Comparing Word-Based and AST-Based Models for Design Pattern Recognition

Sivajeet Chand
Sushant Kumar Pandey
Jennifer Horkoff
Miroslaw Staron
sivajeet@student.chalmers.se
sushant.kumar.pandey@gu.se
jennifer.horkoff@cse.gu.se
miroslaw.staron@cse.gu.se
Chalmers | University of Gothenburg
Sweden

Miroslaw Ochodek
Poznan University
Poland
Miroslaw.Ochodek@cs.put.poznan.pl

Darko Durisic
Volvo Cars
Gothenburg, Sweden
darko.durisic@volvocars.com

ABSTRACT

Design patterns (DPs) provide reusable and general solutions for frequently encountered problems. Patterns are important to maintain the structure and quality of software products, in particular in large and distributed systems like automotive software. Modern language models (like Code2Vec or Word2Vec) indicate a deep understanding of programs, which has been shown to help in such tasks as program repair or program comprehension, and therefore show promise for DPR in industrial contexts. The models are trained in a self-supervised manner, using a large unlabelled code base, which allows them to quantify such abstract concepts as programming styles, coding guidelines, and, to some extent, the semantics of programs. This study demonstrates how two language models—Code2Vec and Word2Vec, trained on two public automotive repositories, can show the separation of programs containing specific DPs. The results show that the Code2Vec and Word2Vec produce average F1-scores of 0.781 and 0.690 on open-source Java programs, showing promise for DPR in practice.

CCS CONCEPTS

• **Software and its engineering → Software implementation planning**.

KEYWORDS

Programming Language Models, Design Patterns, NLP

ACM Reference Format:
Sivajeet Chand, Sushant Kumar Pandey, Jennifer Horkoff, Miroslaw Staron, Miroslaw Ochodek, and Darko Durisic. 2023. Comparing Word-Based and AST-Based Models for Design Pattern Recognition. In *Proceedings of the 19th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE '23), December 8, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3617555.3617873

1 INTRODUCTION

Design patterns (DPs) are typically defined as descriptions of communicating classes that collectively provide a standardized solution to a recurring design issue [6]. DPs were introduced to enhance reusability and maintainability; however, if appropriately selected and implemented they can also boost other software quality characteristics [2], for instance, testability, and scalability. Over time, DPs evolved from generic solutions to be used in standards for large software development. AUTOSAR is one of the examples of such a standard [12]; it is a standardized framework for automotive software development, improving the efficiency, safety, and interoperability of electronic control units.

One of the challenges when recognizing DPs in source code is the fact that they can be implemented differently and can be adapted to fit specific contexts. Manual inspection can be used to identify code fragments that match known design patterns; however, manual recognition is time-consuming and error-prone. As an alternative, automated pattern detection tools have been developed to help detect design patterns in source code. These tools typically use a library of known design patterns and compare the code to identify possible matches. Some popular tools include Java Design Patterns Detector (JDPD), Design Pattern Detector (DPD), and SourcererCC [20, 22]. Recently, researchers have explored the use of machine learning-based approaches for DP recognition (DPR) [23].

In the context of automotive software, using appropriate design patterns becomes essential due to the unique characteristics of the domain, including strict safety requirements and the need for real-time responsiveness. Incorrectly identifying or implementing these design patterns could impact the vehicle's functionality and safety. It is important to recognize specific design patterns that align with safety and real-time performance criteria, as they play a critical role in ensuring the dependable operation of automotive systems. However, several challenges are associated with DPR techniques in automotive software. In this study, working with our industry partner, we found two challenges associated with existing tools and methods for DPR: 1) methods based on machine learning (ML) [8, 23] require a significant amount of labelled training data, and 2) existing methods [20, 22] mainly focus on classical Gang of Four (GoF) Object-Oriented (OO) DPs and programming language dependent, excluding domain-specific patterns.

One way to address these limitations may be to use programming language models (PLMs) for DPR. PLM-based approaches can be capable of capturing the semantic and contextual information inherent in code, as highlighted by Compton et al. [4] and Parthasarathy et al. [16]. By training a language model on a large dataset of unlabelled automotive code examples, we hypothesize that the model can learn to recognize patterns in automotive code. Such a model can be used for DPR across different software components and could detect patterns specific to a particular system or component.

PROMISE '23, December 8, 2023, San Francisco, CA, USA

ACM ISBN 979-8-4007-0375-1/23/12.
https://doi.org/10.1145/3617555.3617873

In this article, we conducted an empirical study to evaluate PLMs as part of an effective DPR tool. We pre-train two existing benchmark PLMs using two automotive codebases. Although we are interested in the long-term in evaluating non-standard DPs, we start by evaluating three standard GoF DPs from publicly available Java-implemented programs. We found that PLMs can help make effective DPR tools. We are addressing the following research question: *RQ-1: To what extent can these PLMs recognize design patterns in Java after pre-training?* Section 2 will discuss the related work, and section 3 will present our suggested approach. Section 4 will analyze results and discussion, and section 5 will refer to the threats to validity. Finally, we will conclude our work in section 6.

2 RELATED WORK

This section provides an overview of the applications of PLMs in software engineering and an overview of the current state of DPR methods. Several recent studies have explored and demonstrated the potential of using PLMs in software engineering tasks. For example, Movshovitz et al. [13] investigated the use of PLM to predict programmer comments from Java source files. Roziere et al. [18] proposed an unsupervised programming language function translation model capable of translating functions across three programming languages and achieved high accuracy. Jung et al. [9] developed a model to automatically generate commit messages using a dataset of 345K code modifications, which showed promise in improving software development collaboration and efficiency. GraLan [14] presented a graph-based statistical language model that suggests code from a source code corpus. The results showed that the API suggestion engine outperformed state-of-the-art approaches.

Several methodologies have been proposed for DPR in source code. Tsantalis et al. [20] proposed a similarity scoring system based on graph vertices that do not rely on pattern-specific heuristics, making extending to novel design structures easier. Xiong et al. [22] incorporated static analysis and inference techniques to improve the accuracy of DPR, achieving higher precision and recall than three other detection approaches, but without considering semantic information. Zanoni et al. [23] combined graph matching and ML-based approaches to propose a DPR model called MARPLE-DPD, which was tested on ten open-source software systems, the approach achieved f-score of 0.85, 0.77, 0.56 for Singleton/Adapter, Decorated, and Composite DPs, respectively. Antoniol et al. [3] proposed a DPR method called DeMIMA, achieving perfect recall and 34% precision when tested on five open-source projects. Parthasarathy et al. [16] utilized TransCoder PLM to detect design patterns across two files, focusing on controller handler industry patterns, achieving the best results after pre-training the model on an industrial codebase. In this work, we evaluate a larger set of patterns using differing PLMs.

3 APPROACH

Our approach includes training two language models—Code2Vec and Word2Vec—on two large automotive software codebases – AndroidAuto and GENIVI. These two PLMs are widely used in different SE applications [4, 13]. The codebases are from the infotainment domain, and both are used in contemporary commercial vehicles. In order to evaluate the DPR ability of these models, we use Singleton, Builder, and Prototype DPs. We fed the extracted embeddings to k-means clustering for prediction of implemented DPs. The replication packages of this study is available here[1].

[1]https://github.com/sivajeet/DPR-using-code2vec-and-word2vec

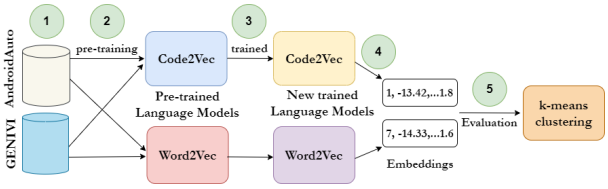


Figure 1: Pre-training with Word2Vec and Code2Vec on Genevi and Android Auto codebases.

3.1 Data Acquisition and Integration

The project's unlabelled training data came from two open-source repositories: Genivi[2][7, 15] and AndroidAuto [3], primarily utilized in vehicles' infotainment systems. These repositories contain a vast amount of Java code that was utilized to train and test the project's models (over 24 million LOC). Genivi [7, 15] is an open-source platform for automotive infotainment systems that provides a standardized software development framework and APIs for building connected car applications. This codebase includes a variety of libraries and components that can be used to build various software modules for automotive use cases. By using the Genivi codebase for training, the project can benefit from the existing implementations of various software modules for automotive use cases. Additionally, the Genivi codebase has been widely used and tested in production environments, which can provide a level of assurance and reliability to the project's models. More recently, this codebase has been utilized in various applications within autonomous software systems, as demonstrated in [7, 15].

AndroidAuto is a mobile application developed by Google that seamlessly integrates a smartphone's features with a car's dashboard screen and entertainment systems. This codebase has also been employed in various app developments [17] and autonomous software for modern vehicles [5]. As a relatively new software codebase, AndroidAuto has gained significant traction in the automotive industry and is often used as a replacement for the Genivi software. We employed this codebase because it contains the latest, high-quality source for automatic software systems. Therefore, investigating the differences in design patterns used by newer software in the same domain is particularly interesting. By training the model on the AndroidAuto codebase, the project can learn and recognize the DPs used in AndroidAuto, which can help identify potential differences in DPs. This can provide valuable insights into the DPs used in different software platforms in the automotive industry and help improve the design of future automotive software systems. This codebase has also been employed in various app developments [17] and autonomous software for modern vehicles [5].

3.2 Data Preprocessing

To pre-train the PLM models, the input data should be pre-processed to a model-understandable form. As our models are to be trained to understand Java code, only files with the ".java" extension have been extracted. Word2vec is a pre-trained model that creates vectors of words. It takes ".txt" extension files as input, so all the ".java" files have been converted to ".txt" files to suit the model. Whereas, Code2vec is a pre-trained model that takes Java files as input. For pre-training Code2Vec, and Word2Vec, 6,250 example java/txt have been put into the test set, 14,323 examples have been put into the validation set, and 23,364 examples have been used to train the model.

[2]https://github.com/GENIVI/
[3]https://android.googlesource.com/platform/packages/services/Car/

Table 1: Training performance metrics for pre-training PLMs using two codebases.

Model	*Accuracy*	*Precision*	*Recall*	*F1-score*
Word2Vec	91%	82.87%	82.20%	82.53%
Code2Vec	90%	80.47%	79.70%	80.08%

3.3 Pre-training of Programming Language Models

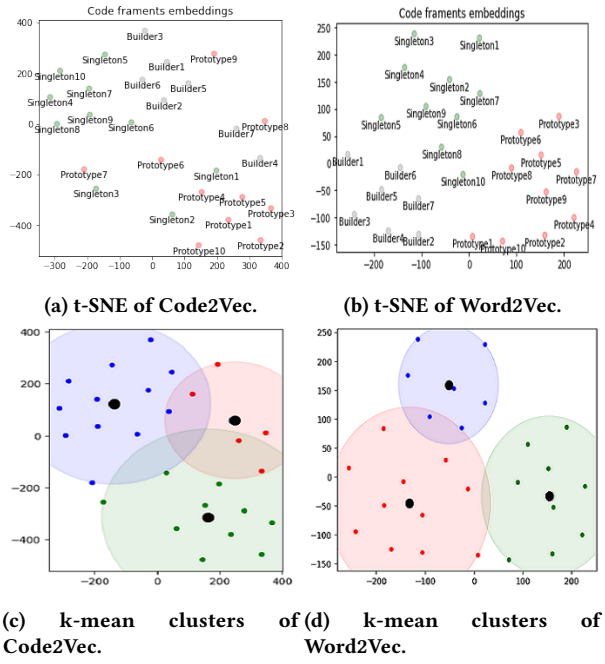
In the pre-training phase, the two codebases, Genevi and Android Auto, were fed into Code2Vec and Word2Vec models, represented as step 1 and 2 in Fig. 1. The training was stopped after 50 epochs (step 3), as the loss versus epoch plot was stable. Next, the models after pre-training were tested by providing Java programs of selected DPs to extract embeddings from the encoder block. The models' behavior over DP in the source code was visualized, as shown in step 4 of Fig. 1.

3.3.1 Word2Vec. Word2Vec is widely adopted for acquiring word embeddings via neural networks. These embeddings constitute vectorized depictions of words that appear in natural language. The Word2Vec technique generates a vector of a particular word by incorporating the neighboring words within its vicinity. These vectors encapsulate vital details about the relationships between a given word and other words that co-occur within the corpus. In our case, we have leveraged the skip-gram model [11] of Word2Vec, which takes the target word as the input and forecasts the surrounding context words.

In our current implementation, we require embedding a .txt file (which was produced after preprocessing java files). To achieve this, we have treated each source code word as if it were a natural language word and analyzed the neighboring words to generate a vector representation for each word. The resultant vectors for each word in the particular .txt file were then averaged to obtain a global vector representation. For instance, consider a code file with the words "`Print (Hello World)`" where the 3-dimensional vector for "`Print`" is [0.256411, 0.652133, 0.432324], "`Hello`" is [0.545321, 0.654456, 0.345235], and "`World`" is [0.432966, 0.972121, 0.657321]. In this scenario, the average vector for the code file is calculated as [(0.256411 + 0.545321 + 0.432966)/3, (0.652133 + 0.654456 + 0.972121)/3, (0.432324 + 0.345235 + 0.657321)/3]. These averaged vectors are employed as the vector representations for each text file in the dataset.

The model was trained using 50 epochs with a 23,364 source code files corpus. After careful evaluation, the model at epoch 39 was selected as the final model. The hyperparameters were tuned such that the learning rate was set to 0.25, and the vector size was fixed at 300. We evaluate the pre-training performance by predicting mask tokens, which involves guessing the missing words or tokens in a sentence. It is worth noting that the model exhibited exceptional training performance while predicting the next masked token during pre-training, with an accuracy score of 91%, precision score of 82.87%, recall score of 82.20%, and F1-score of 82.53%, as illustrated in Table 1.

3.3.2 Code2Vec. Code2Vec capitalizes on the concept of Abstract Syntax Tree (AST) paths to deconstruct a program. In particular, AST paths leverage the program's syntax to comprehend its semantic structure. More information of Code2Vec can be found in given work [1]. In our data split, the training set programs have an average of 211.39 context paths, while the validation set has 195.37 and the test set has 196.69. The Code2Vec approach used 50 epochs and a vector size 300, aligned with Word2Vec. It was noticed that after 30 epochs, there was no significant improvement in pre-training predicting masked tokens in all four performance measures (accuracy, F1, recall, and precision). The model that emerged as the best-performing was generated at iteration 32, employed as the final model. The model recorded a pre-training accuracy score of 90% in detecting masked tokens, a precision score of 80.47%, recall score of 79.70%, and F1-score of 80.08%, as shown in Table 1.



(a) t-SNE of Code2Vec. **(b) t-SNE of Word2Vec.**

(c) k-mean clusters of Code2Vec. **(d) k-mean clusters of Word2Vec.**

Figure 2: t-SNE and k-mean Clustering Visualizations of different Design Patterns after pre-training PLMS.

3.4 Design Patterns Code Input

We utilized the classical GoF DPs, which have been extensively employed for DPR in related literature [19, 20, 22]. Specifically, we employed the Singleton, Prototype, and Builder patterns in our study. For this preliminary study, we selected 10, 10, and 7 publicly available Java programs implementing the Singleton, Prototype, and Builder DPs, respectively, as shown in the second column of Table 2. These programs are of different sizes: Builder examples range from 41 to 105 LOC, Prototype examples from 26 to 67 LOC, and Singleton examples from 18 to 81 LOC. They have 285, 206, and 94 unique words in Builder, Prototype, and Singleton DP programs, respectively. Obtaining automotive source code from repositories that implement GoF DPs is challenging due to the prevalence of copyrighted content. Instead, more generic programs which were identified as containing the DP of interest were sourced from GitHub and Stack Overflow. This labelled dataset was used to calculate precision and recall in Section 4.

3.5 k-means Clustering

We extracted the embeddings of each Java program holding a DP example after pretraining of PLMs and fed them into a k-means clustering algorithm as shown in step 5 of Fig. 1. We employed this technique to predict the DP classes of 27 Java programs. Then we compared the predicted classes to the actual DP classes of given programs to get the performance measures. More explanation is given in the result section 4.

4 RESULTS AND DISCUSSION

We summarize our results for *RQ-1: To what extent can these PLMs recognize design patterns in Java after pre-training?* After the pre-training phase, we used our 27 Java programs, which are labelled with different DPs, as input to these models, then applied k-means clustering to the output embeddings produced by the model. Figures 2a and 2b depict t-SNE visualizations of the extracted embedding vectors of 27 unique Java programs, pre-trained using PLMs. These Java programs contained three distinct DPs, as depicted in Figures 2a and 2b. The presented figures indicate that programs implementing different DPs form easily identifiable clusters, characterized by the green, red, and blue circles.

The embedding vectors generated by the PLMs were subjected to the k-means clustering algorithm to predict the implemented DPs. The results of this clustering process are presented in Figures 2c and 2d, which demonstrate the clusters formed by the Code2Vec and Word2Vec models, respectively. The centroid of each k-means cluster is denoted by a large black dot in the figures. These clustering results are largely consistent with the t-SNE visualizations discussed earlier. To determine the optimal number of clusters for each PLM, we employed the Elbow technique [10]. The results suggest that the ideal number of clusters would be in the range of four to five. Given that our study used three distinct classes of DPs, this may mean that there are some Java programs that are not correctly classified by the PLMs. This may also indicate different sub-types of some DPs, future work should investigate this possibility further. The average DP prediction performance for our 27 input programs is shown in Table 2. Our results demonstrate that Code2Vec outperformed Word2Vec in recognizing Prototype and Singleton DPs, achieving mean F1-scores of 0.842 and 0.833, respectively. In addition, Code2Vec exhibited superior mean performance overall, with the highest precision, recall, and F1-scores of 0.794, 0.774, and 0.781, respectively. However, Word2Vec outperformed Code2Vec in detecting Builder DPs, achieving the highest F1-score of 0.74. Code2Vec can identify characteristic sequences of method calls and variable assignments in DPs like the Builder, whereas Word2Vec treats each code element independently. Code2Vec's ability to capture complex relationships between code elements makes it superior for DPR compared to Word2Vec.

Table 2: Design patterns & files and Performance measure of PLMs after pre-training on detecting DPs.

Design Pattern	*No. of files*	*Precision*		*Recall*		*F1-score*	
		C2V	*W2V*	*C2V*	*W2V*	*C2V*	*W2V*
Builder	7	**0.751**	0.672	0.601	**0.810**	0.667	**0.740**
Prototype	10	**0.803**	0.600	**0.889**	0.600	**0.842**	0.600
Singleton	10	**0.833**	0.801	**0.833**	0.689	**0.833**	0.740
Mean	-	**0.794**	0.690	**0.774**	0.710	**0.781**	0.690

These results suggest that PLMs can be a valuable tool in developing DPR systems for automotive applications.

Overall, our results suggest that PLMs have the potential to be a valuable tool for DPR in the automotive industry. However, further research is needed to fully understand the capabilities and limitations of PLMs in this context.

Summary: **Our suggested approach for DPR using PLMs achieved high precision, recall, and F1-score metrics, with Code2Vec outperforming Word2Vec for most DPs. However, the results may be influenced by several factors and require further validation on a wider range of Java programs.**

5 VALIDITY EVALUATION

In our study, we have adopted the proposed framework by Wohlin et al. [21] for addressing potential threats to validity. We identified two risks for construct validity. The use of Builder, Prototype, and Singleton DPs, relying on a specific programming construct, was mitigated by implementing varying sizes to reduce the influence of a single keyword in extensive programs. The correct implementation of DP is context-dependent. Variations in DP implementations arise due to programming language constraints, project requirements, developer choices, etc.; for instance, with the Singleton DP, variations can exist in instance creation, thread safety, initialization, and other design decisions. These implementation variations can affect our results. We only considered files that implemented a single DP, so our findings cannot be applied to classes that implement multiple DPs.

Considering internal validity, it is important to note that the performance of our approach may depend on several factors, such as the quality and complexity of the Java source code being analyzed, the choice of PLM, and the specific implementation of the clustering or classification algorithms. Considering external validity, our approach was evaluated on a limited number of Java programs and may not represent all possible scenarios. Future research could explore the generalizability of our approach to a wider range of Java programs, which contain a single class with multiple implemented DPs, including domain-specific DP, and other programming languages. If more, overlapping DPs are considered, k-means clustering could become less effective when DPs are less distinctive. If so, further means of distinguishing DPs can be considered, for instance, more advanced clustering methods e.g., Hierarchical Clustering, Density-based clustering) to better differentiate DPs.

The selection of PLMs from numerous existing models was a crucial decision in our study. The study is also limited to Java programs, and we cannot generalize our findings to other programming languages. The second risk relates to the use of the t-SNE and k-means methods for visualizing and predicting classes using embedding vectors, which may not be optimal due to their large size. We evaluated three DPs using two pipelines and pre-training on two large codebases. This increases the generalizability of our results, although further studies are needed.

6 CONCLUSION AND FUTURE WORK

In conclusion, PLMs have shown immense potential in software engineering tasks, and we confirm this potential for DPR specifically as part of this preliminary study. Our pre-training of two established language models on large automotive codebases and subsequent visualization of selected Java programs using t-SNE, and k-means clustering algorithm resulted in high-performance measures in terms of precision, recall, and F1-score. With these results, we find further evidence to show that these language models are capable of storing architecture information of the source code. This discovery has significant implications for the development of robust prediction tools for DPR and other software engineering tasks.

In our future work, we are currently conducting parallel experiments on an industrial codebase and using different PLMs, for instance, CodeBert, and RoBERTa. We plan to collaborate with our industrial partners to validate our findings in the real world and develop an automated tool to identify architectural information in source code of various programming languages.

Acknowledgements. This study was financed by the CHAIR (Chalmers AI Research Center) project "T4AI", Vinnova, Software Center, Volvo Cars, AB Volvo, and the National Science Centre of Poland project "Source-code-representations for machine-learning-based identification of defective code fragments" (OPUS 21), registered under no. 2021/41/B/ST6/02510. The computations/data handling/ were enabled by resources provided by the National Academic Infrastructure for Supercomputing in Sweden (NAISS) and the Swedish National Infrastructure for Computing (SNIC), which is partially funded by the Swedish Research Council through grant agreement no. 2022-06725 and no. 2018-05973.

REFERENCES

[1] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.

[2] Apostolos Ampatzoglou, Georgia Frantzeskou, and Ioannis Stamelos. 2012. A methodology to assess the impact of design patterns on software quality. *Information and Software Technology* 54, 4 (2012), 331–346.

[3] Giuliano Antoniol and Yann-Gaël Guéhéneuc. 2008. Demima: A multilayered approach for design pattern identification. *IEEE Transactions on Software Engineering* 34, 5 (2008), 667–684.

[4] Rhys Compton, Eibe Frank, Panos Patros, and Abigail Koay. 2020. Embedding java classes with code2vec: Improvements from variable obfuscation. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 243–253.

[5] Riccardo Coppola and Maurizio Morisio. 2016. Connected car: technologies, issues, future trends. *ACM Computing Surveys (CSUR)* 49, 3 (2016), 1–36.

[6] Erich Gamma, Ralph Johnson, Richard Helm, Ralph E Johnson, and John Vlissides. 1995. *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH.

[7] Kapilan Kulayan Arumugam Gandhi and Chamundeswari Arumugam. 2017. An approach for secure software update in Infotainment system. In *Proceedings of the 10th Innovations in Software Engineering Conference*. 127–131.

[8] Yann-Gaël Guéhéneuc and Giuliano Antoniol. 2008. Demima: A multilayered approach for design pattern identification. *IEEE transactions on software engineering* 34, 5 (2008), 667–684.

[9] Tae-Hwan Jung. 2021. Commitbert: Commit message generation using pre-trained programming language model. *arXiv preprint arXiv:2105.14242* (2021).

[10] David J Ketchen and Christopher L Shook. 1996. The application of cluster analysis in strategic management research: an analysis and critique. *Strategic management journal* 17, 6 (1996), 441–458.

[11] Chris McCormick. 2016. Word2vec tutorial-the skip-gram model. *Apr-2016.[Online]. Available: http://mccormickml. com/2016/04/19/word2vec-tutorial-the-skip-gram-model* (2016).

[12] Alexander Mirnig, Tim Kaiser, Artur Lupp, Nicole Perterer, Alexander Meschtscherjakov, Thomas Grah, and Manfred Tscheligi. 2016. Automotive user experience design patterns: an approach and pattern examples. *Int. J. Adv. Intell. Syst* 9 (2016), 275–286.

[13] Dana Movshovitz-Attias and William Cohen. 2013. Natural language models for predicting programming comments. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. 35–40.

[14] Anh Tuan Nguyen and Tien N. Nguyen. 2015. Graph-Based Statistical Language Model for Code. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 858–868. https://doi.org/10.1109/ICSE.2015.336

[15] Murali Padmanabha, Daniel Kriesten, and Ulrich Heinkel. [n. d.]. System Design of a Modern Embedded Linux for In-Car Applications. ([n. d.]).

[16] Dhasarathy Parthasarathy, Cecilia Ekelin, Anjali Karri, Jiapeng Sun, and Panagiotis Moraitis. 2022. Measuring design compliance using neural language models: an automotive case study. In *Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering*. 12–21.

[17] Christoph Rieger and Tim A Majchrzak. 2016. Weighted evaluation framework for cross-platform app development approaches. In *Information Systems: Development, Research, Applications, Education: 9th SIGSAND/PLAIS EuroSymposium 2016, Gdansk, Poland, September 29, 2016, Proceedings 9*. Springer, 18–39.

[18] Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. *Advances in Neural Information Processing Systems* 33 (2020), 20601–20611.

[19] Hannes Thaller, Lukas Linsbauer, and Alexander Egyed. 2019. Feature maps: A comprehensible software representation for design pattern detection. In *2019 IEEE 26th international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 207–217.

[20] Nikolaos Tsantalis, Alexander Chatzigeorgiou, George Stephanides, and Spyros T Halkidis. 2006. Design pattern detection using similarity scoring. *IEEE transactions on software engineering* 32, 11 (2006), 896–909.

[21] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering*. Springer Science & Business Media.

[22] Renhao Xiong and Bixin Li. 2019. Accurate design pattern detection based on idiomatic implementation matching in java language context. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 163–174.

[23] Marco Zanoni, Francesca Arcelli Fontana, and Fabio Stella. 2015. On applying machine learning techniques for design pattern detection. *Journal of Systems and Software* 103 (2015), 102–117.

Received 2023-07-07; accepted 2023-07-28