CEUR-WS.org/Vol-2978/saerocon-paper4.pdf

What is in a Name? An Analysis of Associations Among Java Packaging and Artifact Names

Farshad Ghassemi Toosi, Anila Mjeda

Computer Science Department at Munster Technological University, Cork Campus
Computer Science Department at Munster Technological University, Cork Campus

Abstract
Modern Programming Languages (Object Oriented Languages), are equipped with sophisticated mechanisms to assist developers in organizing the source code. For instance, Java and Python use package names to resolve symbols. In Java, a package is a namespace declared at the top of each class or interface.

There are several reasons for using packages in the source code: 1) Packages can prevent naming conflicts, (e.g., identical class name in two packages is possible with no conflict). 2) Packages can categorize the **relevant** and/or **similar** classes or interfaces in some conceptual and logical containers that assist developers in easier maintenance and a better understanding of the design of the software's architecture. 3) Structured packaging is one of the core components of a clean architecture design. Developers may apply different strategies to structure the packages and these differences have repercussions in the quality and maintainability of the software architecture.

In this work, we run a set of experiments on a number of open-source Java projects and analyse the packaging structures from a source-code structural and artifact (class, method, variable) names perspective. These experiments aim to investigate 1) the existence of any associations between the packaging structure and textual factors (artefact names) of the classes inside the package; and 2) what textual factors (artifact names) tend to be more associated with the package structure. The results of this research indicate that, on average, class names and inheritance (supper class names) tend to be considered as a packaging strategy. The focus on identifying 'naturally' occurring similarities in the packaging of software in the 'wild' is underpinned by the long-term objective to build developer-friendly architecture conformance protocols which help prevent architectural erosion.

1. Introduction

Object oriented programming is underpinned by the idea of creating classes and using objects of those classes for higher reusability and better maintenance. The object oriented programming paradigm is based on bringing related fields and functions/methods together for a particular concept that is called a class. Different objects then can be instantiated from classes with different data and implementation but they all share the same original type, i.e., the class. For example, a class may represent a *car* and its objects can be a *hatchback* or a *sport utility vehicle*. In object oriented programming, methods and fields within a given class are expected to be logically grouped in one container called class.

Some of the modern object oriented languages, including Java and Python, have another mechanism called packaging that lets developers have a higher level of grouping where related classes can be located in a high-level container or module, called a *package*.

Usually, the visual representation of a software's architecture is a graph-like design where the software components are program packages that, in their turn, may contain other packages (hierarchical packages) [1, 2]. In most software architecture design practices, modules or components are seen as a package or a set of packages [3, 4, 5]. Hence, the intuition is that package structure can have a direct impact on the quality of the software architecture. Indeed this intuition has attracted the interest of other researchers of the field such as Ebad et al., [6].

One of the fundamental aspects of an architectural design is to consider the functionalities and interactions between components at different granularities [7] with a view of facilitating work among the components in a package.

Researchers [8, 9, 10, 7] show that a clean software architecture has a direct relation to the structured packaging; furthermore, they show how implicit packaging can cause architectural mismatching. They use the term *unstructured packaging* as a lack of packaging strategy. For instance, all classes would be located in one package or there are random packages, and classes are assigned to them based on no particular strategy. As a result of such packaging structure (or unstructured packaging), there will be several of unrelated classes with no naming and textual relevancies to each other in a package [8]. Naming relevancy, in particular, is important since artefacts (class, method, variable) are meant to be named by

ECSA2021 Companion Volume
farshad.toosi@mtu.ie (F. G. Toosi); anila.mjeda@mtu.ie (A. Mjeda)
https://www.linkedin.com/in/farshad-ghassemi-toosi-428a5852/ (F. G. Toosi); https://www.linkedin.com/in/anila-mjeda-32a5064/ (A. Mjeda)
0000-0002-1105-4819 (F. G. Toosi); 0000-0003-1311-6320 (A. Mjeda)

© 2021 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings (CEUR-WS.org)

developers according to their responsibilities and functionalities.

Java is one of the object oriented languages that offers the packaging mechanism. Every Java class is inside of a package (unless there is no package declared, then the class will be part of the default package). In this work, we are using Java as the language of our case study to answer the following research questions:

1. Are there any existing associations between the package structure and textual factors within the package?. The textual factors in question include artefact names e.g., class, method and variable names.
2. What type of names and at what granularity tend to have more weight on influencing packaging structure?

By answering the above two research questions, we try to discover the level of textual cohesion among components of each package to understand if there is any textual packaging structure in the project or not, and if so, what type of artefact name has a heavier role.

2. Background

In large programs, it is difficult to have an architecture for a software system that conforms to the system's packaging structure. Object-oriented software, has an inherent affinity for structure such as packages as one of its appealing promises. Albeit, that affinity does not necessarily automatically translate to a structure that is *relevant* to the architecture of the system. This issue has seeded research into improving the packaging structure of the software system. Shaw et al. [10] propose that the potential existing problem with reusing components of a software system is not necessarily due to the bad architectural design but the packaging strategy as well. Shaw et al., in a different work [7], emphasise the importance of a packaging strategy to enforce compatible components to be located in the same package.

The quality of the software architecture depends on several factors; one of which is the applied packaging strategy [8, 9, 10, 7]. The packaging strategy refers to the criteria that is used to combine components in packages.

One of the first empirical studies to investigate the structure of written code [11], relied on static and dynamic analysis (of FORTRAN code) and looked at it at a statement level. Existing research tends to look at improving existing package design, such as through package structure analysis [12], using package cohesion to assess organization and reusability of code [13, 14], or using artificial intelligence algorithms or multi-objective approaches based on remodularization objectives [15].

Additionally, there is considerable research to automatically optimise inter-package dependencies [16]. A review of looking at object-oriented code issues in this space as refactoring opportunities, can be found in [17].

Interestingly for our research, Baxter et al [18] investigated some of the reasons behind the structures and structural relationships in Java code, while Abedeen et al. [16] proposed a set of metrics to assess modularity principles for packages in large legacy systems (namely information hiding, changeability and reusability principles) [19]. Coming up to twenty years ago, Hautus [12] proposed a tool to run a package structure analysis through Java code and highlight potential weak areas to the human with an aim to refactor the source code.

Yet, there is still no standard and unique definition of relevant and/or similar classes and developers might consider different criteria to insert two or more classes into a package. The latter becomes problematically evident when analysing code in the wild. Furthermore, packages typically appear in software architecture documentation as not-dividable components of package diagrams, drilling down within packages and investigate their relevancy validity within, has an added value.

It is exactly this gap that is the focus of this research. Indeed, the research reported in this paper represents the initial steps into identifying relevancy (through similarity) factors within packages (or architecture components) with a long term view of building developer-friendly architecture conformance protocols so as to prevent architectural erosion.

3. Experiment Design

In this work, six open-source Java projects are under study and their details are represented in Table 1.

The experiment tries to find whether there are factors that can define the relation within the members of each package or not. It is worth noting that the factors are mostly textual factors (e.g., artefact names) and not the functional factors (e.g., the functionality of the artifacts) unless the functionality of the artifacts is reflected in their names. The details of these factors are discussed in Section 3.2.

The logic of the experiment is as follows:

1. All classes are put in a pool without considering the package structure (the left bottom rectangle in Figure 1).
2. Pairwise similarity between every pair of classes is calculated based on some similarity factor (see Section 3.2).
3. A clustering technique is applied on the members of the class pool and P clusters are generated (P is the number of packages in the project).

Table 1
Six Open-Source Java projects.

Name	#classes	#packages	Details
JHotDraw	730	64	Visualization tool, MIT license.
Galaxy	39	17	Galaxy Artifacts is an opensource and freeware 4x game, written in Java.
JavaFX	38	9	JavaFX is a cross platform GUI toolkit, MIT licence.
JavaParserCore	516	29	Java parser tool, LGPL license
JavaParserSymbol	167	21	Symbol solver tool, Apache License.
Jung	227	14	Visualization tool, open source, Jung licence.

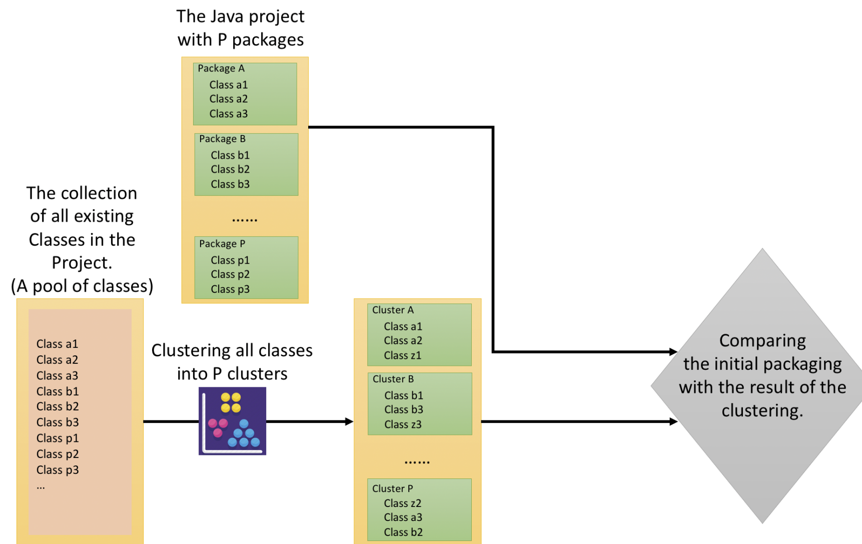

Figure 1: The high-level picture of the proposed model for comparison.

4. The clustering result (the right bottom rectangle in Figure 1) is compared to the package structure of the project (the top rectangle in Figure 1), where each package can be seen as an existing cluster.

Figure 1 shows the general flowchart of the experiment. All the original packages in the system are also seen as a cluster of classes and the objective is to compare the existing packaging to the one arrived at by the proposed clustering algorithm.

3.1. Comparison Analysis

All the experiments in this work are at source-code level and focus on three different types of artefact names: 1) Class names, 2) Method names and 3) Variable/Field names. The comparisons are based on textual/term comparison. Therefore, a simple pre-processing step is required prior to the actual comparison on each name (i.e., Class name, Method name and Variable/Field name). Each name will be converted to some simple-names after the pre-processing. The following list indicates the required actions for pre-processing.

- Camel Case removal. E.g., StudentGrade → Student Grade (StudentGrade as a name is converted to two simple-names: Student and Grade)
- Snake Case removal. E.g., Employee_tax → Employee tax
- Digits removal. E.g., distance100km → distance km, salary100k → salary (Note, words with one character are ignored).
- All lower case. PensionCalculator → pension calculator

3.2. Comparison Factors

As mentioned earlier, the pool of classes is grouped via a clustering algorithm. Clustering algorithms work based

on a similarity or dissimilarity matrix where the similarity/dissimilarity between every pair of entities (classes in this case) is known. Therefore, a similarity needs to be defined between every two classes. Each Java class has several different features and characteristics such as the class name, the method names within the class, field names and many more. In this work, we make use of nine different features of each class and use them as similarity factors for the clustering algorithm. The nine different factors that are examined are as follows:

- Class Names. Two classes are compared according to their names, (**CN**).
- Outgoing Methods. Two classes are compared according to their outgoing method names, (**OM**).
- Incoming Methods. Two classes are compared according to their incoming method names, (**IM**).
- Field Declaration. Two classes are compared according to their declared fields names, (**FD**).
- Variable Accessed. Two classes are compared according to their accessed variables' names, (**AV**).
- Outgoing Class. Two classes are compared according to the class names where they were instantiated, (**OC**).
- Incoming Class. Two classes are compared according to their instantiated class names in them, (**IC**).
- Class Methods Names. Two classes are compared according to their method names, (**CM**).
- Supper Class Names. Two classes are compared according to their supper class names, (**SC**).

Each Java program is analysed and nine different types of information (mentioned earlier) are extracted. In order to extract the details from the Java projects, a Java parser is employed. Among different choices of parsers, JavaParser [20] was selected due to its simplicity in implementation and high reputation.

3.2.1. Class Names

Class Names (**CN**) is the first factor that is used for comparison. Two classes are said to be similar if their names are similar or in other words, if they share some simple-terms. Figure 2 has two packages and each package has two classes. A set of simple-terms is generated for each class in the project:

1. $Circle_area$ class: $\{circle, area\}$.
2. $DrawCircle$ class: $\{draw, circle\}$.
3. $Colors$ class: $\{colors\}$.
4. $SurfaceCircle$ class: $\{surface, circle\}$.

The first class has a degree of similarity with the second class and the fourth class as they share $circle$. Likewise, the second class and fourth class have a degree of similarity while the third class is not similar to any class.

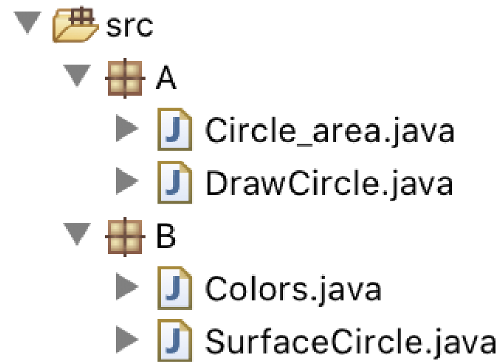

Figure 2: Two packages with their classes.

3.2.2. Outgoing Methods

Outgoing Methods (**OM**) is the second factor that is considered to measure the similarity between two classes. For class *A*, all the methods that are called from class *A* in the project are collected and their names are preprocessed so a set of simple names is generated for class *A*. A similar process is repeated for Class *B*.

Figure 3 shows two classes with their methods and the callee (outgoing) methods inside of them. The set of simple names that can be extracted for Class A based on their callee methods is $\{green, circle, area\}$ and the set of simple names for Class B is $\{green, surface, black, white\}$. There is one simple term common within these two sets, therefore, a degree of similarity exists within Class *A* and Class *B*.

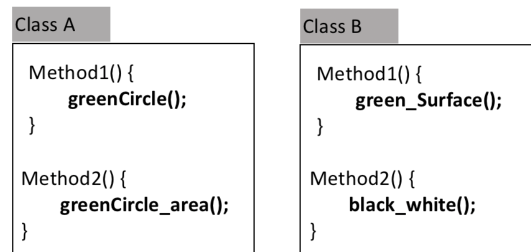

Figure 3: Two Classes with their callee methods.

3.2.3. Incoming Methods

Incoming Methods (**IM**) is the other selected factor to measure the similarity between two classes. This factor, similar to the last one, works based on the method calls. Two classes are said to be similar if their contained meth-

ods are called by methods with similar name (common simple terms).

3.2.4. Field Declarations

Field Declaration (**FD**) is another selected factor and it measures the similarity between classes based on declared fields within the class. Therefore, two classes with similarly declared field names are considered similar. Figure 4 shows two classes with their declared fields. Class A has the following set of simple-terms extracted from its declared fields $\{circle, color, full, area\}$ and Class B has the following: $\{surface, shape, color\}$. Therefore, Class A and B are similar due to the existing of $color$ in both sets of simple terms.

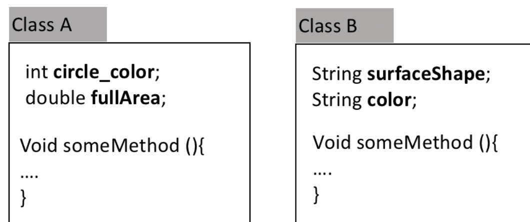

Figure 4: Two classes with their declared fields.

3.2.5. Accessed Variables

Variable Accessed (**AV**) is the other factor we use to measure the class similarities. Two classes are considered similar if they are accessing variables/fields with similar names.

3.2.6. Outgoing Classes

The next factor to measure the package similarity is Outgoing Class names (**OC**). The characterization of being an *Outgoing Class* is a subjective role for a class. Having two classes (Class A and Class B), Class B is said to be an outgoing class for Class A, if Class B is instantiated in Class A. Figure 5 shows two classes, each class has two methods and each method instantiates another class. The name of the instantiated classes for each class are extracted, pre-processed and compared. Class A contains the following set of simple terms extracted from instantiated classes: $\{large, circle, oval\}$ and the set associated with Class B is: $\{large, circle, oval, green, color\}$. As shown in Figure 5, three simple terms are common within these two sets: $\{large, circle, oval\}$. Therefore, Class A and B are similar with some degree. More details of how the degree of similarities is taken into account for comparison, will be discussed in later sections.

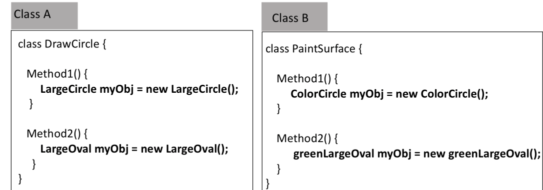

Figure 5: Two classes with their details.

3.2.7. Incoming Classes

Incoming class (**IC**) is another notion we use in this experiment as a similarity factor. Class A is considered as an incoming class for Class B if Class B is instantiated in Class A. In Figure 6 *DrawCircle* is the incoming class for *PaintSurface* class. Two classes are said to be similar if their classes are instantiated with the same class or classes with similar names.

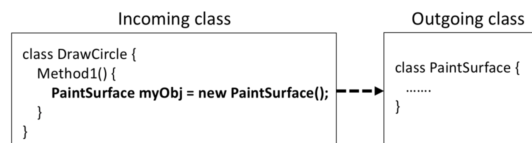

Figure 6: Two classes, one instantiates the other one.

3.2.8. Class Methods

The other employed factor in this work is method name (**CM**). Two classes are considered similar if they have methods with similar names. Figure 7 shows two classes with their contained methods. Class A has the following set of simple-names extracted from method names: $\{paint, surface, get, color\}$ and class B has the following set: $\{color, circle, oval, green, large\}$. Since there is one term common in both sets, therefore, Class A and B are similar with some degree.

3.2.9. Supper Classes

Classes are also compared by their supper classes. For each class, all the super class names are collected, pre-processed and a set of simple-terms is generated. Similar to other similarity factors, the common simple-terms for each pair of classes is an indication of the degree of similarity. In Figure 8, there are two classes with some super classes for each. The set of simple-terms for A is: $\{shape, geometry\}$ and for B is: $\{square\}$. Since there is no common simple-term in these two sets, there is also

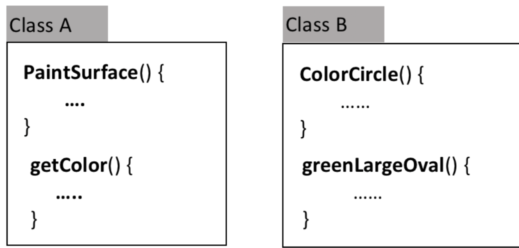

Figure 7: Two classes with their contained classes and methods.

no similarity between these two classes based on this factor.

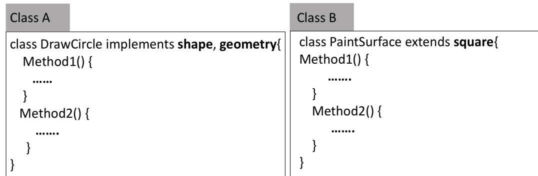

Figure 8: Two classes with their supper classes.

4. Clustering

Clustering is a task of splitting individuals into a number of groups or clusters where the members of a cluster are more similar to other members of the same cluster than the members of other clusters. In this experiment, classes are considered as individuals, therefore, classes with more similarity would be clustered in one group. As mentioned in previous section, there are nine different similarity factors considered in this work, therefore, for each Java project, clustering runs nine times, each time with a different factor. The goal is to measure how much a given factor, as a similarity criteria, conforms to the existing packaging in the system.

Clustering is an unsupervised learning technique that has applications in many different fields and domains. *K-Means* [21], *Affinity propagation* [22], *DBSCAN* [23] and *Spectral Clustering* [24] are a number of clustering algorithms and the choice of algorithm depends on the nature of the data.

4.1. Spectral Clustering

In this experiment, we employ *Spectral Clustering* [24] to cluster the pool of classes (see Figure 1). Spectral Clustering algorithm is based on eigendecomposition calculation and is more suitable for a set of individuals where connectivity relations (e.g., similarity between two individuals) can be defined between them. Unlike other clustering algorithms (e.g., K-Means), Spectral Clustering, requires the relations/similarity between individuals to be computed as a matrix in advance and eigendecomposition can be applied on that matrix. Therefore, *Spectral Clustering* was found a good fit to be the clustering algorithm in this work. As seen in the previous section, each class gets a set of simple-terms (based on the applied similarity factors); the number of common simple-terms among two sets from two classes is considered as the measure of relations/similarity between two classes. Therefore, a $N \times N$ (N is the number of classes) similarity matrix should be created for Spectral clustering.

Spectral Clustering, like most other clustering algorithms, requires to know the number of clusters/groups in advance. As shown in Figure 1, the number of clusters for the applied clustering algorithm is P where P is the number of existing packages in the project. Once Spectral Clustering returns P clusters of similar classes (based on a given similarity factor), one can compare those P clusters against the existing P packages in the system.

4.2. Clustering vs Packaging

The objective is to analyse the individual similarity factors and see how much each of them conform the packaging structure. To do this, the clustering that is resulted from each factor needs to be compared against the packaging structure. Since the clustering is done on P clusters (P is the number of packages in the project), therefore, there are two sets of groups where each set contains P number of groups of classes. In order to measure the similarities between two sets of groups, we make use of *Normalized Mutual Information* [25] technique from SKlearn in Python. *Normalized Mutual Information* measures the similarity between two clusterings [26] and returns a value between 0 to 1. Given two clusterings by two different techniques, *Normalized Mutual Information* specifies how much these two clustering are correlated. Figure 9 shows two clusterings where each clustering has three clusters with their members. As it is shown, there are some differences between the results of these two techniques. For instance, the first cluster of PS contains a_1, a_3 and a_3 and the first cluster of CR contains a_1, a_3 and z_1. The degree of similarity between the results of these two techniques by *Normalized Mutual Information* is 0.2804.

5. Evaluation

In this experiment, six different Open Source Java project are analysed (see Table 1). For each project, nine differ-

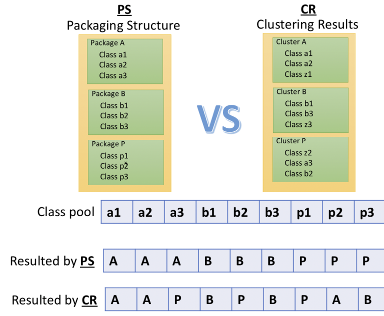

Figure 9: Two different clusterings.

ent similarity factors are separately employed to apply a clustering technique and compared against the packaging structure in the system. The nine factors are fully described in section 3.2.

5.1. Results and Discussion

In total, there are 54 + 6 experiments performed. The first 54 experiments are for 6 projects and for each project 9 individual similarity factors are tested. We run an extra experiment for each project where the similarity factor is the accumulative of all the 9 individual factors.

Figures 10 to 15 show the percentage similarity between the applied clustering technique (Spectral Clustering) and the packaging structure.

The very first observation from all the results indicates the association between class names and packaging. Except for the *Java Parser Core* project, the class name has the highest impact on the packaging. Even for *Java Parser Core*, the class name comes in second-highest score. The other observation that can be realized from all diagrams is the association between supper class names and packaging. Except *JavaFX* project and the *Galaxy* project, supper class names are the second 'winners'. Method names for one project (*JavaFX*) have a higher association with the packaging compared to other projects. On the other hand, class instantiation (incoming and outgoing classes) on average has smaller association with the packaging.

As mentioned earlier, six extra experiments are performed to see the impact of overall similarity factors when they are accumulated all together. Table 2 depicts the results for each individual project. On average the *Galaxy* project has a strong naming association with the packaging followed by Java Parser Core and Java Parser Symbol.

Table 2
Accumulated Factors.

Name	Similarity
JHotDraw	0.364
Galaxy	0.818
JavaFX	0.4562
JavaParserCore	0.622
JavaParserSymbol	0.58140
Jung	0.387

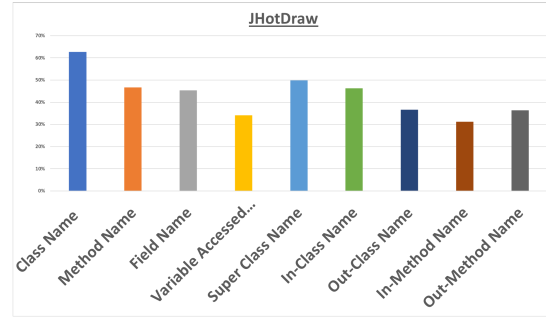

Figure 10: The percentage of each similarity between the applied clustering technique and the packaging structure using 9 similarity factors.

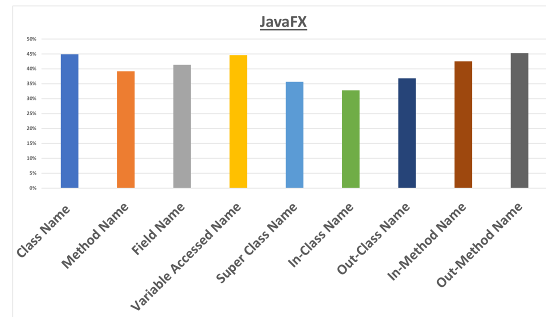

Figure 11: The percentage of each similarity between the applied clustering technique and the packaging structure using 9 similarity factors.

6. Conclusion

In this work, we presented a comparative analysis on six different Java Projects to discover the applied packaging strategy from textual and naming point of view. Our findings (see Table 3) illustrate that there is a textual similarities among components at each package to some extend (*the first research question*). On average, the textual similarity is stronger when class names are chosen as a similarity factor (*the second research question*).

Table 3
Details of all experiments for 6 subject systems. Green indicates the applied factor that shows the highest similarity between the packaging structure and the clustering technique and orange indicates the second highest and blue indicates the third highest.

	Class Name	Method Name	Field Name	Variable Accessed Name	Super Class Name	In Class Name	Out Class Name	In Method Name	Out Method Name
JHotDraw	0.6264	0.4674	0.454	0.3415	0.4989	0.4636	0.3658	0.312	0.3638
JavaFX	0.449	0.392	0.4139	0.446	0.357	0.3286	0.368	0.425	0.453
J-P Core	0.6159	0.332	0.392	0.3937	0.635	0.455	0.336	0.393	0.3977
J-P Symbol	0.6783	0.552	0.423	0.4392	0.534	0.3219	0.361	0.452	0.37
Jung	0.458	0.313	0.243	0.254	0.331	0.24	0.162	0.226	0.189
Galaxy	0.803	0.755	0.748	0.7518	0.734	0.713	0.6913	0.689	0.714
Average	0.6051	0.468567	0.44565	0.4377	0.514983	0.42035	0.380683	0.416167	0.414583

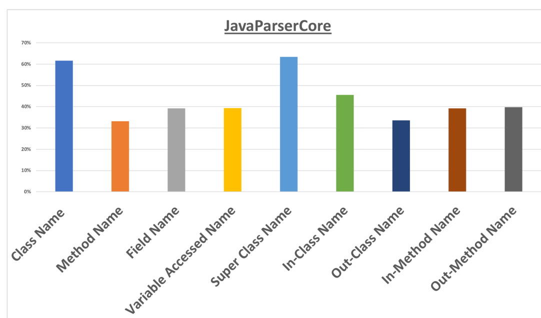

Figure 12: The percentage of each similarity between the applied clustering technique and the packaging structure using 9 similarity factors.

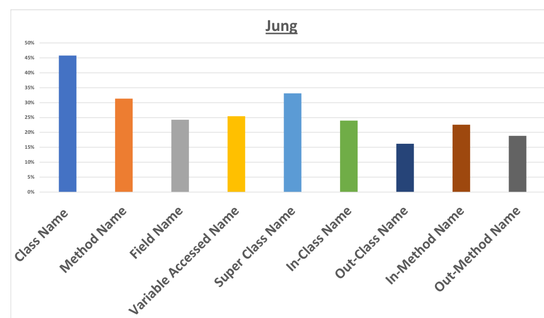

Figure 14: The percentage of each similarity between the applied clustering technique and the packaging structure using 9 similarity factors.

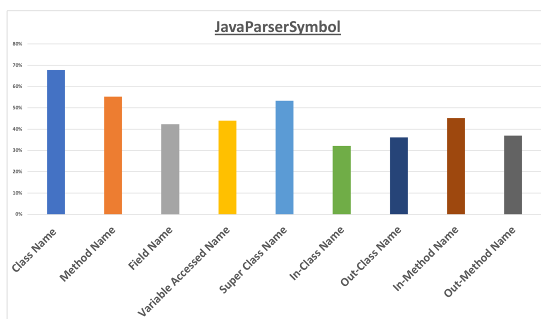

Figure 13: The percentage of each similarity between the applied clustering technique and the packaging structure using 9 similarity factors.

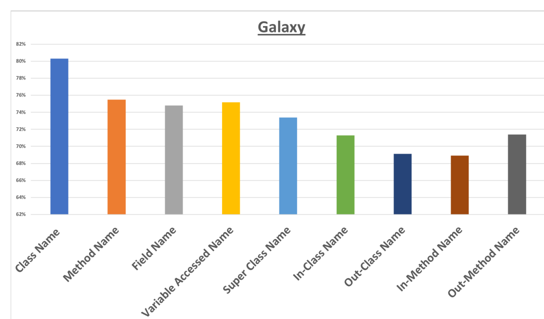

Figure 15: The percentage of each similarity between the applied clustering technique and the packaging structure using 9 similarity factors.

The second factor, after class names, that shows strong similarities among packages' components is, on average, the super class name. This also indicates that most inheritances are within the packages that is potentially an indication for low cohesion and high decoupling between packages. Method names, as the third strong factor, on average show relatively high similarity among the packages' components.

Although we can confirm that there are a couple of patterns common in all projects (similarity of class names),

still almost every project behaves differently. This can be further confirmed by looking at the results in Table 2 where each project shows a different aggregated degree of similarity packaging ranging from 0.36 to 0.81.

Looking from another angle, since class names score high in terms of similarity factors among the contents in a package, they can potentially be used to validate the relevancy within a package or other architectural construct. This claim, however, requires more experimentation on a larger number of subject systems.

This research is only based on the artifact (class, method and variables) names, therefore, the role of the developers' naming style plays an important role in the results.

In future work, we plan to include other similarity factors such as factors that define the functionality of the artefacts. This, with a long term objective of using these 'naturally' occurring similarities in the packaging of software in the 'wild' to build developer-friendly architecture conformance protocols which help prevent architectural erosion.

References

[1] M.-A. Storey, C. Best, J. Michand, Shrimp views: An interactive environment for exploring java programs, in: Proceedings 9th International Workshop on Program Comprehension. IWPC 2001, IEEE, 2001, pp. 111–112.

[2] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, G. Zelesnik, Abstractions for software architecture and tools to support them, IEEE transactions on software engineering 21 (1995) 314–335.

[3] J. Veit, Modules, Components, and Elements – Software Architecture Terms explained (2021). URL: https://dev.to/jessica_veit/modules-components-and-elements-software-architecture-terms-explained-g59.

[4] Tutisani, Modular Software Architecture - Tutisani Consulting, 2021. URL: https://www.tutisani.com/software-architecture/modular-software-architecture.html.

[5] J. T. Taylor, W. T. Taylor, Software architecture, in: Patterns in the Machine, Springer, 2021, pp. 63–82.

[6] S. A. Ebad, M. Ahmed, Investigating the effect of software packaging on modular structure stability, Computer Systems Science and Engineering 34 (2019) 283–296.

[7] M. Shaw, D. Garlan, Formulations and formalisms in software architecture, in: Computer Science Today, Springer, 1995, pp. 307–323.

[8] Vasiliy, 5 Most Popular Package Structures for Software Projects, 2020. URL: https://www.techyourchance.com/popular-package-structures/.

[9] R. C. Martin, J. Grenning, S. Brown, Clean architecture: a craftsman's guide to software structure and design, Prentice Hall, 2018.

[10] M. Shaw, Architectural issues in software reuse: It's not just the functionality, it's the packaging, in: Proceedings of the 1995 Symposium on Software reusability, 1995, pp. 3–6.

[11] D. E. Knuth, An empirical study of fortran programs, Software: Practice and experience 1 (1971) 105–133.

[12] E. Hautus, Improving java software through package structure analysis, in: IASTED International Conference Software Engineering and Applications, 2002, pp. 1–5.

[13] V. Gupta, J. K. Chhabra, Package coupling measurement in object-oriented software, Journal of computer science and technology 24 (2009) 273–283.

[14] P. J. Kaur, S. Kaushal, A. K. Sangaiah, F. Piccialli, A framework for assessing reusability using package cohesion measure in aspect oriented systems, International Journal of Parallel Programming 46 (2018) 543–564.

[15] A. Prajapati, J. K. Chhabra, Madhs: Many-objective discrete harmony search to improve existing package design, Computational Intelligence 35 (2019) 98–123.

[16] H. Abdeen, S. Ducasse, H. Sahraoui, I. Alloui, Automatic package coupling and cycle minimization, in: 2009 16th Working Conference on Reverse Engineering, IEEE, 2009, pp. 103–112.

[17] J. Al Dallal, Identifying refactoring opportunities in object-oriented code: A systematic literature review, Information and software Technology 58 (2015) 231–249.

[18] G. Baxter, M. Frean, J. Noble, M. Rickerby, H. Smith, M. Visser, H. Melton, E. Tempero, Understanding the shape of java software, in: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, 2006, pp. 397–412.

[19] H. Abdeen, S. Ducasse, H. Sahraoui, Modularization metrics: Assessing package organization in legacy large object-oriented software, in: 2011 18th Working Conference on Reverse Engineering, IEEE, 2011, pp. 394–398.

[20] JavaParser.org, JavaParser - Home, 2021. URL: "https://javaparser.org".

[21] J. A. Hartigan, M. A. Wong, Ak-means clustering algorithm, Journal of the Royal Statistical Society: Series C (Applied Statistics) 28 (1979) 100–108.

[22] K. Wang, J. Zhang, D. Li, X. Zhang, T. Guo, Adaptive affinity propagation clustering, arXiv preprint arXiv:0805.1096 (2008).

[23] K. Khan, S. U. Rehman, K. Aziz, S. Fong, S. Saras-

vady, Dbscan: Past, present and future, in: The fifth international conference on the applications of digital information and web technologies (ICADIWT 2014), IEEE, 2014, pp. 232–238.
[24] J. Liu, J. Han, Spectral clustering, in: Data Clustering, Chapman and Hall/CRC, 2018, pp. 177–200.
[25] R. Koopman, S. Wang, Mutual information based labelling and comparing clusters, Scientometrics 111 (2017) 1157–1167.
[26] A. F. McDaid, D. Greene, N. Hurley, Normalized mutual information to evaluate overlapping community finding algorithms, arXiv preprint arXiv:1110.2515 (2011).