TransDPR: Design Pattern Recognition Using Programming Language Models

Sushant Kumar Pandey[†] Miroslaw Staron [†] Jennifer Horkoff [†] Mirosław Ochodek [+]
Nicholas Mucci[*] Darko Durisic[*]

[†]Chalmers | University of Gothenburg, Sweden
[+]Poznan University of Technology, Poland
[*]Volvo Cars, Gothenburg, Sweden
Email: [sushant.kumar.pandey, miroslaw.staron, jennifer.horkoff]@gu.se[†],
Miroslaw.Ochodek@cs.put.poznan.pl[+], [nicholas.mucci, darko.durisic]@volvocars.com[*]

2023 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM) | 978-1-6654-5223-6/23/$31.00 ©2023 EU | DOI: 10.1109/ESEM56168.2023.10304862

Abstract—**Current Design Pattern Recognition (DPR) methods have limitations, such as the reliance on semantic information, limited recognition of novel or modified pattern versions, and other factors. We present an introductory DPR technique by using a Programming Language Model (PLM) called TransDPR, which utilizes a Facebook pre-trained model (TransCoder), which is a Cross-lingual programming Language Model (XLM) based on a transformer architecture. We leverage an n-dimensional vector representation of programs and apply logistic regression to learn design patterns (DPs). Our approach utilizes the GitHub repository to collect singleton and prototype DP programs written in C++ source code. Our results indicate that TransDPR achieves 90% accuracy and an F$_1$-score of 0.88 on open-source projects. We evaluate the proposed model on two developed modules from Volvo Cars and invite the original developers to validate the prediction results.**

Index Terms—**Design patterns recognition, Programming language models, NLP, Machine learning, Deep learning**

I. Introduction

Modern software is getting more complex, more connected and smarter, for example in the telecommunication domain where critical IoT, factories of the future and autonomous transportation call for lower latency and higher performance. In the automotive domain, autonomous vehicles call for safer and more secure products lead to increased complexity of software [1]. In this context, design patterns (DPs) [2] become more important for practitioners as they allow for high-quality reuse on the conceptual level. DPs refer to a set of classes, objects, and relationships tailored to alleviate a specific design problem within a particular context [3].

The role of DPs in code goes beyond reusability, and is a means of communicating between architects and developers. For example, automotive software architects can prescribe AUTOSAR communication patterns to software developers for implementation. During the quality assurance phase, the architects need to monitor the quality of the implementation of these prescribed patterns. Therefore, recognizing implemented DPs lay the foundation for good quality design and, potentially, for quality improvements [4], [5].

Design Pattern Recognition (DPR) [6]–[9] tools use various techniques to identify patterns, such as code analysis, data flow analysis, and static analysis. A standard DPR approach employs static analysis to identify sets of classes/methods in the system under test, which are linked by relationships such as association and inheritance that are undetectable in DP definitions.

Although methods have been developed to identify DPs, they often work on only a select set of DPs in Java. Existing approaches tend to focus on syntactial patterns, with limited efforts have been devoted to recognizing DPs using semantic (lexical) information [8]. This has a significant limitation as the patterns used in industry are often semantical (e.g., like the communication DPs in AUTOSAR). Furthermore, existing approaches often require compiling code as input, and metrics and static analysis techniques must often have rules expanded pattern by pattern, making it challenging to cover a wide range of different patterns. Our industrial collaborators, Volvo Cars (VC) employ a wide variety of patterns and would like pattern recognition on code fragments in C++. These factors together (inability to capture semantics, limited patterns, code compilation needs, C++), mean that new techniques are needed.

A few successful approaches to extract semantic information are currently available, such as code summarization [10]–[12], code fragmentation [13], and similar code detection [14]. Such techniques could be more versatile against a diverse set of patterns and can work over code fragments. We posit that these factors, as well as utilizing the semantic information of source code can lead to improved DPR models which are practical for our industry partners.

Our research team is currently pursuing a multi-faceted approach to enhancing the development of DPR tools for various industries. Our efforts include extensively exploring a wide range of PLMs, programming languages, DPs, and codebases. In this article, we present one of our studies aimed at exploring the possibilities and limitations of using large language models for DP recognition. This article presents TransDPR, a new DPR model that utilizes semantic information extracted from source code. This model is designed specifically for the C++ programming language, as used by our industrial collaborators, Volvo Cars (VC).

We are using a novel approach to applying PLMs. This approach has comparable (slightly better) performance compared to the state-of-the-art on open-source examples files and has reasonable initial performance on the industrial examples, even given a relatively small labeled training dataset. Thus, we find this avenue of research, PLMs for industrial DPR, promising. Our approach employs TransCoder, a BERT-based PLM model developed by Facebook research, to extract semantic information from the source code. TransCoder is a transformer architecture-based Cross-lingual programming Language Model (XLM) [15]. We then use the extracted semantic information to create a training set, which we use to train a learning model that recognizes implemented DPs in the source code. For now, our emerging results study focuses on two DPs, Singleton and Prototype, allowing us to compare our results to state-of-the-art approaches. In future projects, we evaluate PLMs for DPR with more diverse DP types.

We address the following research questions:

RQ-1: What is the performance of TransDPR on open source examples?

RQ-2: What is the performance of TransDPR on VC modules, according to developer evaluation?

978-1-6654-5223-6/23/ $31.00 ©2023 European Union

RQ-3: How can TransDPR distinguish differences within and between open-source and industrial DP examples?

We address these questions by analyzing both open source code and source code from an automotive company (VC) that contains the studied DPs, and validate the VC results together with modules developers.

We first conducted experiments on the Singleton and Prototype DPs obtained from a public GitHub repository. To evaluate the effectiveness of our proposed model on real projects, we then tested the trained model on two modules developed by VC, where we also got feedback from the original developers of those modules to assess the prediction results. Our experiments demonstrate that our proposed approach achieved an accuracy of 90% and an F_1-score of 0.88 on open-source projects. Furthermore, our results indicate that TransDPR accurately predicted DPs in 10 out of 16 C++ files from the two modules from our industrial partner. We shared our replication package [1] The structure of the manuscript is as follows: In Section II, related work is discussed, followed by the research design in Section III. Section IV presents the results and discussions related to the research questions. All threats to validity are explained in Section V. Finally, Section VI concludes the work and provides directions for future research.

II. Related work

Dwivedi et al. [16] proposed a novel approach for DPR using Layer Recurrent Neural Network and Decision Tree. They evaluated the model on the JHotDraw 7.0.6 open-source project, focusing on detecting the Abstract Factory and Adapter DPs. Tsantalis et al. [5] developed a DP detection method based on similarity scoring between the vertices of a graph. Their approach can recognize DPs that are modified from the standard representation. They tested their method on three open-source projects. These studies showcase innovative methods for detecting DPs in software projects and demonstrate their efficacy in detecting standard and modified DPs. However, it should be noted that the method's reliance on static information and its time-consuming nature due to the splitting of systems into subsystems may limit its applicability in contexts where dynamic information is crucial.

Zanoni et al. [6] proposed a ML-based approach for detecting DPs in source code. Their method utilized graph matching and ML techniques, and they evaluated their approach on ten open-source projects using five DPs. They introduced a tool named MARPLE-DPD and achieved the best performance by utilizing SVM, decision tree, and random forest algorithms. In our work, we have also compared our model with the random forest.

Mayan et al. [3] proposed a DP detection method based on graph theory, consisting of two sequential phases that consider a pattern's semantic and syntactic structure. They evaluated their approach on three open-source projects and achieved 0% to 100% precision and recall. Our investigation is also interested in using semantic information, but does so using a different method – PLMs.

The work of parthasarathy et al. [17] inspired us to explore PLMs for DPR. They conducted an industrial case study using a PLM to detect DP. They pre-trained a model on an industrial codebase and tested it on two industrial controller-handler DPs, with reasonable results. In this study, we focus on different and more classical DPs, allowing us to compare PLM performance to the state-of-the-art.

Thaller et al. developed a method named Feature-Role Normalization for DPR [18]. They introduced flexible and easily understandable human-machine software based on microstructures and feature maps. Using a wide range of training data, the researchers evaluated their methodology on four DPs and compared it with traditional ML methods, outperforming random forest and CNN methods.

Xiong et al. [19] proposed a practical method for detecting DPs from source code, specifically focusing on idiomatic implementation for the Java language. They employed static analysis and inference techniques, and analyzed the integration of DPs' structural, behavioral, and semantic aspects. They found that their proposed method achieved a high detection rate with 85.7% precision and 93.8% recall. However, their work is limited to DPs in Java.

III. Research design

A. TransDPR

TransDPR includes a pre-trained PLM transcoder. We extracted the embeddings from the encoder block, created a training set, and used logistic regression to predict the design pattern, as illustrated in Figure 1. We will provide a detailed explanation of each step in the subsequent sections. TransCoder is a powerful PLM developed by Facebook AI research. With over 10 million parameters, TransCoder is a benchmark model for language models, and we are investigating its potential for DPR. In addition to TransCoder, we also explored other transformer-based PLM models, such as ACoRA [2] and OpenAI CodeX [20], in our parallel studies. However, we found that these models were similar in their ability to extract semantic information from programs and recognize DPs. Since DPs are often language-agnostic and can be applied to multiple programming languages, therefore, a DPR tool that can analyze code written in multiple languages can be more effective than one limited to a specific language.

TransCoder is a cutting-edge cross-lingual language model (XLM) that enables the translation of functions from one programming language to another solely based on monolingual source code. This was achieved through a process of pretraining, where the model was initialized with cross-lingual word representations, as demonstrated in previous studies [21], [22]. Specifically, in the context of programming language translation, the word embeddings for `Char`, `Character` and `Bool` in C++, Java, and Python, respectively, were trained to be close to each other in an n-dimensional space through unsupervised alignment of monolingual word embeddings [23], [24]. The model was trained using sequence-to-sequence and attention mechanisms and followed the pretraining objective of [25]. The publicly available Google BigQuery was used to train the model, and the standard clang library was used for the C++ tokenizer. Additionally, fastBPE was used to concatenate the tokenized source files.

In our study, we utilized the TransCoder model to extract context, i.e., semantic information for a given program. Specifically, we extracted the embedding vector from the encoder block of the model and then labeled these embeddings according to the DP present in the original code. Next, we created a training set using the DP labeled embeddings, which were then fed into a logistic regression layer to learn the semantic pattern of the DPs.

We employed two distinct assessment methods to evaluate the proposed model's performance. Firstly, we conducted a train-test split on the available data (labelled open source program) to examine the prediction accuracy of the TransDPR pipeline. This allowed us to evaluate the pipeline's ability to recognize DPs in programming code accurately. Secondly, we conducted a real-world evaluation of the model

[1] https://github.com/sushantkumar007007/TransDPR-Design-Pattern-Recognition-Using-Programming-Language-Models.git

[2] ACoRA https://github.com/mochodek/acora

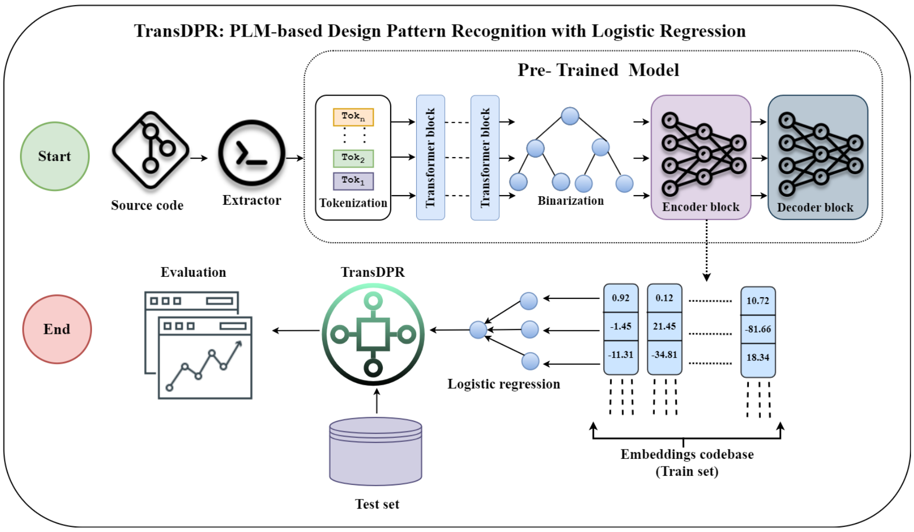

Fig. 1: TransDPR: The top row of the figure illustrates the pipeline steps involved in the Pre-trained Model, and the bottom row shows the training and evaluation of the proposed model.

TABLE I: Program files and design patterns information.

Design Patterns	***No. of examples***	***Language***	**Sources**
Singleton	18	C++	GitHub, & Stack Overflow
Prototype	8	C++	GitHub, & Stack Overflow
Unknown	8	C++	Volvo Cars (CSP)
Unknown	8	C++	Volvo Cars (ED)

by assessing its performance on developed modules within an original equipment manufacturer (OEM, VC) setting. We solicited feedback from the original developers of these modules to gain insight into the model's effectiveness in practical scenarios. By employing these two distinct assessment methods, we could comprehensively evaluate the performance of the TransDPR model both in a controlled setting and in real-world usage scenarios.

B. Open Source Projects and Volvo Cars Modules

In this study, we conducted a data collection process to gather open-source C++ programs that implement the singleton and prototype DPs from various public repositories, including GitHub. Two authors of this study are involved in collecting programs. First, we searched for singleton and prototype DPs programs on Google. Many search results were duplicate, and we feel that we reached a level of near saturation in Google-indexed C++ pattern examples. These programs were explicitly labelled with the use of such patterns; thus, manual labelling of the programs was not necessary. However, we did check that the programs were labelled correctly, i.e., that they used the patterns they claimed. The loc of these programs lies between 9 and 782.

Finally, we collected 18 and 8 C++ implementations of singleton and prototype DPs, respectively, as reported in Table I. We shared all the programs in our replication package. Note that although repositories of labelled DPs exist, to our knowledge, they are in Java only (e.g., PMart), thus the need to create our own collection.

We also collected two C++ VC modules, which are part of an automotive core computing system. We asked our industrial partner to suggest developed modules/files with implemented DPs. These modules, namely Electric Drive system (ED) and Core System Platform (CSP), are integral parts of a service-oriented architecture that interfaces with discrete ECUs in a car running in a POSIX environment. We collected eight source code files for each of these modules, as referenced in Table I. The ED programs communicate with the ECUs that control the traction motors in an electric car, which is facilitated by the CSP modules. It is worth noting that due to a non-disclosure agreement and privacy concerns, we are unable to reveal the complete details of these modules and programs.

C. Data augmentation

In order for TransDPR to work correctly, we need both examples of DPs (singletons and prototypes) and counter examples (non-singletons and non-prototypes). The best results are achieved when the examples and counter-examples are perfect pairs – i.e., for each example the counter-example is a program that is embedded on the opposite side of the latent space. This guarantees that the classifiers (linear regression) and thus TransDPR are not biased towards either a DP or its counter-part. In order to achieve that, we created the perfect counter-examples superficially by multiplying the embedding vectors by -1. Initially, we had 26 embedding vectors, one for every collected open-source program, and after applying data augmentation to each vector, we doubled the number of embedding vectors to 52 in our training set. For example, given the embedding vector $\vec{P_1}$ for a singleton program, we applied the data augmentation method by multiplying it with (-1), yielding the augmented embedding vector $\widehat{\vec{P_1}}$ as shown in Equation 2.

This process created an embedding vector for an imaginary program, which guarantees non-biased classification, but does not tell us which program is the counter-example. However, if provided with a large code base (like the entire Github), we could find the real, nearest programs to these imaginary programs (which can be done in our further work). The alternative approach, creating counter-examples by writing non-singleton and non-prototype programs, can lead to bias as these pairs may not divide the latent space into equal parts (as we do not control the way in which program embeddings are calculated).

$$\vec{P_1} = 1 + [-3.34233 \quad .. \quad -1.87637 \quad -0.34562] \tag{1}$$

$$\widehat{\vec{P_1}} = (-1) * [-3.34233 \quad .. \quad -1.87637 \quad -0.34562] \tag{2}$$

D. Experimental setup and hyperparameter tuning

To evaluate the performance of our proposed model, we employed two methods. First, we divided the data into a training set and a testing set in a 70%-30% ratio

Algorithm 1: TransDPR

```
1  Inputs ← PLM, [$P_1$; $DP_1$], [$P_2$; $DP_2$], [$P_3$; $DP_3$],....,[$P_n$; $DP_n$], $P_k$,
     $P_{k+1}$   /* Label and few unlabel DPs C++ programs */
2  Output → Trained model & $DP_k$, $DP_{k+1}$   /* trained model &
     predicted design patterns */
3  ------------------------------------------------------------
4  Encoding Extraction:
5  for each P, i = 1 to n do
6      $E_i$ = encoding($P_i$)   /* embedding vector from decoder
         block of TransCoder */
7  end
8  ------------------------------------------------------------
9  Data Augmentation:
10 for each P, i = 1 to n do
11     $P_i^{new}$ = (-1)$P_i$   /* Eq. 1 & Eq. 2, here (-1)$P_i$ is
         imaginary program */
12     $DP_i^{new}$ = (not)$DP_i$   /* (not)$DP_i$ is the DP of the
         imaginary program */
13     $NDP_i^{new}$ ← $DP_i^{new}$
14 end
15 ------------------------------------------------------------
16 Training:
17 Data ← [$E_1$; $DP_1$], [$E_2$; $DP_2$]...., [$E_n$; $DP_n$], [$E_1^{new}$; $NDP_1^{new}$],
     [$E_2^{new}$; $NDP_2^{new}$]....,[$E_n^{new}$; $NDP_n^{new}$]  /* data creation from
     embeddings of programs */
18 Data→ $Data_{train}$, $Data_{test}$   /* train test split */
19 LR($Data_{train}$)
20 for every instance i of $Data_{train}$ do
21     $\kappa_i$ = $P_{score}(E_i)$   /* probability score of each
         design pattern */
22     Optimize($\hat{y}$, y)$_{gradientdescent}$   /* update weight */
23 end
24 ------------------------------------------------------------
25 Evaluation & Prediction:
26 Compare($\kappa_i$, $\kappa_i$')   /* compared over $Data_{test}$ */
27 Calculate accuracy(y, $\hat{y}$), $F_1$-score(y, $\hat{y}$)
28 Predict[$P_k$, $P_{k+1}$]   /* Predict DP of unlabeled Prog. */
```

and assessed the prediction performance of the TransDPR model on the 52 open source examples, including the 26 inverted examples. We repeated each experiment 10 times to eliminate any potential random bias and calculated the mean performance measures. We did not have enough instances to predict each DP class. So, we labelled one DP as a positive class and considered the remaining instances as negative classes to obtain predictions for each class. Additionally, we experimented with different ratios, and the results demonstrated that the optimal performance was achieved with a 70%-30% split. We conducted these experiments on a Windows system integrated Intel's integrated graphics with 16 GB RAM.

Second, we obtained results from our TransDPR pipeline for the VC modules, the ED system and the CSP. In order to evaluate the validity and accuracy of these findings, we sought the input of the original developers of these modules to label the files as using Singleton, Prototype, neither, or both. These developers did not see the results of the pipeline when making their determination.

In this study, we employed F_1-score [26] and accuracy [27] to evaluate the performance of the TransDPR. We utilized the Torch framework and Sklearn. We tried to tune hyperparameters, but can't guarantee optimal results despite optimization efforts.

E. TransDPR Algorithm: Encoding, Data Augmentation, Training, Evaluation & Prediction

We provide the pseudocode for the TransDPR pipeline, including the steps for encoding extraction (line 4 to 7), data augmentation (line 9 to 14), training (line 16 to 23), and evaluation & prediction (line 25 to 28), in Algorithm 1. The input to the algorithm consists of the TransCoder PLM, and *n* source code programs (P_1 to P_n) with their corresponding DP labels (DP_1 to DP_n). These, along with the augmented (-1) programs, are used to train the classifier. The algorithm also takes as input C++ programs (P_k, P_{k+1}) which may or may not contain DPs, used for evaluation & prediction. For this study, we consider only singleton and prototype DPs. The algorithm's output (prediction) is the predicted DP of the unknown source code (P_k, P_{k+1}).

Encoding extraction. We extract the semantic information of each program using the encoder block of TransCoder, as indicated in line number 4 of Algorithm 1. The function **encoding**(P) denotes the extraction of embeddings from the pre-trained TransCoder model, resulting in an embedding vector of 1024 tokens. **Data augmentation.** We perform a data augmentation step for each extracted embedding of the labelled P_1 to P_n programs. The process of data augmentation (line number 9 to line 14) involves the implementation of the steps detailed in Equations 1 and 2 for all P_1 to P_n programs, as outlined in section III-C. The resulting augmented data, represented by P_1^{new} through P_n^{new}, should be far away from the original programs, and thus should not have the input DPs.

Training. We construct a dataset (Data in line 17) that comprises $2n$ embedding vectors (the original n and the augmented n) and their corresponding labels (Singleton, Prototype, non-Singleton, or non-Prototype. The data was divided into two separate sets: a training set ($Data_{train}$) and a test set ($Data_{test}$), as indicated in line number 18. The logistic regression function (LR($Data_{train}$)) was applied to the training set, generating a probability score for each predicted class, and the weight update (line 22) was optimized using gradient descent [28]. In this context, κ_i refers to the probability score (P_{score}) associated with each predicted DP of a program. The default value for LR to make a class prediction is 0.5 probability score. The weight update of predicted classes ($\hat{y}$) and actual class (y) was optimized using gradient descent via the Optimize($\hat{y}$, y) function.

Evaluation & Prediction. In addition, we computed the overall predicted score for each class of the test set, as depicted in line number 26 to 28. In this context, κ_i represents the combined probability score observed in the training set, while κ_i' represents the corresponding score in the test set. Finally, we assessed the model's performance in terms of accuracy and F_1-score by comparing the predicted DP of the test set with the actual DP, and predicted DP of the unlabelled programs (line 28).

IV. Results and discussion

This section will present results and address the RQs.

A. What is the performance of TransDPR on open source examples?

We first evaluate TransDPR's efficacy in the open source test set. For this set, the performance of the TransDPR in terms of accuracy, precision, recall, and F_1-score is presented in Table II. We evaluated the performance of the singleton and prototype DPs by treating one class as the positive example and the remaining instances as the negative examples. Overall measures are evaluated by treating prototypes and singletons as one set of positive examples, and all other programs as the negative examples. We see results are much stronger for Prototype vs. Singleton. It is important to note that high accuracy can also indicate the presence of overfitting issues. However, the overall performance of the TransDPR to predict different classes is high, i.e., the mean f-score to predict each class is 0.88.

TABLE II: Performance comparison of TransDPR.

Classes	Accuracy	Precision	Recall	F_1-score
Singleton	50%	0.50	0.60	0.55
Prototype	90%	0.90	0.95	0.923
Overall	90%	0.90	0.90	0.88

RQ-1 Summary: TransDPR had a an 88% F$_1$-score on our 52 (26 original and 26 augmented) open source examples. Results are comparable to state-of-the-art metrics or static analysis approaches.

B. What is the performance of TransDPR on VC modules, according to developer evaluation?

The TransDPR pipeline was used to predict DP labels for the 16 industrial VC modules. The results are presented in the second column of Table III. Specifically, the model predicted 14 source files of the VC modules to have singleton DPs, while two modules (edsc, pot) of ED were predicted to have prototype DPs.

TABLE III: Predicted design patterns of different VC modules/files, and corresponding developer's feedback.

Modules	Predicted	Developer's feedback	Comparison
CSP(cdh)	*Singleton*	Singleton	✓
CSP(cmt)	*Singleton*	Singleton	✓
CSP(hvm)	*Singleton*	Singleton	✓
CSP(hvc)	*Singleton*	Both	✓
CSP(hvct)	*Singleton*	Both	✓
CSP(hvcm)	*Singleton*	Singleton	✓
CSP(imc)	*Singleton*	Prototype	×
CSP(cct)	*Singleton*	Singleton	✓
ED(ct)	*Singleton*	Prototype	×
ED(edic)	*Singleton*	Non-singleton, & non-prototype	×
ED(edsc)	*Prototype*	Non-singleton, & non-prototype	×
ED(edr)	*Singleton*	Singleton	✓
ED(pot)	*Prototype*	Prototype	✓
ED(prt)	*Singleton*	Singleton	✓
ED(stt)	*Singleton*	Prototype	×
ED(sbt)	*Singleton*	Prototype	×

To ensure the accuracy of our produced results, we sought the validation of the original developers of these modules, without sharing our prediction results with them. The third column of Table III presents the developers' feedback regarding the DPs. Upon analysing the results, it was observed that our model correctly predicted 7 out of 8 modules for CSP. However, for ED, only 3 out of 8 modules were correctly predicted, resulting in a total of 10 out of 16 modules being accurately predicted. Notably, there were two modules in CSP that were implemented using both singleton and prototype methods. Despite this, our model predicted only singleton for these modules.

RQ-2 Summary: TransDPR accurately predicted the DPs in 10 out of 16 source files, but failed to identify dual-pattern implementation in two files. Domain expert feedback is crucial for accurate DP recognition in industry.

C. How can TransDPR distinguish differences within and between open-source and industrial DP examples?

We conducted an investigation into the properties of the embedding space employed by TransCoder with regard to the separation of DP, artificial non-DP, and VC code fragments. This investigation was carried out using Principal Component Analysis (PCA) and hierarchical clustering techniques. PCA was applied to reduce the dimensionality of the embeddings to three dimensions to facilitate their visualization as points in Figure 2. In this figure, there are different types of programs represented by shapes, open source programs (circle), while squares (ED) and triangles represent (CSP) Volvo cars modules. The figure shows that Singleton programs (blue) and prototype programs (purple)are making separate imperfect groups, indicating that Singletons and Prototypes can be identified to some degree through PCA analysis.

Some open-source singleton programs are located close to Volvo Cars' ED and CSP Singleton programs. This suggests these VC programs might share similarities or characteristics with Singleton DP. Similarly, open-source

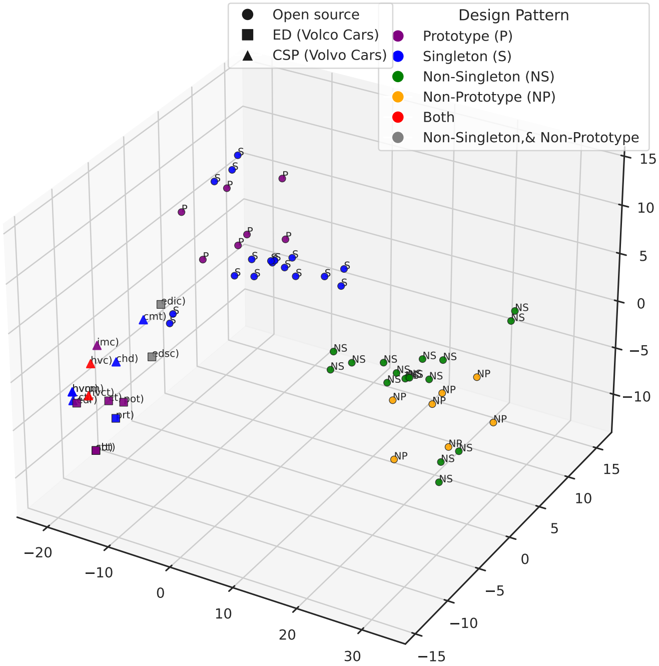

Fig. 2: PCA-based visualization of open source programs (circle) and Volvo Cars modules (square, triangle). The shapes are in a different colours based on implemented DPs.

prototype programs exhibit similar relationships with prototype implemented VC modules. The maximum distance between real programs and their imaginary counterparts indicates a significant difference in their characteristics. The large distance between them shows that they have dissimilar DPs. The PCA figure helps us understand the relationships and similarities between different programs based on their DPs. It highlights the distinctions between open-source programs and VC modules, and the relationships between singleton and prototype programs with the VC modules. These observations can provide valuable insights for further analysis.

The dendrogram in Figure 3 shows the distances between all programs used in our study. From the top, we spot the area DP-NON-DP (in red). The area marks that the distance between these two groups is close to the max cosine distance – 2.00 (on the Y-axis). This indicates that the presence of DPs in the industry programs (group CSP-ED) is closer to the set of programs with DPs than programs without DPs (group NON-DP). Since, a) the NON-DP group was constructed in mind with this separation, and b) our industrial partners confirmed the presence of DPs in their code (CSP-ED), we can initially conclude that the method – TransDPR – can separate the data correctly.

The dendrogram diagram shows interesting results in the group with DPs and the industrial code as shown in the Fig. 3. First, not all DPs are grouped together. The group S-A$_1$ contains a group of ten singletons together, which means that these instances are similar to one another more than to anything else. However, not all singletons are in that group – they are spread over S-A$_2$, S-P$_1$, and S-P$_2$. Since the group S-A$_1$ is quite distant from other groups (please note the cosine distance to other groups that is around 0.75), this means that being singletons is a strong property that TransDPR captures (and quantifies). A similar observation can be made about the pure prototype group – P-A. There, the prototypes are very close to one another and distant to other groups, but the cluster is smaller (both in terms of the number of entities and also in terms of the distance from one another – see the distance within the group at ca. 0.25

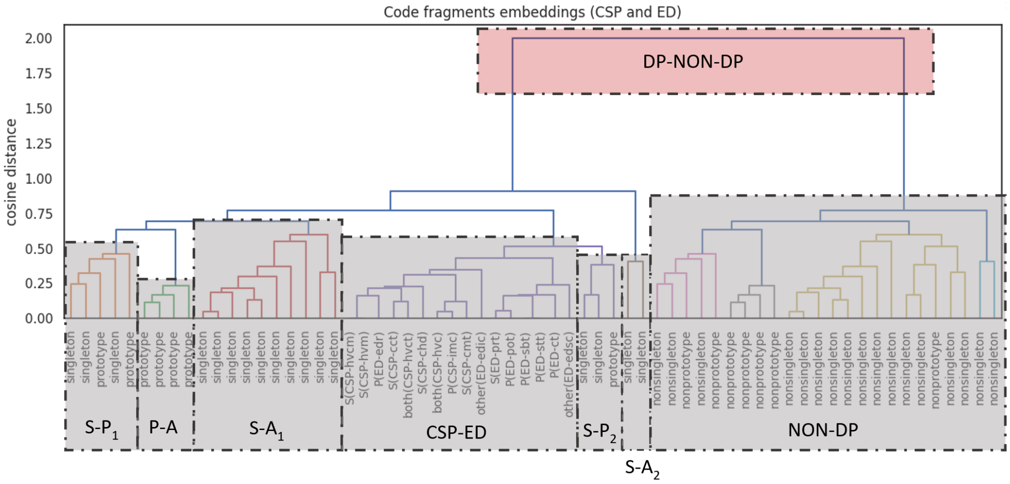

Fig. 3: Dendrogram visualization clustering of example open source and VC modules (cosine distance; linkage method = max. distance). The dendrogram is divided into seven groups of DPs. From left to right, 1) Singleton-Prototype (S-P_1), 2) only Prototype (P-A), 3) only Singleton (S-A_1), 4) VC programs (CSP-ED), 5) Singleton-Prototype (S-P_2), 6) only Singleton (S-A_2), and 7) non-DP implemented (NON-DP). All non-VC programs are open source. *Here:* S(x), P(x), both(x), other(x) indicates file "x" implements singleton, prototype, both singleton and prototype and another DP (neither singleton nor prototype), respectively.

and with the other group at around 0.75).

Two other groups mix singletons and prototypes (S-P_1, and S-P_2), which means that there are other properties of these programs that can be captured by the TransDPR model; other than being singletons or prototypes. That is something that we need to study more in our future work – we need to understand which are these properties. Finally, the important observation is also the fact that CSP-ED group of programs contains all industrial code that is more similar to one another than to other groups, these programs also implemented DPs. With the exception of one ED program, there are two subgroups – one for CSP programs and one for ED programs. Since the company code follows one design convention, one domain and one product, forming one single cluster shows that TransDPR can capture that. However, that also means that there are other properties of the source code that TransDPR captures; properties that have nothing to do with DPs.

RQ-3 Summary: Our investigation revealed that open programs implemented by prototype and singleton DP form small, distinct clusters, while VC modules form a larger cluster with two sub-clusters for each module. Furthermore, the VC modules exhibit similarity to two singleton and one prototype programs. Although DP affects distances, the source of the code (open source vs. specific industrial modules) has a stronger effect.

V. Validity Evaluation

Several concerns were identified in our experimental design. Firstly, the selection of DPs, where our training set was limited to singleton and prototype DPs only. This limits the generalizability of our findings to other DPs. However, we decided to use industry source code, and the availability of experts to vet our results, which weighed stronger than having a large number of non-vetted examples. The fact that we decided to use industrial code dictated other design choices. We complemented this data with the data from open source, but due to the lack of vetted C++ datasets with DPs, our open-source dataset is also limited at the moment (part of current work).

In our work, we use an augmented data set which multiplies embeddings that capture a DP with -1, in order to produce an embedding that is far away from the original. We assume that this inverted embedding does not contain the original DP. However, the resulting embeddings are not real, compilable programs, thus they may not be realistic examples for training. However, our results using this technique are still relatively positive. In addition, the small size of our training set may result in overfitting and sparse data problems, which could affect the accuracy of our results.

Finally, larger programs were divided into smaller subprograms based on functions to extract semantic information as embedding vectors from the pre-trained TransCoder model. The embedding vector for each function was computed, and their mean was taken. – although this is an established technique in NLP, it can introduce information loss.

VI. Conclusion and Future work

Our proposed DPR method, called TransDPR, uses semantic information from open-source programs to overcome limitations of current work by extracting semantic information of a program in form of embeddings, using a pipeline of a Pre-Trained model and a classifier. We created a small training set using extracted embeddings from open-source code. Our TransDPR pipeline trained on this data achieved an accuracy of 90% and an F_1-score of 0.88 on open-source code. We also validated our model in real-life applications by evaluating it over two Volvo Cars modules, comparing the predicted results with the original developers of those modules. Overall, our proposed method has shown promising results and could improve the efficiency of software reengineering processes.

We are currently engaged in a parallel investigation of alternative PLMs and DPs within Java. Our research endeavours also include pre-training existing PLMs via alternative codebases, enabling the models to assimilate coding practices that can aid in developing DPR tools for automated software in industries.

Acknowledgements. This study was financed by CHAIR (Chalmers AI Research Center) project "T4AI", Vinnova, Software Center, Volvo Cars, AB Volvo, Chalmers, University of Gothenburg, and National Science Centre, Poland project "Source-code-representations for machine-learning-based identification of defective code fragments" (OPUS 21), registered under no. 2021/41/B/ST6/02510, also provided funding.

References

[1] V. Antinyan, A. B. Sandberg, and M. Staron, "A pragmatic view on code complexity management," *Computer*, vol. 52, no. 2, pp. 14–22, 2019.

[2] E. Gamma, R. Helm, R. Johnson, R. E. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH, 1995.

[3] B. B. Mayvan and A. Rasoolzadegan, "Design pattern detection based on the graph theory," *Knowledge-Based Systems*, vol. 120, pp. 211–225, 2017.

[4] M. Vokác, "An efficient tool for recovering design patterns from c++ code." *J. Object Technol.*, vol. 5, no. 1, pp. 139–157, 2006.

[5] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis, "Design pattern detection using similarity scoring," *IEEE transactions on software engineering*, vol. 32, no. 11, pp. 896–909, 2006.

[6] M. Zanoni, F. A. Fontana, and F. Stella, "On applying machine learning techniques for design pattern detection," *Journal of Systems and Software*, vol. 103, pp. 102–117, 2015.

[7] N. Pettersson, W. Löwe, and J. Nivre, "Evaluation of accuracy in design pattern occurrence detection," *IEEE Transactions on Software Engineering*, vol. 36, no. 4, pp. 575–590, 2010.

[8] N. Nazar, A. Aleti, and Y. Zheng, "Feature-based software design pattern detection," *Journal of Systems and Software*, vol. 185, p. 111179, 2022.

[9] H. Yarahmadi and S. M. H. Hasheminejad, "Design pattern detection approaches: a systematic review of the literature," *Artificial Intelligence Review*, vol. 53, no. 8, pp. 5789–5846, 2020.

[10] P. W. McBurney and C. McMillan, "Automatic source code summarization of context for java methods," *IEEE Transactions on Software Engineering*, vol. 42, no. 2, pp. 103–119, 2015.

[11] P. W and C. McMillan, "An empirical study of the textual similarity between source code and source code summaries," *Empirical Software Engineering*, vol. 21, no. 1, pp. 17–42, 2016.

[12] J. Zhang, X. Wang, H. Zhang, H. Sun, and X. Liu, "Retrieval-based neural source code summarization," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020, pp. 1385–1397.

[13] N. Nazar, H. Jiang, G. Gao, T. Zhang, X. Li, and Z. Ren, "Source code fragment summarization with small-scale crowdsourcing based features," *Frontiers of Computer Science*, vol. 10, no. 3, pp. 504–517, 2016.

[14] Q. U. Ain, W. H. Butt, M. W. Anwar, F. Azam, and B. Maqbool, "A systematic review on code clone detection," *IEEE access*, vol. 7, pp. 86 121–86 144, 2019.

[15] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2017. [Online]. Available: https://arxiv.org/abs/1706.03762

[16] A. K. Dwivedi, A. Tirkey, R. B. Ray, and S. K. Rath, "Software design pattern recognition using machine learning techniques," in *2016 ieee region 10 conference (tencon)*. IEEE, 2016, pp. 222–227.

[17] D. Parthasarathy, C. Ekelin, A. Karri, J. Sun, and P. Moraitis, "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*, 2022, pp. 12–21.

[18] H. Thaller, L. Linsbauer, and A. Egyed, "Feature maps: A comprehensible software representation for design pattern detection," in *2019 IEEE 26th international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 2019, pp. 207–217.

[19] R. Xiong and B. Li, "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, 2019, pp. 163–174.

[20] W. Zaremba and G. Brockman, "Openai codex," *URL: https://openai.com/blog/openai-codex*, 2021.

[21] G. Lample, A. Conneau, L. Denoyer, and M. Ranzato, "Unsupervised machine translation using monolingual corpora only," *arXiv preprint arXiv:1711.00043*, 2017.

[22] M. Artetxe, G. Labaka, E. Agirre, and K. Cho, "Unsupervised neural machine translation," *arXiv preprint arXiv:1710.11041*, 2017.

[23] A. Conneau, G. Lample, M. Ranzato, L. Denoyer, and H. Jégou, "Word translation without parallel data," *arXiv preprint arXiv:1710.04087*, 2017.

[24] M. Artetxe, G. Labaka, and E. Agirre, "Learning bilingual word embeddings with (almost) no bilingual data," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2017, pp. 451–462.

[25] G. Lample and A. Conneau, "Cross-lingual language model pretraining," *arXiv preprint arXiv:1901.07291*, 2019.

[26] D. M. Powers, "Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation," *arXiv preprint arXiv:2010.16061*, 2020.

[27] J. Huang and C. X. Ling, "Using auc and accuracy in evaluating learning algorithms," *IEEE Transactions on knowledge and Data Engineering*, vol. 17, no. 3, pp. 299–310, 2005.

[28] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv preprint arXiv:1609.04747*, 2016.