Extracting Software Modules as Communities

Cezar Sas
Bernoulli Institute
University of Groningen
Groningen, Netherlands
c.a.sas@rug.nl

Andrea Capiluppi
Bernoulli Institute
University of Groningen
Groningen, Netherlands
a.capiluppi@rug.nl

***Abstract*—Component Based Software Engineering (CBSE) is a development discipline based on the availability of software components, that are described and indexed for internal or external, present or future, reuse. Although the creation of reusable components is requested to be designed from scratch, this is often time consuming and expensive. An alternative is to extract such components from pre-existing OO software.**

In this work, we compare two different community detection algorithms to perform components extraction from existing software. Considering 'components' as 'communities', the aim is to evaluate how independent, yet cohesive, the components are, when extracted by community detection algorithms.

Using a small sample of 3 Java systems, we show how the components can be extracted based on structural information. Furthermore, we consolidate the extracted components using semantic information, to ensure their cohesion. We use three document representation techniques to evaluate the internal cohesion of components.

The results show that both algorithms perform well with each having their own strengths. Leiden extracts less cohesive, but better separated, and better clustered components that depend less on similar ones. Infomap, on the other side, creates more cohesive, slightly overlapping clusters that are more likely to depend more on other semantically similar components.

***Index Terms*—Components Identification, Community Detection, Components Semantic Analysis**

I. Introduction

CBSE is an alternative to Object-Oriented (OO) development that aims to make it easier to develop easily reusable and understandable software by using components [1]. In order to become useful, CBSE has to produce a repository containing enough components, so that developers do not have to rewrite them from scratch [2], [3]. An interesting new development for CBSE has been the transition from *monolithic* applications to *microservices*-based architectures [4].

An alternative to pre-developed and designed components is to extract them from pre-existing software. As an example of this approach, [5] proposed an architecture recovery approach called ROMANTIC, helped by human experts. Similarly, using particle swarm optimization, [6] proposed a method that uses a meta-heuristic search-based clustering. Frequent usage patterns propelled instead the search heuristics in [7].

Other papers, although not necessarily focusing on component reuse, provided algorithms for component extraction: [8], [9] proposed a metric called MQ to measures edge density of graphs, to help in the module selection. The use of graph clustering was often used for the same purpose in later publications [10], [11].

The aim of this paper is to evaluate how well-formed the components are when they are automatically extracted by community-detection algorithms [12]. Based on the extracted 'communities as modules', we aim to test the cohesiveness of those communities (of features) in the context of the overall software system. For the stated aim, we analysed a small sample of three OSS Java projects (*antlr4*, *avro*, and *openj9*). Those come from a larger sample[1] of some 700 Java projects, that will be analyzed as part of the first author's PhD work.

The two objectives of this paper are (i) to evaluate the effectiveness of extracting *structure*-informed modules from a software system, and (ii) to detect whether the extracted modules are semantically cohesive.

In order to assess the first objective, we used two community detection algorithms, *Infomap* [13] and *Leiden* [14]. The second objective is checked in two ways: at first we evaluated whether there is low cohesion between different modules (e.g., semantic separation); additionally, we also checked whether the identified modules are cohesive *per-se*. As part of the replication package, we make our source code publicly available[2].

II. Background

In the following section, we introduce the three steps in this work: how we extracted the structural dependencies (II-A), how we detected the software modules as 'communities' (II-B) and how we assessed the cohesiveness of the modules (II-C).

A. Structural dependencies

The first step of our approach is the extraction of the dependency graph for each project in our sample. Using the Arcan [15] tool, we obtained the nodes and the edges describing the dependencies between classes, where the edge weight is the number of uses [16] of one class of the other. We obtained the number of vertices and edges described in Table I. For example, in *antlr4*, we have a dependency between *org.antlr.v4.automata.ParserATNFactory.java* and *org.antlr.v4.tool.LexerGrammar.java* as the former imports

[1] *Awesome Java* is a GiHub project that aggregates several hundreds of curated Java projects, available at https://github.com/akullpp/awesome-java

[2] https://github.com/SasCezar/ComponentSemantics/tree/BENEVOL

the latter. We also note the weight of such dependency (in this case, 1), that is used by Infomap and Leiden to create communities.

TABLE I: Size of the analyzed projects in terms of vertices (documents) and edges (dependency between documents).

	antlr4	avro	openj9
# Nodes	384	292	910
# Edges	2,386	1,175	3,865

B. *Community detection via graph clustering*

The second step of our approach uses the relationships identified by the structural dependencies as input for the Community Detection (CD) algorithms. CD (or graph clustering) approaches have been used for various tasks in the software engineering domain besides component identification. Examples include aiding the transition from monolithic applications to microservices architectures [4], architecture reconstruction [17], and refactoring of software packages [18].

In this step we compare two different types of community detection algorithms, Infomap and Leiden: both are popular and effective clustering tools that have been deployed in different domains. Table II summarises, for the three systems, the number of identified components by the two algorithms.

a) Infomap [13]: is an algorithm based on random walks and Huffman coding [19]. They frame the problem of partitioning the graph into communities as a code optimization problem. To find an efficient code, they look for a partition that minimizes the expected description length of a random walk in the graph. In the resulting partition, a community is made of a group of nodes among which information flows quickly and easily.

b) Leiden [14]: is an improvement of Louvain [20], with Leiden guaranteeing well connected communities. They maximise the community modularity by isolating modules with the most dense internal connections (i.e., higher cohesion) and the least amount of connections between outside nodes (i.e., lowest coupling). Leiden (and so Louvain) starts by assigning different communities to each vertex. They merge the nodes iteratively, based on the gain in modularity (if there is no gain, the node remains in its current community). The procedure stops when, moving a node into another community, there is no further gain in the modularity.

Table II summarises the number of components (e.g., the communities) as extracted by the Infomap and Leiden algorithms. As visible, Leiden is consistently extracting more (but smaller) components, whereas Infomap can aggregate more classes in the same (larger) component.

TABLE II: Nr of components extracted by Leiden and Infomap

	antlr4	avro	openj9
Leiden	7	12	26
Infomap	3	6	16

C. *Semantic cohesion from source code*

The third step of our approach is based on the extraction of semantic cohesion and separation within the source code. We used two popular natural language processing techniques, TF-IDF and BERT.

a) TF-IDF: is a measure that reflects the importance of a word in a document based on a collection of documents. It is the ratio between the number of times the word appears in the document and the amount of information that word provides. The latter is obtained by dividing the total number of documents by the number of documents containing the term and then taking the logarithm. TF-IDF does not capture the position in the text, semantics, co-occurrences in different documents, meaning that the created representations are useful as lexical level features. However, unlike BERT, its vocabulary adapts to the collection. The final representation of a collection (in our case a software project) is a document-term matrix, where the columns are the terms, and the rows documents.

b) BERT [21]: is a state-of-the-art neural network model for learning dense vector representation of words (embeddings). It makes use of the Transformer [22], an encoder-decoder architecture that uses attention [23] to learn contextual relations between words in a text. BERT representations are contextual, meaning that word's representation is based on the other words in the sentence (context). We can obtain an embedding of a sentence or document by taking the vector of the first (special) token. In this paper we used the Hugging Face's *bert-base-uncased* pre-trained model [24]. Although this pre-trained model has a vocabulary optimized for natural text, we also adopted a subword segmentation technique called WordPiece [25] (e.g. *playing* becomes *play* + *#ing*). While this still provides additional bits of information, some technical words are nevertheless poorly tokenized (e.g. *servlet* becomes *ser* + *#v* + *#let*).

III. Extracting the documents embeddings

The extraction of features representing the documents is performed in three different ways:

a) Embeddings from package- and class-names: The first feature extraction uses BERT to perform an embedding of the cleaned text of the package and class name. The cleaning consists of splitting the package name by the dots; following, we remove the first two elements of the list as they represent the organization that developed the software. The final step is to split the camel case strings into words and remove java keywords. For example *org.antlr.v4.tool.LexerGrammar* becomes *"v4 tool lexer grammar"* which is used as input to BERT to create the embedding.

b) Embeddings from source code: The second feature extraction method uses BERT again; however, this time, we perform the embedding on the set of identifiers extracted from the source code file. For the extraction of the identifiers contained in the source code, we used the *tree-sitter* parser generator tool[3]. It makes easy to get the identifiers, without

[3] https://github.com/tree-sitter/tree-sitter

keywords, from the annotated Concrete Syntax Tree created using a grammar for Java code. We clean the identifiers as before, and remove common Java terms that do not add much semantically (e.g., '*main*', '*println*', etc). Figure 1 shows an example of the preprocessing.

```
// Input Source Code
import java.util.Scanner;

class SquareArea {
   public static void main (String[] args) {
     System.out.println("Enter Side of Square:");
     Scanner scanner = new Scanner(System.in);
     double side = scanner.nextDouble();
     double area = side * side;
     System.out.println("Area of Square is: " + area);
   }
}

// Output Identifiers
['area', 'side', 'next', 'demo', 'square', 'system']
```

Fig. 1: Input and output for the extraction of the identifiers set of the source code.

c) TF-IDF bag of words from source code: The last feature extraction method that we use is a modification of the second. Instead of using BERT for the creation of the document embedding, we use TF-IDF. For each of the analysed systems, we limit the vocabulary size to the top $1,000$ terms.

IV. Experiments and Results

Our experiments are designed to evaluate different community detection algorithms in terms of semantic cohesion of components and separation between different components.

A. Semantic Cohesion

We first evaluate the internal semantic cohesion of components using the cosine similarity. This should indicate how cohesive are the components found by the community detection algorithm. Given two vectors, the cosine similarity ranges between $+1$, and -1. With $+1$ meaning that the two vectors have the same orientation, -1 that their meaning is opposite, and 0 that they are orthogonal.

We perform a pairwise similarity between all the vectors describing the nodes in the community and then take the average. The average of the components mean similarity for each project is presented in Table III. We can notice that there is a large difference in the similarity scores between BERT and TF-IDF. This is also present in the other analysis. This is due to the different type of representation obtained by the two methods, TF-IDF is sparse and only syntactic, making it more difficult for the latter to obtain high similarity scores for documents that contain different words with similar meaning.

The analysis shows that the components extracted from both Leiden and Infomap have a high cohesion with each of the three features. While both have a high cohesion, Infomap gives more cohesive components 5 out of 9 times.

TABLE III: Average cohesion of components

Project	BERT				TF-IDF	
	Package		Document			
	Leiden	Infomap	Leiden	Infomap	Leiden	Infomap
antlr4	0.8672	0.8804	0.8932	0.9055	0.3096	0.3661
avro	0.8171	0.8487	0.9197	0.9256	0.4617	0.4491
openj9	0.8767	0.8645	0.9097	0.9043	0.4466	0.4371

B. Semantic Separation

The extracted components should have low semantic similarity. This is measured by the cosine similarity between the representing vectors of each community. Since each component is composed of many nodes, each with its own embedding, the community vector is obtained aggregating in a single vector all the information, and using the mean function.

Table IV presents the results for the semantic separation. Leiden has lower values in 7 out of 9 cases when compared to Infomap. When contextualized with the results of the semantic cohesion, we have a similar result as before. The average external separation is smaller than the internal 2 out of 9 for Infomap and 3 for Leiden. We can see visually both the semantic cohesion and separation in Figures 2. One of Infomap's components is significantly larger than the others (also in the other projects), and this distorts both the cohesion and separation. For this reason we need to compute a clustering measure, the Silhouette, using the nodes and not an aggregated representation.

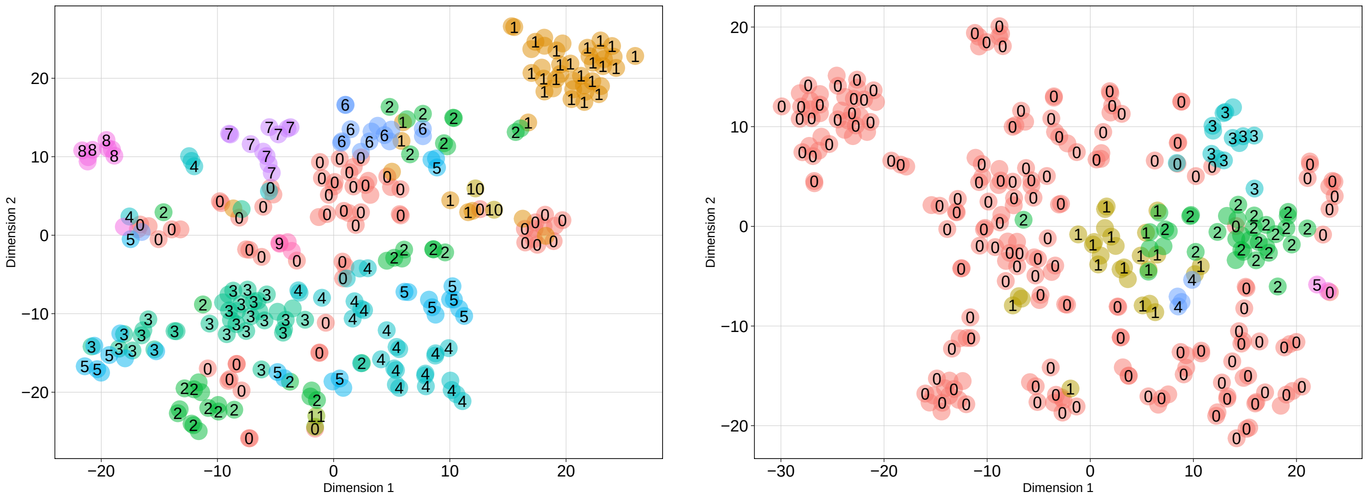
TABLE IV: Average similarity between components

Project	BERT				TF-IDF	
	Package		Document			
	Leiden	Infomap	Leiden	Infomap	Leiden	Infomap
antlr4	0.9384	0.9448	0.9705	0.9729	0.4679	0.5649
avro	0.8677	0.8741	0.9329	0.9545	0.3336	0.4740
openj9	0.8523	0.8421	0.9425	0.9401	0.2256	0.2315

C. Silhouette

The average similarity scores above do not give a global idea of how well the nodes are grouped together semantically. We can get a combined measure of both the internal cohesion and external separation by using the Silhouette coefficient. This is a measure of how similar an object is to its own cluster (cohesion), as compared to other clusters (separation). The values range from $+1$ (i.e., perfect cluster assignment) to -1 (i.e., each element is assigned to the wrong cluster). A value of 0 indicates overlapping clusters. The silhouette can be computed using various distance metrics, like the Euclidean distance: in the experiments below, we use the cosine distance, defined as $1 - cos\_similarity$.

The silhouette results are presented in Table V. The results give a better overview of the previous analysis. We can see that Leiden gets consistently better results than Infomap (8 times).



(a) Leiden

(b) Infomap

Fig. 2: *avro*'s TF-IDF features reduced to two dimensions. The spatial position of the node represents its semantic, the color (and label) the component it belongs to. Leiden's communities 2, 5, and 0 are scattered, while the others are more cohesive. In Infomap community 0 spans the entire semantic space, while communities 3, 1, and 2 are cohesive.

This is caused by more semantically sparse components as it can also be noticed in Figure 2b.

TABLE V: Silhouette scores for the extracted communities

Project	BERT				TF-IDF	
	Package		Document			
	Leiden	Infomap	Leiden	Infomap	Leiden	Infomap
antlr4	+0.0707	+0.0750	+0.0152	+0.0084	+0.1028	+0.0783
avro	+0.0292	−0.0420	−0.0069	−0.1385	+0.1263	+0.0470
openj9	+0.0497	−0.0104	−0.0502	−0.0882	+0.1184	+0.0585

D. *Correlation Between Dependencies and Similarity*

The final analysis measures the correlation between the number of dependencies connecting the components and their semantic similarity. We would like this score to be slightly negative, since having similar components with high dependency might indicate that the algorithm splits a component into two, or that in the project there are two components that perform different parts of the same tasks. A slightly negative value instead, will reflect that the components interact less with more similar ones.

For this experiment, we measured first the number of edges between each community in a pairwise fashion. In this case, we consider the edges as undirected and use the total amount of dependencies between communities. Moreover, we measured the semantic similarity between components, and, as for the semantic separation, we used the mean of the embeddings of each vector as the component representation.

We compute the correlation of the two variables using Pearson's r. The results for each project are shown in Table VI. For both Leiden and Infomap there is no significant correlation between the semantic similarity of components and their level of dependencies. However, Infomap has two negative values, even if overall it is lower only 4 times, and only on the smaller projects. This is caused by it having a large component with many nodes that span across a wide semantic space.

TABLE VI: Pearson's r for the number of dependencies between components and the semantic similarity.

Project	BERT				TF-IDF	
	Package		Document			
	Leiden	Infomap	Leiden	Infomap	Leiden	Infomap
antlr4	0.1188	0.0049	0.2299	0.2681	−0.0150	−0.1108
avro	0.2762	0.1145	0.2065	0.2361	0.0705	−0.0405
openj9	0.1614	0.1766	0.1249	0.1472	0.1263	0.1813

V. Conclusion

In this work we compared two different community detection algorithms for the software components identification task. We evaluated them using three different representation methods and showed the characteristics of each algorithm. Leiden extracts less cohesive, but better separated, and better clustered components that depend less on similar ones. Infomap, on the other hand, creates more cohesive, slightly overlapping clusters that are more likely to depend more on other semantically similar components.

Beside the small sample size, an issue with our approach is that the aggregated scores, and the small differences between the internal and external similarity, give only a shallow idea of what the algorithms identify.

As future work, we plan to qualitatively evaluate each of the extracted components individually, and to asses possible automatic refinements to the content of the components based on their semantics. Furthermore, components should be assigned a topic to make them easily understandable and accessible. Finally, we plan to enlarge the analysis to the entire set of projects contained in the *Awesome Java* curated set.

Acknowledgments

We would like to thank Darius Sas for the helpful discussions, and for providing a beta version of Arcan.

References

[1] G. Booch, *Software Component with ADA*. Benjamin-Cummings Publishing Co., Inc., 1987.

[2] N. Md Jubair Basha and S. A. Moiz, “Component based software development: A state of art,” in *IEEE-International Conference On Advances In Engineering, Science And Management (ICAESM -2012)*, 2012, pp. 599–604.

[3] R. Holmes, *Pragmatic software reuse*. Citeseer, 2008.

[4] G. Mazlami, J. Cito, and P. Leitner, “Extraction of microservices from monolithic software architectures,” in *2017 IEEE International Conference on Web Services (ICWS)*, 2017, pp. 524–531.

[5] S. Chardigny, A. Seriai, M. Oussalah, and D. Tamzalit, “Extraction of component-based architecture from object-oriented systems,” in *Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008)*, 2008, pp. 285–288.

[6] A. Rathee and J. K. Chhabra, “Mining reusable software components from object-oriented source code using discrete pso and modeling them as java beans,” *Information Systems Frontiers*, pp. 1–19, 2019.

[7] ——, “Improving cohesion of a software system by performing usage pattern based clustering,” *Procedia Computer Science*, vol. 125, pp. 740 – 746, 2018, the 6th International Conference on Smart Computing and Communications.

[8] S. Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen, and E. R. Gansner, “Using automatic clustering to produce high-level system organizations of source code,” in *Proceedings. 6th International Workshop on Program Comprehension. IWPC’98 (Cat. No.98TB100242)*, 1998, pp. 45–52.

[9] Y. Chiricota, F. Jourdan, and G. Melancon, “Software components capture using graph clustering,” in *11th IEEE International Workshop on Program Comprehension, 2003.*, 2003, pp. 217–226.

[10] Jong Kook Lee, Seung Jae Jung, Soo Dong Kim, Woo Hyun Jang, and Dong Han Ham, “Component identification method with coupling and cohesion,” in *Proceedings Eighth Asia-Pacific Software Engineering Conference*, 2001, pp. 79–86.

[11] S. Allier, H. A. Sahraoui, and S. Sadou, “Identifying components in object-oriented programs using dynamic analysis and clustering,” in *Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research*, ser. CASCON ’09. USA: IBM Corp., 2009, p. 136–148.

[12] A. Lancichinetti and S. Fortunato, “Community detection algorithms: a comparative analysis,” *Physical review E*, vol. 80, no. 5, p. 056117, 2009.

[13] M. Rosvall and C. T. Bergstrom, “Maps of random walks on complex networks reveal community structure,” *Proceedings of the National Academy of Sciences*, vol. 105, no. 4, pp. 1118–1123, 2008.

[14] V. A. Traag, L. Waltman, and N. J. van Eck, “From louvain to leiden: guaranteeing well-connected communities,” *Scientific reports*, vol. 9, no. 1, pp. 1–12, 2019.

[15] F. A. Fontana, I. Pigazzini, R. Roveda, D. Tamburri, M. Zanoni, and E. Di Nitto, “Arcan: A tool for architectural smells detection,” in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. IEEE, 2017, pp. 282–285.

[16] L. Pruijt, C. Köppe, J. M. van der Werf, and S. Brinkkemper, “The accuracy of dependency analysis in static architecture compliance checking,” *Software: practice and Experience*, vol. 47, no. 2, pp. 273–309, 2017.

[17] I. Şora, G. Glodean, and M. Gligor, “Software architecture reconstruction: An approach based on combining graph clustering and partitioning,” in *2010 International Joint Conference on Computational Cybernetics and Technical Informatics*. IEEE, 2010, pp. 259–264.

[18] W.-F. Pan, B. Jiang, and B. Li, “Refactoring software packages via community detection in complex software networks,” *International Journal of Automation and Computing*, vol. 10, no. 2, pp. 157–166, 2013.

[19] D. A. Huffman, “A method for the construction of minimum-redundancy codes,” *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.

[20] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, “Fast unfolding of communities in large networks,” *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2008, no. 10, p. P10008, oct 2008.

[21] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of deep bidirectional transformers for language understanding,” in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 4171–4186.

[22] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, “Attention is all you need,” in *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*, I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett, Eds., 2017, pp. 5998–6008.

[23] D. Bahdanau, K. Cho, and Y. Bengio, “Neural machine translation by jointly learning to align and translate,” in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015. [Online]. Available: http://arxiv.org/abs/1409.0473

[24] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. L. Scao, S. Gugger, M. Drame, Q. Lhoest, and A. M. Rush, “Huggingface’s transformers: State-of-the-art natural language processing,” *ArXiv*, vol. abs/1910.03771, 2019.

[25] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, L. Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, and J. Dean, “Google’s neural machine translation system: Bridging the gap between human and machine translation,” *ArXiv*, vol. abs/1609.08144, 2016.