HAL
archives-ouvertes.fr

Identifying class name inconsistency in hierarchy: a first simple heuristic

Abdelghani Alidra, Moussa Saker, Nicolas Anquetil, Stéphane Ducasse

▶ To cite this version:

Abdelghani Alidra, Moussa Saker, Nicolas Anquetil, Stéphane Ducasse. Identifying class name inconsistency in hierarchy: a first simple heuristic. IWST 2017 - 12th International Workshop on Smalltalk Technologies, Sep 2017, Maribor, Slovenia. pp.14:1–14:8, 10.1145/3139903.3139920 . hal-01663603

HAL Id: hal-01663603
https://hal.inria.fr/hal-01663603

Submitted on 14 Dec 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Identifying class name inconsistency in hierarchy: a first simple heuristic

Abdelghani Alidra
Moussa Saker
Nicolas Anquetil
Stéphane Ducasse

ABSTRACT

Giving good class names is an important task. Good programmers often report that they take several attempts to find an adequate one. Often programmers do not name consistently classes within a package, project or hierarchy. This is a problem because it hampers understanding the systems. In this article we present a simple heuristic (a distribution) to characterise class naming. We combine such a heuristic with structural information to identify inconsistent class names. In addition, we use this simple heuristic to give packages a shape. We applied such heuristic to 285 packages in Pharo to identify misnamed classes. Some of these misnamed classes are reported and discussed here.

KEYWORDS

program understanding, vocabulary, symbolic information

ACM Reference format:
Abdelghani Alidra, Moussa Saker, Nicolas Anquetil, and Stéphane Ducasse. 2017. Identifying class name inconsistency in hierarchy: a first simple heuristic. In *Proceedings of xxxx, Maribor, Sep 2017 (IWST'17),* 8 pages.
https://doi.org/10.475/123_4

1 INTRODUCTION

Software evolution is a major phase in the software life cycle. Adapting software to new requirements can account to as much as 75 percent of the total cost of the software [Som00] [Dav95]. Very often, software professionals spend an important amount of time reading source code and trying to understand it. In this sense, understanding source code is considered as a critical task in software engineering and reengineering.

An important aspect of source code understanding rests on the vocabulary used to name code entities such as packages, classes and methods because it reveals developer's intention. Thus the quality of such a vocabulary can be crutial for code comprehensibility.

The problem is that, while good programmers report that they take several attempts to find appropriate entity names, the majority of programmers often do not consistently name their own entities. Moreover, when the source code evolves, it often happens that the initial vocabulary is biased or the entity names are no more accurate to generalisation. This is known as vocabulary erosion [AL11]. This is a problem because bad entity names hamper understanding the systems and complicates the maintenance tasks. Further, some vocabulary choices reveal conceptual and design problems.

In literature, source code vocabulary has been studied from two different perspectives:

A first category of works were interested in using the vocabulary as an implicit knowledge to ease programs understanding. Most existing approaches in this category use NLP (Natural Langage Processing) techniques to extract relations between code entities based on the vocabulary they use [KDG05, KDG07]. Here, the objective is to extract meaningful informations for the software engineer in order to perform tasks such as, software measurement, concept location, traceability recovery, software evolution...[CTH16, DRGP11]

A second category of approaches addressed the problem of vocabulary consistance and how to better choose entity names in order to enhance code comprehensibility. For instance, the work presented in [AL98] was one of the first attempts to question the pertinence of vocabulary of source code. Authors in [DP05a] proposed a formal model for renaming identifiers in a concise way. [LMFB06a] studied the impact on program comprehensibility of the use of single letters, abbreviations and full words in entity names and [HØ09] used NLP techniques to assess the consistence of Java method names against their implementation.

In this article, we present a new approach to assess the consistency of class names against design choices. Our approach only takes into account the prefix of the class names (the last word in the sequence of words determined by conventional separators such as case permutation and underscore) because it is the element that reflects the best the abstraction or the concept the class represents.

Our approach is based on a new visualisation that captures information contained both in the class suffixes and in the class hierarchy graph. We call this visualisation, *a conceptual blueprint*. In the same spirit than class blueprints [LD01], the objective of the conceptual blueprint is to help software engineers understand source code, detect inconsistencies and perform maintenance tasks.

More specifically, the proposed visualisation helps understanding the rational behind a project or package architecture as well as pointing the potential inconsistencies in terms of class names, architecture or inheritance hierarchy.

Besides the presentation of the technical aspects that the conceptual blueprint implies, we established a vocabulary that we developed based on the insights we obtained during several case studies. This vocabulary identifies most common visual patterns i.e., recurrent graphical situations we encountered during the validation of this work. The contribution of this article are as follows: the definition of the conceptual blueprint (a new visualisation for pointing vocabulary and/or design issues based on class suffixes

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
IWST'17, Sep 2017, Maribor
© 2016 Copyright held by the owner/author(s).
ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.475/123_4

and inheritance informations), the identification of visual package shapes, and the definition of a vocabulary base on these visual shapes.

The rest of this article is organised as follows: Section 2 presents the general context of the problem we are addressing. Section 3 introduces our conceptual blueprint visualisation. We report some anecdotal cases and discuss them with some details in Section 4. False positives are identified in Section 5. Finally, some relevant related works are reported in Section 6 and we conclude our article with Section 7.

2 PROBLEM DESCRIPTION

Classes are the building blocks and the primary abstractions from which object-oriented applications are built. Thus, understanding the concepts that lie behind classes is a key activity. Often, programmers express such concepts in the class names. Specifically, when programmers use an english like language to name their classes, the general concept that the class abstracts is pushed at the end of its name. For instance and obviously, RBLintRule is a rule, FloatingPointException is an exception and ListModel represents a model.

In the present article, we use the term suffix to refer to the last word in the class names. Indeed, a class name can be seen as a sequence of words that are easily identifiable thanks to the use of naming conventions such as the camel case style or the underscore character. For instance, considering the class name FloatingPointException the words sequence is Floating + Point + Exception and the class suffix is obviously Exception.

In our approach, we only consider the class names. Other entities such as package, methods or identifiers can be considered but this has not been explored here. Besides, we only consider class suffixes and not the entire class name. The rational behind this is that, most professional software systems both in industry and academia use english as basics to name their classes. Due to the general structure of the english language where the adjective is put before the noun it qualifies (for instance, BigClass or SmallModel) the most important keywords came at the end of the class name. Therefore, our approach applies only to programs that use english as a basis to name their classes.

In this article, we explore the problem of assessing the consistency of the vocabulary used to name classes. A class name vocabulary is consistent if the concept it suggests reflects the concept that the class actually implements. However, what we observed, is that many classes are named in a manner that does not correctly reflect their functionality. This is a problem because it hampers the understanding of the system and complicates the maintenance and evolution tasks. Indeed, when a developer examines a system, the class name is the first information it collects before looking at its inside structure given for instance by the class attributs, methods or inheritance hierarchy. That is what happens for instance when Pharo [1] developers use the class browser. If this information is ambiguous or erroneous, the analysis process can quickly become frustrating and cumbersome.

[1] http://pharo.org/

To address this problem, we introduce the Conceptual Blueprint, a new visualisation that captures the conceptual as well as the hierarchy information of the classes in a package or a system. Classes are grouped according to their suffixes and clusters are coloured to reflect the hierarchy of the belonging classes. Besides, classes that have independent hierarchies (we call these, root classes i.e., classes that are directly subclasses of the Object class) are explicitly identified. This visualisation allows us to categorise packages according to identifiable visual shapes that often reflect a vocabulary inconsistency or a design issue. We have observed however, that in some cases the identified shapes can find a logical interpretation when additional conceptual or structural information is considered. Thus, we report these false positives and discuss the rational behind them.

3 CONCEPTUAL BLUEPRINT

In this section we present the Conceptual Blueprint, a way to visualize the conceptual and structural aspects of the classes inside a package or a project. First, we present the general structure of the visualisation and discuss the way we display vocabulary and hierarchy information. We shortly discuss the layout and colouring algorithm we use before finally, displaying and discussing a first conceptual blueprint visualization.

3.1 The graphical element of a conceptual blueprint

Figure 1, depicts a template conceptual blueprint. From the outside to the inside, we have the following elements: *package boxes, suffix boxes* and *class boxes*. Specifically, the package boxes are labeled with the name of the package they represent. They contain boxes that map to the different suffixes of the classes that belong to the package. For instance, if a Package X contains three classes, two of which end with the suffix Y and the third with suffix Z, then we will have two suffix boxes, the first labeled with X and the second labeled with Z. Finally, class box are contained inside the suffix boxes.

Note that the suffix boxes are ordered according to the number of the classes ending with that suffix. This means that, the suffix box labeled with Y from the previous example, will be placed to the left of the suffix box labeled with Z because the number the classes belonging to the first is bigger than the number of classes belonging to the second.

Knowing that the suffix boxes contain boxes that represent classes belonging to them (referring to the previous example suffix box labeled with Y will contain the two boxes representing the classes ending with suffix Y), the size of suffix boxes reflects the number of these classes. This also means that big suffix boxes will always be placed to the left of small ones.

Our visualization makes also use of colours to reflect hierarchy information about classes. Especially, different colours are used to represent different class hierarchies. In the class boxes, the colour of a box is based on the root of the class inheritance hierarchy. This means that two classes belonging to the same inheritance hierarchy will be coloured the same.

The suffix boxes on the other hand, may contain classes belonging to different hierarchies. In this case, the colour of the suffix box

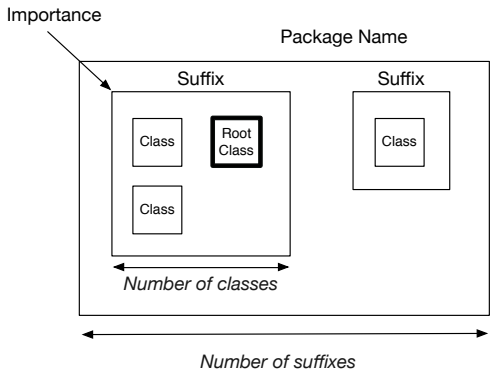


Figure 1: A template conceptual hierarchical blueprint

is the same as the colour of the dominant hierarchy that uses that suffix. For instance, if the suffix S is used in n classes of hierarchy X and m classes of hierarchy Y and n > m, then the colour of the box representing suffix S is the same as the colour of the classes belonging to hierarchy X. Finally, root classes are easily distinguishable by bordering the shape that represents them.

3.2 The layout and colouring algorithm

In this section, we briefly describe the algorithm used to layout and colour the different elements in the conceptual blueprint. First, the algorithm collects classes names in the package or in the project. To extract the suffixes, the algorithm simply splits the names using the camel case style and keeps the last word in the obtained sequence of words.

Then, for each package, the algorithm associates the set of classes that correspond to a given suffix. A box representing every suffix is created inside the package box and contains other boxes representing classes. We represent classes inside the suffix boxes for two reasons. First, the size of the suffix box will be proportional to the number of classes it contains, which allows us to identify top suffixes in the package (since we bind a suffix to a concept, this means most important topics are represented with larger boxes). Secondly, in the interactive visualization, it is possible to see the names of the classes that belong to the suffix simply by passing the mouse over the corresponding box.

The suffixes are ordered according to the number of classes they are associated to. In the visualization, this means that most important suffixes will be disposed to the upper-left corner of the package box. Again, this allows us to identify top suffixes quickly. Moreover, this permits for different visual shapes to emerge which will be discussed later.

The algorithm uses colours to distinguish different inheritance hierarchies. The idea behind this is the assumption that concepts can also be identified in the inheritance hierarchy (for instance, classes inheriting from the class Model represent models and those inheriting from the class TestAsserters are tests). To this end, the algorithm computes the inheritance sequence of every class defined in the package/system. We stop the inheritance sequence below the Object class since Object does not represent any particular concept and most classes inherit from it. This rule is however relaxed in the case of some particularly large projects like Roassal where other hierarchy roots are considered (typicaly , the RTObject class in the case of Roassal. Then, the Root inheritance class is simply the last one in the inheritance sequence. To associate colours to class hierarchies, the algorithm first orders them according to their size in the current package/project /emphi.e., the number of classes in the project/package that belong to this hierarchy. This is performed due the reduced number of distinguishable colours available so that these colours are assigned to most important hierarchies, whereas, less important ones will be coloured with different shades of gray.

Once colours have been associated to the different hierarchies of the project/package, the algorithm colours the class boxes with the colour of the hierarchy they belong to (for instance, all the subclasses of TestCase will be coloured in red and all subclasses of Model will be coloured in blue). Colouring the suffix boxes is less trivial. Indeed, more than one hierarchy could use a particular suffix. To overcome this problem, the algorithm associates colours to suffixes in the following way: First, it collects all the classes in the package/system that uses this suffix. Then it collects all the corresponding hierarchies keeping track of the number of classes using the suffix in each hierarchy. finally, the algorithm associates to the suffix the colour of the hierarchy with the larger number of classes using that suffix.

This colouring algorithm presents various advantages. First, it facilitates the detection of largely spread hierarchies in the system. Inside a particular package, it allows one to detect hierarchies that use different (potentially, inconsistent) suffixes. Lastly and most importantly, it points out suffixes that are used in different (potentially, independent) hierarchies. These points will be further discussed and illustrated in Section 4

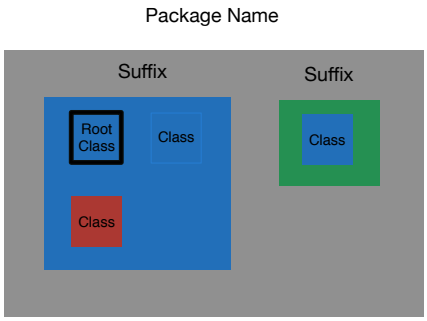


Figure 2: A template conceptual blueprint with colours

Finally, The colouring algorithm borders those classes that are hierarchy roots in order to give more visibility about the original suffix that was used in a given hierarchy.

In Figure 2 we see the template conceptual blueprint augmented with colours. We see that there are two suffixes and four classes. The first suffix (the one to the left) contains three classes one of which is root. The root class and the one to its left have the same colour as the suffix box, meaning that, most classes in the considered project ending with this particular suffix are subclasses of this particular root class. The third class in the first suffix has a different colour

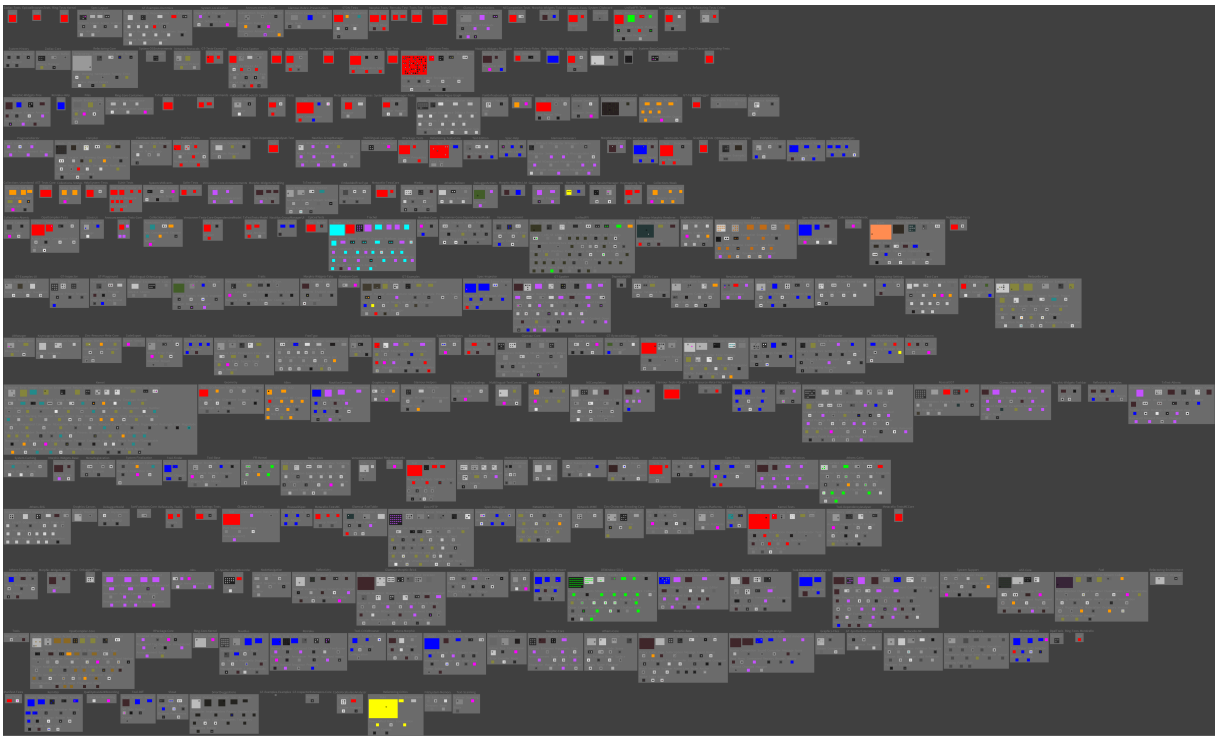


Figure 3: The conceptual blueprint of the 285 packages of Pharo.

and thus belongs to a different hierarchy. Regarding the second suffix, one can notice that the class inside has a different colour from the suffix itself and the same as the first suffix. This means that this class is a subclass of the root class in the first suffix. Moreover, the colouring schema tells us that classes ending with the second suffix usually belong to a different hierarchy that the one this class belongs to. The implications of these observations and concrete illustrations are given in the next section.

4 ANECDOTAL CASES

In this section we report some findings based on the proposed approach. We applied our approach to the Pharo system where very small packages (less than 5 defined classes) were excluded because we found that they do not bring any additional knowledge to our study. We also excluded the Roassal2 package because it was rather a system in the system and needs to be considered separately. Figure 3 depicts the conceptual blueprint of the studied system. In the next section we discuss false positives.

4.1 A first glimpse

A first glimpse at the Pharo Conceptual blueprint reveals some valuable information. For instance, one can clearly see top used hierarchies and their distribution over the Pharo packages (cross-cutting concerns). Not surprisingly, the Test hierarchy (red suffix boxes) holds the first place with 1209 subclasses. This is very reasonable because Pharo developers make an extensive use of agile techniques especially when it comes to unit testing. Moreover, it is noticeable that most test classes use uniformly the Test suffix and that this suffix is used almost exclusively with the test classes (very few ambiguous or scattered Test suffixes. See the next subsections for details).

The second top used hierarchy in the considered system is the hierarchy of Model with 497 subclasses (blue suffix boxes in figure 3). However, contrary to the Test hierarchy, subclasses of Model use very scattered vocabulary. This is clearly noticeable because blue suffix boxes are labeled with different suffixes. Still, the suffixes used with the Model hierarchy are almost exclusively associated to this hierarchy which can be concluded from the uniform color of the blue boxes in the conceptual blueprint.

Another top used hierarchy is the Announcement one. It is represented with the pink colour in figure 3. Again, this hierarchy is very scattered over many suffixes inside the same package as well as between different packages. We examine in section 5.1 the reason of such a phenomenon. Moreover, most suffixes associated to the Announcement hierarchy are uniformly associated to it though they are used in different packages. The only two exceptions are the Changed and Selected suffixes.

One can also notice that there are few packages that contain suffix boxes having a different color from the one of the classes they contain. A priori, these are ambiguous suffixes. However, a closer look at these packages revels that most of them are false positives. This is because we colour the suffix box with the same colour of the hierarchy that is most associated with the suffix through out the considered packages. This hierarchy might have a slight, non significant majority it is still considered as dominant by our

algorithm. The dominant hierarchy might also be located in some single, domain specific package, with many subclasses thus altering the concept associated to the given suffix. We further explain this limitation in section 5.3.

4.2 The Single Monolithic shape

We define a single monolithic shaped package as one that uses only one suffix to name all its defined classes *and* all the defined classes belong to the same inheritance hierarchy. It then has a single suffix box uniformly coloured. Such packages make use of only one concept and they name their classes consistently. In Figure 4 we see an actual monolithic package.



Figure 4: A monolothic shaped package.

When querying the Pharo system for such packages, we got 29 responses most of which are test packages. This result is not surprising because, very often, programmers respect the convention consisting in naming the test classes with the suffix Test (though, some exceptions). Other such packages are small ones only defining few classes.

4.3 The Pseudo monolithic shape

A pseudo monolithic shaped package is one that uses many different suffixes to name its defined classes which all belong to the same hierarchy. Figure 5 depicts an actual pseudo monolithic shaped package. Such packages are potentially ones that are using non consistent vocabulary. Indeed, following the class naming conventions, classes belonging to the same hierarchy should use the same suffix. This rule has however some reasonable exceptions that we will discuss later.

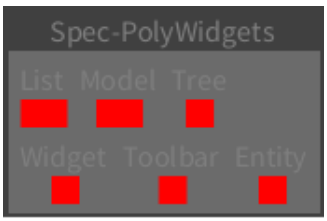


Figure 5: A pseudo monolothic shaped package.

4.4 Scattered Vocabulary

Scattered vocabulary packages are those that use many suffixes with classes that belong to the same hierarchy. Contrary to the previous category, these packages make use of more than one concept (as represented by more than an inheritance hierarchy). This is a problem because the suffix of the class may suggest that the classes in the package abstract different concepts whereas the hierarchy suggests the contrary, *i.e.,* they are specialisations of the same concept. For instance, Figure 6 depicts the conceptual blueprint of the RPackage-Tests package.



Figure 6: A vocabulary scattered shaped package.

In this package, the class TestRPackagePrequisites uses the suffix Prequisites while most other classes use the suffix Test. This may suggest that TestRPackagePrequisites is not a test. A closer look at the code of this class reveals that it has nothing particular that may justify the use of a different suffix. Actually, we uncounted this pattern in many test packages, where test classes take different (thought, close) suffixes than the conventional Test suffix, mainly the Tests and (Test)Case ones. We believe this difference is most often not justified and is rather a bad practice. When querying for this kind of packages the system returned 186 packages out of 285. It is worth mentioning however that, except from the aforementioned test packages, we found several false positives in this category. Some of these false positives are reported in section 5.

4.5 Ambiguous suffixes (Ladybug packages

A more precise diagnosis than the one introduced in the previous section is the ambiguous suffix pattern (we also call it the Ladybug pattern or package). A ladybug package is one that defines classes that use suffixes that are usually used with a different hierarchy (thus reflecting a different concept). For instance, in the package Refactoring-Tests-Core, the class RBLintRuleTest uses the suffix Test but is not a Test itself (it does not inherit from TestCase). Thus we consider that the name of this method is ambiguous and can be misleading. Figure 7 depicts such a package.

Ambiguous packages can be easily identified in the conceptual blueprint because they show suffixes containing class boxes of a different colour than the one of the suffix box itself. However, an important issue when detecting this pattern, is how to determine the threshold that determines that a given suffix is associated to a given concept. One can also argue that the same word (suffix) can designate different concepts depending on the target domain of the applications. This issue is further detailed in section 5.3.

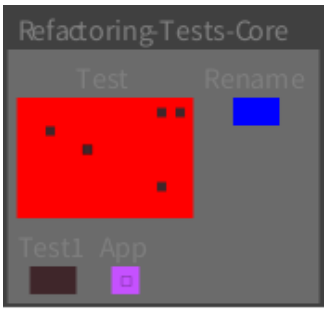


Figure 7: A sample package with an ambiguous suffix.

4.6 Dirty suffixes

Following the idea that suffixes in class names reflect the concept the class abstracts, we identified some packages containing *dirty* suffixes. A *dirty* suffix is one that is used in two inheritance root classes in the same package. This is an abnormal situation because, programmers are supposed to use suffixes to reflect related concepts in their code. Besides, related concepts are most likely to share part of their code through inheritance. Consequently, if two unrelated classes (or hierarchies) use the same suffix in the same package, this often reveals a conceptual or structural problem in the package. Let us consider the example given by figure 8. One can notice that two different hierarchies as represented by the classes STONStreamWrite and STONWriter are both using the suffix Writer. Our intuition is that the STONStreamWrite should be a subclass of STONWrite or the two classes are not related and should use different suffixes.

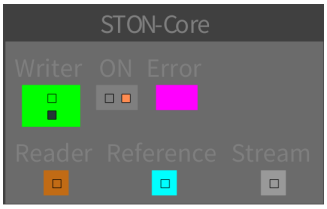


Figure 8: A sample package with a dirty suffix

4.7 Mosaic packages

A last interesting package shape we encountered is the mosaic package visual shape. Visually, such a shape is distinguishable thanks to the numerous small suffix boxes of very disparate colours and many bordered class boxes.

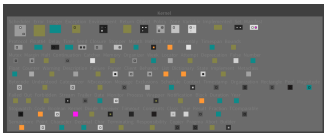


Figure 9: A sample mosaic shaped package

Usually, this kind of patterns is associated with key packages in the system (the one represented in figure 9 is the Kernel package) where many new concepts are introduced and are made use of in other packages of the system. One phenomenon we also noticed is that such packages usually have scattered, ambiguous and even dirty suffixes. We believe that this is due to the complexity of these packages but this has to be addressed using appropriate naming conventions.

5 FALSE POSITIVES

When analyzing the conceptual blueprint of Pharo, we diagnosed some package shapes as potential misnaming or conceptual problems as explained in the previous sections. However, when considering these packages more closely, we found that some of these diagnosis were incorrect. In the next subsections we report these cases and explain why we consider them as false positives.

5.1 Announcements

Many scattered suffixes we found when analyzing the Pharo conceptual blueprint were related to subclasses of Announcement. Indeed, Announcement subclasses were using very different suffixes in different packages but also in the same package. We initially diagnosed these as misnaming problems but, when considering these cases more closely, we found that programmers used a very specific naming convention for this kind of classes. Actually, most Announcement subclasses were using *verb+ed* or adjective suffixes to represent their intention. Figure 10 depicts a sample package with such a pattern (Announcement subclasses are coloured in pink). This pattern was repeated so often (more than 50% of cases) that we decided that this was a false positive and that was the rule rather than the exception.

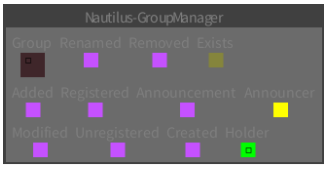


Figure 10: A false positive due to the Announcement special naming convention

5.2 Traits

Trait misnaming problems were mostly dirty suffixes; A given suffix (for instance Test suffix in figure 11) included Traits which are different from the dominant hierarchy using this suffix. The reason behind this problem was that Trait names were composed by systematically placing a *T* letter at their beginning instead of Trait at the end of the name. Figure 11 depicts a sample package illustrating a Trait related false positive where all non-red class boxes inside the Test(s) suffix box are actually traits ending with suffix Test(s) while the concept is represented by the *T* at the beginning of the Trait name.

5.3 Dominant hierarchy

Some detected misnaming problems were due to the algorithm we used to colour the prefix boxes. This algorithm has been explained



Figure 11: A false positive due to Traits special naming convention

in section 3.2. This algorithm considers that a hierarchy is dominant in a given suffix, as soon as the number of its subclasses using this suffix is greater than the number of subclasses using this suffix in other hierarchies. This leads to abnormal situations where for instance, a hierarchy that is defined in a single package with n subclasses using a suffix s, is considered dominant over many other hierarchies with n-1 subclasses. Obviously, in such a situation, the algorithm should have considered that there is no single dominant hierarchy for the considered suffix. We are currently working on the enhancement of the colouring algorithm to address this kind of cases.

5.4 Concept refinement

In Roassal the Shape hierarchy contains element ending not anymore with the Shape prefix but with Arrow. Another example is that the suffix changed from Model to Node in the dependency analyser tool. There the developer consistently specialized the concept of a model to a specific more precise name.

Such behaviour is a false positive in the sense that our approach spotted it as class name inconsistency. However, having such concept refinement is a good practice where the developer refined and created. One way to spot such false positive is that usually a developer creates a new hierarchy and not a single node with a new prefix.

6 RELATED WORKS

This paper addresses the problem of class name consistency against the design hierarchy, based on new approach called *a conceptual blueprint* that captures informations in classes names suffixes and design hierarchy.

The impact of identifiers naming in source code on software comprehension is discussed throw literature. Anquetil and Lethbridge [AL98] proposed a naming convention to achieve a reliable naming requires amongst others that two software artefacts with the same name should implement the same concept. Rajlich and Wilde [RW02] 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 which is frequently necessary in maintenance tasks. There are several different approaches assessing relevance of identifier names in source code, in series of works Marcus and Maletic used LSI (Latent Semantic Indexing) to identify and recover links between documentation and source code [MM03] and to detect conceptual clones [MM01]. Deissenboeck and Pizka [DP05b] proposed a formal model between concepts and names, they presented a tool with " identifier dictionary" to provide maintainers with a guidelines for turning a concept into a name and to better understand the precise concept. We broaden this approach by providing a visual notation that gives an overview of the classes and their inheritence relationships. Lawrie et al.[LMFB06b] carried out an empirical study to asses the quality of code source identifiers, their study involved 100 programmers and indicated that full words as well as recognizable abbreviations lead to better comprehension. Haiduc et al. [HM08] studied several open source programmes and found that 40 percent of the system domain terms were used in source code.

7 CONCLUSION

Code maintenance and evolution rely heavily on code comprehension. Because classes are the building blocks of object oriented applications and the most important abstractions of the application domain concepts, they play a key role in code comprehension. Thus, choosing good class names can help developers perform their tasks more easily. In this paper, We addressed the problem of class names consistency by introducing the conceptual blueprint, a new visualization which aims to assist developers asses the quality of class names in order to facilitate code comprehension and maintenance.

We based our reasoning on the assumption that programers express their intention in class names. More particularly, we restrained ourselves to suffixes in class names and combined this information to the one explicitly expressed in classes hierarchy.

Though the proposed approach is very simple, we demonstrated its usefulness by applying it to 285 packages in Pharo. We then were able to characterise classes misnaming through package shapes and we identified some erroneous packages.

Additionally, Our case study allowed us to identify some false positives, *i.e.*, class names that our approach detected as inconsistent but a closer look revealed their correctness. These false positives are an important aspect of our experiment because they helped us identify the limits of our approach and propose better diagnosis techniques in the future.

Specifically, In order to improve the precision of our approach, we will consider the following points.

- Better location of pertinent information in class names. So long, we have only considered suffixes in class names. Our current experiment showed us that this information can be located in a different part of the class name as in the case of Traits. Sometimes, this information is represented using vocabulary patterns such as using adjectives as we have shown in section 5.1 in the case of Announcements.
- Extension of the approach to consider concept refinement. A current direction we are exploring is the use of ontologies in order for instance, to check for concept refinement between different suffixes or detect part of the speech in class names.
- Validation of the proposed approach on large scale. Systems such as Roassal, Seaside and Moose are potential candidates. Application of the approach to other object oriented languages is to be considered.

REFERENCES

[AL98] Nicolas Anquetil and Timothy C. Lethbridge. Assessing the relevance of identifier names in a legacy software system. In *Proceedings of the 1998*

conference of the Centre for Advanced Studies on Collaborative research, CASCON'98, pages 213–222. IBM Press, 1998.

[AL11] Nicolas Anquetil and Jannik Laval. Legacy software restructuring: Analyzing a concrete case. In *Proceedings of the 15th European Conference on Software Maintenance and Reengineering (CSMR'11)*, pages 279–286, Oldenburg, Germany, 2011.

[CTH16] Tse-Hsun Chen, Stephen W. Thomas, and Ahmed E. Hassan. A survey on the use of topic models when mining software repositories. *Empirical Software Engineering*, 21(5):1843–1919, 2016.

[Dav95] Alan Mark Davis. *201 Principles of Software Development.* McGraw-Hill, 1995.

[DP05a] F. Deissenbock and M. Pizka. Concise and consistent naming [software system identifier naming]. In *13th International Workshop on Program Comprehension (IWPC'05)*, pages 97–106, May 2005.

[DP05b] Florian Deißenbock and Markus Pizka. Concise and consistent naming [software system identifier naming]. In *Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on*, pages 97–106. IEEE, 2005.

[DRGP11] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. Feature location in source code: A taxonomy and survey. In *Journal of Software Maintenance and Evolution: Research and Practice*, 2011.

[HM08] Sonia Haiduc and Andrian Marcus. On the use of domain terms in source code. In *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, pages 113–122. IEEE, 2008.

[HØ09] Einar W Høst and Bjarte M Østvold. Debugging method names. In *European Conference on Object-Oriented Programming*, pages 294–317. Springer, 2009.

[KDG05] Adrian Kuhn, Stéphane Ducasse, and Tudor Gîrba. Enriching reverse engineering with semantic clustering. In *Proceedings of 12th Working Conference on Reverse Engineering (WCRE'05)*, pages 113–122, Los Alamitos CA, November 2005. IEEE Computer Society Press.

[KDG07] Adrian Kuhn, Stéphane Ducasse, and Tudor Gîrba. Semantic clustering: Identifying topics in source code. *Information and Software Technology*, 49(3):230–243, March 2007.

[LD01] Michele Lanza and Stéphane Ducasse. A Categorization of Classes based on the Visualization of their Internal Structure: the Class Blueprint. In *Proceedings of 16th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '01)*, pages 300–311. ACM Press, 2001.

[LMFB06a] D. Lawrie, C. Morrell, H. Feild, and D. Binkley. What's in a name? a study of identifiers. In *14th IEEE International Conference on Program Comprehension (ICPC'06)*, pages 3–12, June 2006.

[LMFB06b] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. What's in a name? a study of identifiers. In *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on*, pages 3–12. IEEE, 2006.

[MM01] Andrian Marcus and Jonathan I Maletic. Identification of high-level concept clones in source code. In *Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on*, pages 107–114. IEEE, 2001.

[MM03] Andrian Marcus and Jonathan I Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 125–135. IEEE, 2003.

[RW02] Václav Rajlich and Norman Wilde. The role of concepts in program comprehension. In *Program Comprehension, 2002. Proceedings. 10th International Workshop on*, pages 271–278. IEEE, 2002.

[Som00] Ian Sommerville. *Software Engineering.* Addison Wesley, sixth edition, 2000.