulary. They interpret such clusters as linguistic topics that reveal the intention of the code. They compare the topics to each other, identify links between them and provide automatically retrieved labels. They propose a correlation matrix-based visualization to show how the linguistic topics are distributed over the system. While the basis of their work takes as input identifier names and comments, the clustering and the correlation matrix introduce a distance between the visualization and the physical location in the code. This makes the results difficult to comprehend. In addition, they do not help one to understand the consistency of class names within a hierarchy.

Yano *et al.* (Yano & 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 & Lanza 2007) visualization, and proposed better map labelling. Their visualization is related to lemmas of class/method names. But this visualization does not help one to understand the inconsistencies in class names.

Other. Marcus *et al.* (Marcus et al. 2003) propose a matrix-based representation of files. Each dot 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. 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.

Assessing the usefulness of a visualization is a difficult task. From a methodology perspective, and to evaluate data visualization, Elmqvist *et al.* (Elmqvist & Yi 2015) proposed a pattern language that describes various challenges faced during visualization validation. They present 12 patterns classified into three categories: study-level, method-level, and trial-level.

10.2. About class names

Butler *et al.* did several studies around 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 (capitalisation 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. While their work correlates 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 which can impose local naming conventions or the creation of subconcepts. 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. In 2011, they (Butler et al. 2011a) propose an automated way to tokenize identifier names.

In the same year, the same authors (Butler et al. 2011b) studied the class naming conventions. As such they are really related to the visualization presented in this article. They identified conventional patterns found in the use of parts of speech. Secondly, they 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.

The visualization presented in this article displays and stresses the regularity in name suffixes. However, our visu-

alization uses the same patterns and focuses on the relationship between a class and its superclass focusing on the suffix.

Singer and Kirkam (Singer & 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. Our work does not offer an analysis per se but helps one to spot the deviations from the conventions while taking into account that a hierarchy can spawn multiple packages.

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.

10.3. Method names

The matter of consistent naming was also a subject of interest for other research on method names.

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 which 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 the positive results were obtained by 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 & 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. This is clearly not related to our approach.

10.4. Identifiers

Anquetil and Lethbridge (Anquetil & Lethbridge 1998) proposed a naming convention based amongst others on the fact that two software artefacts with the same name should implement the same concept.

Kuhn *et al.* (Kuhn et al. 2005) studied how terms are distributed in a system using Latent Semantic Indexing (LSI). The authors cluster software artefacts that use similar terms. Marcus *et al.* (Marcus et al. 2004) used LSI to identify and recover links between documentation and source code. Moreno *et al.* (Moreno et al. 2013) investigated the vocabulary relationship between source code and bugs.

Li *et al.* (Li and Liu, Hui and Nyamawe, Ally S 2020) analyzed 109 existing research papers on renamings of software entities. Including 29% of the dataset dedicated to the preprocessing of identifiers. The other research was categorised according to the process of renaming: identification of renaming opportunities, renaming execution, and the detection or the evaluation of renamings. The survey showed that 36% of the literature focuses on the identification of rename opportunities–including 79% of the 39% that recommend new names, only 18% focuses on the execution of renamings, and 17% on the detection and analysis of renamings.

Arnaoudova *et al.* (Arnaoudova et al. 2014) surveyed renaming identifiers with 71 developers to understand the importance of renaming operations. The results of the survey highlighted the assumption that developers consider renaming refactoring to be a challenging activity which they frequently perform on the source code. Eshkevari *et al.* (Eshkevari et al. 2011) explored how, when, and why identifiers change in code showing the importance of identifier names in source code consistency. Lacomis *et al.* (Lacomis et al. 2018) worked on automatically renaming identifiers by assigning meaningful identifier names to variables using statistical machine translation (SMT). Recently, Nie *et al.* (Nie et al. 2020) present a multi-input neural network generation model for Coq Lemma names.

10.5. Concept location

Rajlich and Wilde (Rajlich & Wilde 2002) mention identifier-based concept recognition as one possible strategy for concept location. Concept location is the problem of finding already known concepts in source code. Abede *et al.* (Abebe et al. 2011) investigated how lexicon bad smells affect Information Retrieval-based concept location. Falléri *et al.* (Falleri et al. 2010) proposed an approach that automatically extracts and organises concepts from software identifiers in a WordNet-like structure: lexical views. Haiduc and Marcus (Haiduc & Marcus 2008) studied how domain terms are used in comments and identifiers.

Deissenboeck and Pizka (Deissenboeck & Pizska 2005) proposed a formal model of concepts and names. They presented a tool with an identifier dictionary to provide maintainers with a guideline for turning a concept into a name and to better understand the precise concept. Lawrie *et al.* (Lawrie et al. 2007) carried out an empirical study to assess the quality of source code identifiers. They indicated that full words, as well as recognizable abbreviations, lead to better comprehension. Poshyvanyk *et al.* (Poshyvanyk & Marcus 2007) address the problem of concept location in source code by presenting an approach which combines formal concept analysis (FCA) and latent semantic indexing (LSI).

To better understand how developers select concept names to attribute to identifier names, Feitelson *et al.* (Feitelson et al. 2020) experimented with 334 participants– both students and professionals. They found that the probability that participants would choose the same names was very low: 6.8%. They followed this experiment by introducing 100 subjects to a three-step naming model: (i) selecting the concepts of the entity in question, (ii) choosing the right words to represent the selected concepts, and (iii) finally constructing the name. The focus of this experiment was on assigning names, not understanding names. Results showed that the three-step model encourages the use of more concepts and longer names. In Feitelson's experience, the selected names were strongly influenced by the words used in the scenario descriptions. Regev *et al.* (Regev et al. 2021) extended the previous research: instead of using languages such as Hebrew and English, they used emojis and graphs in the scenario descriptions. The motivation was to reduce the accessibility bias by not introducing any intermediate words, considering that emojis represent a universal mode of communication. However, their attempt to reduce even more the accessibility bias than the research done by Feitelson *et al.* (Feitelson et al. 2020), was not successful as emojis only helped in reducing the accessibility bias to a similar degree.

10.6. Class comments

To understand the purpose of Java comments in assessing source code maintainability, Pascarella *et al.* (Pascarella et al. 2019) investigated 14 Java projects using machine learning to classify code comments at the line level. They present a taxonomy to improve techniques that use comments as a way to measure the maintainability of a software system. In the context of assessing class comment types and quality, Rani *et al.* (Rani, Panichella, Leuenberger, Ghafari, & Nierstrasz 2021) analyzed class comments of Pharo releases over 11 years (2008 to 2019). They studied the contents of comments and their writing style as well as how they change over time. They report 23 types of information in comments that can be identified by various patterns found in class comments. These patterns help one to identify comment information automatically. Nonetheless, they found that developers maintain both old and new class comments following different conventions when writing class comments while still using the writing style of the template. In addition, Rani *et al.* (Rani, Panichella, Leuenberger, Di Sorbo, & Nierstrasz 2021) investigate the different language-specific class comment types used in different programming languages: Java, Python, and Smalltalk. These works do not focus on the class name understanding or identification of inconsistencies.

Such work on class comments is complementary to the visualization presented in this article in the sense that comments are expressed in free form and can be desynchronized from the actual element they refer to. In our visualization, we only take into account class names.

10.7. Code Review

Code review is a common practice in software engineering. (Bacchelli & Bird 2013) surveyed 17 industrial developers from different backgrounds to help find defects in software. The study reveals that while finding defects remains the main motivation for review, reviews are less about defects than expected and instead provide additional benefits such as knowledge transfer, increased team awareness, and the creation of alternative solutions to problems.

(Panichella & Zaugg 2020) empirically investigated code review approaches and tools that developers judge as useful in modern code review (MCR). The results show that due to new technologies, developers perform more tasks of code review, hence, additional feedback is expected by the reviewers. Such feedback concerns recommendations that help enhance MCR techniques and build recommenders to automatically review source code.

In the same context and as reported in previous sections, developers using our visualization extracted hidden knowledge from their project: they found dead code, duplicated code, bad design, violations and new naming conventions. Moreover, developers found some personalized patterns of class naming in their projects.

11. 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. We illustrated the visualization on two projects – one written in Pharo and the other in Java. We showed that the visualization supports the evolution of projects: it helped the evolution of a large project over several years. 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. 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.

About the authors

Nour Jihene Agouf is a PhD student funded by Arolla, a company located in Paris and supervised by Inria Lille researchers. Her PhD topic is on software mapping and visualization for software analysis, and more generally on software reengineering. Contact her at: a.n.djihan@gmail.com

Dr. Stéphane Ducasse is research director at INRIA Lille leading the RMoD Team, France. During 10 years, he co-directed with O. Nierstrasz the Software Composition Group. He is president of ESUG. He is one of the leaders of Pharo: a new exciting dynamic language http://www.pharo.org with an industrial consortium http://consortium.pharo.org. Contact him at You can contact him at stephane.ducasse@inria.fr or visit http://stephane.ducasse.free.fr.

Anne Etien is a full Professor at the University of Lille, France. Her research interests focus on the reengineering of complex legacy systems, tests and software migration. On these topics, she has published several articles in journals and peer-reviewed conferences, and supervised several PhD students. Contact her at anne.etien@inria.fr

Appendices

We do not consider the ClassName Distribution tool to be the focus of this article. However so that readers can understand the evaluations made in this article, we present the tool's functionalities. In addition, for reproducibility purposes, we describe exactly the algorithms used by the visualization.

A. The ClassName Distribution tool

The visualization presented in this article 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 9.

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 9 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 9) 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 B.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 9. 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 3.4 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).

Figure 9 Tool's interface with TestAsserter suffix classes highlighted: the tool surrounds with a white border irregularly named classes and darkens the rest of the visualisation.

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.

B. 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:

a) Cleaning and tokenizing the class names,

b) Identifying the spfix,

c) Identifying the color of the class, and

d) 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.

B.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.

B.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.

Algorithm 1 Choosing the representing SP-fix

```
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: endIf
9: endFunction
```

B.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.

B.3.1. Identifying the Colors per Hierarchy. As explained in Section 3.3, 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 1) we attribute to each hierarchy object a type (trait, mono-class, mono-spfix, multi-spfix) (Line 3, Algo 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.

Algorithm 2 Attributing colors to hierarchies function

```
1: Procedure coloring(hierarchies)
2: for (i = 1 : hierarchies size : i++) do
3:     attributeHierarchyType(hierarchy[i])
4:     assignColorTo(hierarchy[i])
5: endFor
   endProcedure
```

As detailed in algorithm 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 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.

Algorithm 3 Attributing a type to a hierarchy

```
1: Procedure attributeHierarchyType(hierarchy)
2: if (hierarchy.subclasses has one element) then
3:     if hierarchy.subclasses first element is trait then
4:         hierarchy.type ← traitType
5:     else
6:         hierarchy.type ← monoClassType
       endIf
7: else
8:     if then(all the hierarchy.subclasses and hierarchy.root
   have the same SPfix)
9:         hierarchy.type ← monoSPFixType
10:    else
11:        hierarchy.type ← multiSPFixType
       endIf
   endIf
12: endProcedure
```

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).

B.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.

B.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 [12] 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.

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

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 atspfix. Each box corresponding to an spfix or a class is created.

Acknowledgments

The authors want to thank Arolla for the funding of Nour-Jihene Agouf's research and gratefully acknowledge the financial support of the Métropole Européenne de Lille - CPER DATA3.

References

Abdeen, H., Ducasse, S., Pollet, D., Alloui, I., & Falleri, J.-R. (2014, February). The package blueprint: Visually analyzing and quantifying packages dependencies. *Science of Computer Programming*, *89*, 298–319. doi: 10.1016/j.scico.2014.02.016

Abdeen, H., Ducasse, S., Sahraoui, H. A., & Alloui, I. (2009). Automatic package coupling and cycle minimization. In *Proceedings of the 16th international working conference on reverse engineering (wcre'09)* (pp. 103–112). Washington, DC, USA: IEEE Computer Society Press. doi: 10.1109/WCRE.2009.13

Abebe, S., Haiduc, S., Tonella, P., & Marcus, A. (2009, October). Lexicon bad smells in software. In *Working conference on reverse engineering (WCRE '09)* (pp. 95–99). Little, France. doi: 10.1109/{WCRE}.2009.26

Abebe, S., Haiduc, S., Tonella, P., & Marcus, A. (2011, September). The effect of lexicon bad smells on concept location in source code. In *International working conference on source code analysis and manipulation (SCAM'11)* (pp. 125–134). Williamsburg, VA, USA. doi: 10.1109/SCAM.2011.18

Allamanis, M., Barr, E. T., Bird, C., & Sutton, C. (2015). Suggesting accurate method and class names. In *Proceedings of the joint meeting on foundations of software engineering* (pp. 38–49).

Alsuhaibani, R. S., Newman, C. D., Decker, M. J., Collard, M. L., & Maletic, J. I. (2021). On the naming of methods: A survey of professional developers. In *International conference on software engineering*.

Anquetil, N., Etien, A., Houekpetodji, M. H., Verhaeghe, B., Ducasse, S., Toullec, C., ... Derras, M. (2020, December). Modular moose: A new generation of software reengineering platform. In *International conference on software and systems reuse (icsr'20)*. doi: 10.1007/978-3-030-64694-3_8

Anquetil, N., & Lethbridge, T. C. (1998). 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* (pp. 213–222). IBM Press. Retrieved from http://portal.acm.org/citation.cfm?id=783160.783164

Anslow, C., Noble, J., Marshall, S., & Tempero, E. (2008). Visualizing the word structure of java class names. In *Companion to the 23rd acm sigplan conference on object-oriented programming systems, languages and applications* (pp. 777–778).

Antoniol, G., Gueheneuc, Y.-G., Merlo, E., & Tonella, P. (2007, October). Mining the lexicon used by programmers during sofware evolution. In *ICSM 2007: Ieee international conference on software maintenance* (pp. 14–23). doi: 10.1109/ICSM.2007.4362614

Araya, V. P., Bergel, A., Cassou, D., Ducasse, S., & Laval, J. (2013, September). Agile visualization with Roassal. In *Deep into pharo* (pp. 209–239). Square Bracket Associates.

Arnaoudova, V., Eshkevari, L. M., Di Penta, M., Oliveto, R., Antoniol, G., & Guéhéneuc, Y.-G. (2014). Repent: Analyzing the nature of identifier renamings. *IEEE Transactions on Software Engineering*, *40*(5), 502–532.

Bacchelli, A., & Bird, C. (2013). Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 2013 international conference on software engineering* (pp. 712–721). Piscataway, NJ, USA: IEEE Press. Retrieved from http://dl.acm.org/citation.cfm?id=2486788.2486882

Beyer, D. (2005). Co-change visualization. In *Proceedings of the 21st ieee international conference on software maintenance, industrial and tool volume* (pp. 89–92). Retrieved from http://citeseer.ist.psu.edu/beyer05cochange.html

Binkley, D., Lawrie, D., Maex, S., & Morrell, C. (2009). Identifier length and limited programmer memory. *Science of Computer Programming*, *74*(7), 430–445.

Butler, S., Wermelinger, M., Yu, Y., & Sharp, H. (2009). Relating identifier naming flaws and code quality: An empirical study. In *European conference on software maintenance and reengineering (csmr)*. IEEE Press.

Butler, S., Wermelinger, M., Yu, Y., & Sharp, H. (2010). Exploring the influence of identifier names on code quality: An empirical study. In *European conference on software maintenance and reengineering (csmr)*. IEEE Press.

Butler, S., Wermelinger, M., Yu, Y., & Sharp, H. (2011a). Improving the tokenisation of identifier names. In *European conference on object-oriented programming (ECOOP)*. Springer.

Butler, S., Wermelinger, M., Yu, Y., & Sharp, H. (2011b). Mining java class identifier naming conventions. In *International conference on software maintenance (ICSM)* (pp. 1641–1643). IEEE Press. Retrieved from https://ieeexplore.ieee.org/document/6080776

Caserta, P., & Zendra, O. (2011). Visualization of the static aspects of software: a survey. *IEEE Transactions on Visualization and Computer Graphics*, *17*(7), 913-933.

Chuah, M. C., & Eick, S. G. (1998, July). Information rich glyphs for software management data. *IEEE Computer Graphics and Applications*, *18*(4), 24–29.

Churcher, N., Irwin, W., & Kriz, R. (2003). Visualising class cohesion with virtual worlds. In *Apvis '03: Proceedings of the asia-pacific symposium on information visualisation* (pp. 89–97). Darlinghurst, Australia, Australia: Australian Computer Society, Inc.

D'Ambros, M., & Lanza, M. (2006). Reverse engineering with logical coupling. In *Proceedings of WCRE 2006 (13th working conference on reverse engineering)* (p. 189 - 198).

Deissenboeck, F., & Pizka, M. (2006). Concise and consistent naming. *Software Quality Journal*, *14*(3), 261–282.

Deissenboeck, F., & Pizska, M. (2005, May). Concise and consistent naming. In *International workshop on program comprehension (iwpc 2005)* (pp. 97–106).

Demeyer, S., Ducasse, S., & Nierstrasz, O. (2002). *Object-oriented reengineering patterns*. Morgan Kaufmann.

Ducasse, S., Gîrba, T., & Kuhn, A. (2006). Distribution map. In *Proceedings of 22nd ieee international conference on software maintenance* (pp. 203–212). Los Alamitos CA: IEEE Computer Society. doi: 10.1109/ICSM.2006.22

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

Ducasse, S., Lanza, M., & Ponisio, L. (2005). Butterflies: A visual approach to characterize packages. In *Proceedings of the 11th ieee international software metrics symposium (metrics'05)* (pp. 70–77). IEEE Computer Society. doi: 10.1109/METRICS.2005.15

Eick, S., Graves, T., Karr, A., Mockus, A., & Schuster, P. (2002). Visualizing software changes. *IEEE Transactions on Software Engineering*, *28*(4), 396–412.

Elmqvist, N., & Yi, J. S. (2015). Patterns for visualization evaluation. *Information Visualization*, *14*(3), 250–269.

Eshkevari, L. M., Arnaoudova, V., Di Penta, M., Oliveto, R., Guéhéneuc, Y.-G., & Antoniol, G. (2011). An exploratory study of identifier renamings. In *Proceedings of the 8th working conference on mining software repositories* (pp. 33–42).

Fabry, J., & Ducasse, S. (2017). *The spec ui framework*. Square Bracket Associates. Retrieved from http://books.pharo.org

Falleri, J.-R., Huchard, M., Lafourcade, M., Nebut, C., Prince, V., & Dao, M. (2010, June). Automatic extraction of a wordnet-like identifier network from software. In *International conference on program comprehension (icpc)* (p. 4 -13). IEEE. doi: 10.1109/ICPC.2010.12

Feitelson, D., Mizrahi, A., Noy, N., Shabat, A. B., Eliyahu, O., & Sheffer, R. (2020). How developers choose names. *IEEE Transactions on Software Engineering*.

Fernandez, I., andJ. P. S. Alcocer, A. B., Infante, A., & Gîrba, T. (2016). Glyph-based software component identification. In *International conference on program comprehension (ICPC)* (p. 1-10).

Froehlich, J., & Dourish, P. (2004). Unifying artifacts and activities in a visual tool for distributed software development teams. In *Proceedings of the 26th international conference on software engineering* (pp. 387–396). Washington, DC, USA: IEEE Computer Society.

Goldberg, A. (1984). *Smalltalk 80: the interactive programming environment*. Reading, Mass.: Addison Wesley.

Haiduc, S., & Marcus, A. (2008, June). On the use of domain terms in source code. In *International conference on program comprehension (ICPC'08),* (pp. 113–122). Amsterdam, The Netherlands. doi: 10.1109/{ICPC}.2008.29

Healey, C. G., Booth, K. S., & Enns, J. T. (1993). Harnessing preattentive processes for multivariate data visualization. In *Gi '93: Proceedings of graphics interface.*

Isobe, Y., & Tamada, H. (2018). Are identifier renaming methods secure? In *2018 19th ieee/acis international conference on software engineering, artificial intelligence, networking and parallel/distributed computing (snpd)* (pp. 322–328).

Kienle, H. M., & Müller, H. A. (2010). The tools perspective on software reverse engineering: Requirements, construction, and evaluation. In *Advanced in computers* (Vol. 79, pp. 189–290). Elsevier.

Kobayashi, K., Kamimura, M., Yano, K., Kato, K., & Matsuo, A. (2013). Sarf map: Visualizing software architecture from feature and layer viewpoints. In *2013 21st international conference on program comprehension (icpc)* (pp. 43–52).

Kuhn, A., Ducasse, S., & Gîrba, T. (2005, November). Enriching reverse engineering with semantic clustering. In *Proceedings of 12th working conference on reverse engineering (wcre'05)* (pp. 113–122). Los Alamitos CA: IEEE Computer Society Press. doi: 10.1109/WCRE.2005.16

Kuhn, A., Ducasse, S., & Gîrba, T. (2007, March). Semantic clustering: Identifying topics in source code. *Information and Software Technology*, *49*(3), 230–243. doi: 10.1016/j.infsof.2006.10.017

Lacomis, J., Jaffe, A., Schwartz, E. J., Le Goues, C., & Vasilescu, B. (2018). Statistical machine translation is a natural fit for automatic identifier renaming in software source code. In *Workshops at the thirty-second aaai conference on artificial intelligence.*

LaLonde, W., & Pugh, J. (1991, January). Subclassing $\neq$ Subtyping $\neq$ Is-a. *Journal of Object-Oriented Programming*, *3*(5), 57–62. Retrieved from http://scgresources.unibe.ch/ scg/Literature/PL/LaLo91a-JOOP0305.pdf

Lanza, M. (2001). The evolution matrix: Recovering software evolution using software visualization techniques. In *Proceedings of the international workshop on principles of software evolution* (pp. 37–42). Retrieved from http://scg.unibe.ch/archive/papers/Lanz01cEvolutionMatrix.pdf doi: 10.1145/602461.602467

Lanza, M. (2003). *Object-oriented reverse engineering — coarse-grained, fine-grained, and evolutionary software visualization* (Doctoral dissertation, University of Bern). Retrieved from http://scg.unibe.ch/archive/phd/lanza-phd.pdf

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

Lanza, M., & Marinescu, R. (2006). *Object-oriented metrics in practice*. Springer-Verlag. Retrieved from http://www.springer.com/alert/urltracking.do?id=5907042

Lawrie, D., Feild, H., & Binkley, D. (2007, August). Quantifying identifier quality: an analysis of trends. *Empirical Software Engineering*, *12*(4), 359–388. Retrieved from https://doi.org/10.1007/s10664-006-9032-2 doi: 10.1007/s10664-006-9032-2

Lawrie, D., Morrell, C., Feild, H., & Binkley, D. (2006, June). What's in a name? a study of identifiers. In *14th ieee international conference on program comprehension (ICPC'06)* (p. 3-12). doi: 10.1109/{ICPC}.2006.51

Li, Wang, S., & Nguyen, T. N. (2021). A context-based automated approach for method name consistency checking and suggestion. In *2021 ieee/acm 43rd international conference on software engineering (ICSE)* (pp. 574–586).

Li and Liu, Hui and Nyamawe, Ally S. (2020). A survey on renamings of software entities. *ACM Computing Surveys (CSUR)*, *53*(2), 1–38.

Liblit, B., A.Begel, & Sweetser, E. (2006). Cognitive perspectives on the role of naming in computer programs. In *Annual psychology of programming workshop.*

Liu, K., Kim, D., andTaeyoung Kim, T. F. B., Kim, K., and-Suntae Kim, A. K., & Traon, Y. L. (2019). Learning to spot and refactor inconsistent method names. In *Proceedings of icse'19.*

Marcus, A., Feng, L., & Maletic, J. I. (2003). 3D representations for software visualization. In *Proceedings of the acm symposium on software visualization* (p. 27-ff). IEEE.

Marcus, A., Sergeyev, A., Rajlich, V., & Maletic, J. (2004, November). An information retrieval approach to concept location in source code. In *Proceedings of the 11th working conference on reverse engineering (WCRE 2004)* (pp. 214–223).

Martin, R. C. (2000). *Design principles and design patterns.* (www.objectmentor.com)

Merino, L., Kozlova, E., Nierstrasz, O., & Weiskopf, D. (2019). VISON: An ontology-based approach for software visualization tool discoverability. In *VISSOFT'19: Proceedings of the 7th ieee working conference on software visualization.* IEEE. Retrieved from http://scg.unibe.ch/archive/papers/Meri19b-vison.pdf doi: 10.1109/VISSOFT.2019.00014

Moreno, L., Bandara, W., Haiduc, S., & Marcus, A. (2013, September). On the Vocabulary Relationship Between Bug Reports and Source Code. In *Ieee international conference on software maintenance (ICSM'13), early research achievement track (ERA)* (p. 452-455). Eindhoven, The Netherlands. doi: 10.1109/ICSM.2013.70

Newman, C. D., AlSuhaibani, R. S., Collard, M. L., & Maletic, J. I. (2017). Lexical categories for source code identifiers. In *Proceedings of saner* (pp. 228–239). IEEE.

Nguyen, S., Phan, H., Le, T., & Nguyen, T. N. (2020). Suggesting natural method names to check name consistencies. In *Proceedings of the acm/ieee 42nd international conference on software engineering* (pp. 1372–1384).

Nie, P., Palmskog, K., Li, J. J., & Gligoric, M. (2020). Deep generation of coq lemma names using elaborated terms. In *International joint conference on automated reasoning* (pp. 97–118).

Osman, H., van Zadelhoff, A., Stikkolorum, D. R., & Chaudron, M. R. (2012). Uml class diagram simplification: what is in the developer's mind? In *Proceedings of the second edition of the international workshop on experiences and empirical studies in software modelling* (p. 5).

Panichella, S., & Zaugg, N. (2020). An empirical investigation of relevant changes and automation needs in modern code review. *Empirical Software Engineering*, *25*(6), 4833–4872.

Pascarella, L., Bruntink, M., & Bacchelli, A. (2019). Classifying code comments in java software systems. *Empirical Software Engineering*, *24*(3), 1499–1537.

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

Pinzger, M., Gall, H., Fischer, M., & Lanza, M. (2005, May). Visualizing multiple evolution metrics. In *Proceedings of softvis 2005 (2nd acm symposium on software visualization)* (pp. 67–75). St. Louis, Missouri, USA.

Poshyvanyk, D., & Marcus, A. (2007). Combining formal concept analysis with information retrieval for concept location in source code. In *ICPC '07: Proceedings of the 15th ieee international conference on program comprehension* (pp. 37–48). Washington, DC, USA: IEEE Computer Society. Retrieved from http://dx.doi.org/10.1109/ICPC.2007.13 doi: 10.1109/{ICPC}.2007.13

Rajlich, V., & Wilde, N. (2002). The role of concepts in program comprehension. In *Program comprehension workshop* (pp. 271–278).

Rani, P., Panichella, S., Leuenberger, M., Di Sorbo, A., & Nierstrasz, O. (2021). How to identify class comment types? a multi-language approach for class comment classification. *Journal of Systems and Software*, *181*, 111047.

Rani, P., Panichella, S., Leuenberger, M., Ghafari, M., & Nierstrasz, O. (2021). What do class comments tell us? an investigation of comment evolution and practices in pharo smalltalk. *Empirical Software Engineering*, *26*(6), 1–49.

Regev, O., Soloveitchik, M., & Feitelson, D. G. (2021). Using non-verbal expressions as a tool in naming research. *arXiv preprint arXiv:2103.08701*.

Singer, J., & Kirkham, C. (2008). Exploiting the correspondence between micro patterns and class names. In *International working conference on source code analysis and manipulation.* IEEE.

Spence, R. (2001). *Information visualization*. Adisson-Wesley.

Stasko, J. T., Domingue, J., Brown, M. H., & Price, B. A. (1998). *Software visualization — programming as a multimedia experience*. The MIT Press.

Storey, M.-A. D., Čubranić, D., & German, D. M. (2005). 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* (pp. 193–202). ACM Press. Retrieved from http://portal.acm.org/citation.cfm?id=1056018.1056045 doi: 10.1145/1056018.1056045

Tempero, E., Noble, J., & Melton, H. (2008). 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* (pp. 667–691). Berlin, Heidelberg: Springer-Verlag.

Tesone, P., Ducasse, S., Polito, G., Fabresse, L., & Bouraqadi, N. (2020). A new modular implementation for stateful traits. *Science of Computer Programming*, *195*, 1–37. doi: 10.1016/j.scico.2020.102470

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

Voinea, L., Telea, A., & van Wijk, J. J. (2005). Cvsscan: visualization of code evolution. In *Softvis '05: Proceedings of the 2005 acm symposium on software visualization* (pp. 47–56). New York, NY, USA: ACM. doi: 10.1145/1056018.1056025

von Landesberger, T., Kuijper, A., Schreck, T., Kohlhammer, J., van Wijk, J. J., Fekete, J.-D., & Fellner, D. W. (2011). Visual analysis of large graphs: State-of-the-art and future research challenges. *Comput. Graph. Forum*, *30*(6), 1719-1749.

Ware, C. (2000). *Information visualization: perception for design*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Wettel, R., & Lanza, M. (2007). Visualizing software systems as cities. In *Proceedings of vissoft 2007 (4th ieee international workshop on visualizing software for understanding and analysis)* (pp. 92–99). Retrieved from http://dx.doi.org/10.1109/VISSOF.2007.4290706 doi: 10.1109/VISSOF.2007.4290706

Wirfs-Brock, R., & McKean, A. (2003). *Object design — roles, responsibilities and collaborations*. Addison-Wesley.

Xie, X., Poshyvanyk, D., & Marcus, A. (2006). Visualization of CVS repository information. In *WCRE'06: Proceedings of the 13th working conference on reverse engineering* (pp. 231–242). Washington, DC, USA: IEEE Computer Society. doi: 10.1109/{WCRE}.2006.55

Yano, K., & Matsuo, A. (2015). Labeling feature-oriented software clusters for software visualization application. In *2015 asia-pacific software engineering conference (apsec)* (pp. 354–361).