# Attention-based Method for Design Pattern Detection

Rania Mzid, Ilyes Rezigui, Tewfik Ziadi

# Attention-based Method for Design Pattern Detection

Rania Mzid[1,2][0000−0002−3086−370X], Ilyes Rezgui[1], and Tewfik Ziadi[0000−0001−9241−8276]3

[1] ISI, University Tunis-El Manar, 2 Rue Abourraihan Al Bayrouni, Ariana, Tunisia
[2] CES Lab ENIS, University of Sfax, Sfax, Tunisia
rania.mzid@isi.utm.tn, ilyes.rezgui@etudiant-isi.utm.tn
[3] Sorbonne Université CNRS, LIP6, F-75005 Paris, France
tewfik.ziadi@lip6.fr

**Abstract.** Design patterns are standard solutions to recurrent software engineering problems. The use of design patterns helps developers improve software quality. However, when integrating design patterns into their systems, software developers usually do not document their use. To this end, the use of an automatic approach for their detection may accelerate program comprehension, assist developers in software refactoring, and reduce efforts during the maintenance task. In this paper, we propose an attention-based approach for design pattern detection. Specifically, we utilize an automatic feature extraction step with a transformer-based model incorporating the attention mechanism. Based on an unsupervised approach, this step learns from source code to identify code attributes and then produces embedding vectors. These vectors capture syntactic and semantic information related to design pattern implementations and serve as input to train a classifier for the design pattern detection task. The attention mechanism is used to produce important representative features of design pattern implementations and improve the accuracy of the classification model. The evaluation shows that our classifier detects GoF design patterns with an accuracy score of 86%, precision of 87%, recall of 86%, and F1-score of 86%. The comparison of our findings with state-of-the-art methods shows an improvement in (i) precision of 25%, (ii) recall of 6%, and (iii) F1-score of 8%.

**Keywords:** Design pattern detection · Feature extraction · classification · Transformer architecture

## 1 Introduction

Design patterns are typically defined as a solution to a recurring problem applicable in a specific context [18]. In software engineering, the application of design patterns significantly simplifies the work of software developers and enhances the quality of software systems across various domains. Patterns may aid in the reuse of existing knowledge regarding similar development difficulties. They propose previously used and optimized solutions to specific problems in a variety

of contexts. The lack of pattern-related data after the application of a design pattern in source code is a common problem. Indeed, software developers apply design patterns without frequently documenting their use. However, knowing their presence in a software system improves the developer's comprehension and provides essential information for the developers to facilitate software refactoring and maintenance tasks. One way to reduce refactoring and maintenance costs is to detect design patterns. However, manually identifying and locating these patterns can be complex and time-consuming. Developers must not only recognize where a pattern is applicable but also ensure that it is applied correctly to reap its full benefits.

Different approaches have been proposed in the literature for design pattern detection. Some of those approaches rely on intermediate representations [5] [17] and aim to convert the source code of the system implementing the design pattern into an intermediate representation, like a graph or a matrix followed by a comparison to find common structural elements. These methods show low performance, and frequently fail to distinguish between design patterns with similar structures. Metric-based approaches [6] [9] for design pattern detection uses quantitative measures to identify and assess the presence of design patterns in source code. However, the possibility of implementing a design pattern in various ways makes the use of software metrics challenging to capture all instances accurately. Indeed, software metrics are generally based on structural analysis, aiming to quantify certain structural characteristics and relationships between the code elements. To address the limitations of the previous two approaches, feature-based methods for design pattern detection have recently emerged. Detecting design patterns in code refers to the process of checking whether a source code implements the features that define that pattern. Code features correspond the specific characteristics of code that define a particular pattern. These methods are usually supported by machine learning techniques and mainly consist of two stages: examining the code statically to extract syntactic and semantic code features (known as feature extraction), then constructing a machine learning model for classification. The feature extraction step is challenging since the extracted attributes have a strong impact on the classification model's performance. In addition, this step is judged to be hard and time-consuming [15] [11] due to the various representations of design patterns (i.e., variants). Another challenge facing feature-based approaches is embedding source code into representative vectors in the dimensional space and that is due to the complex structure of source code itself. Embedding helps capture semantic representation, which consequently contributes to enhancing the accuracy of the classification model. To perform this step, existing approaches perform a transformation of source code to a textual representation that represents code features [11] [16], then employ word embedding techniques such as word2vec [3] to produce the embedding vectors.

Motivated by the limitations of existing works, we propose in this paper a fully automated approach for GoF design pattern [8] detection named $\text{DPD}_{Att}$. The proposed approach, based on the transformer architecture and enriched with

the attention mechanism [21], intends to automatically extract code features using an unsupervised approach. Transformer is a neural network architecture first introduced by Ashish et al. [21] in 2017. It adheres to an encoder-decoder structure, with the encoder tasked with transforming the input into an intermediate representation that the decoder can subsequently employ to generate output. Existing architectures built on top of this transformer often adopt one or more of its building blocks, as some rely on an encoder-only network, whereas others employ a decoder-only or encoder-decoder architecture. In this paper, the feature extraction module is built on the transformer's encoder. The attention mechanism within a transformer-based auto-encoder enables the capture of contextual information within an input sequence. It produces an output vector that encodes the relationships between words or sub-words known as tokens in the input. This method ensures that tokens sharing semantic similarities are represented by vectors with smaller distances in the vector space, making it useful for learning representations of sequential data such as code. Furthermore, to eliminate treating a source code as plain text for embedding, we utilize a code embedding model in this paper. The resulting contextualized embeddings retrieved by our feature extraction module serve as rich representations that include syntactic and semantic information, making them helpful for design pattern detection. The main contributions of this paper can be summarized as follows:

- We provide an attention-based approach for design pattern detection called DPD$_{Att}$. The proposed approach uses the transformer's encoder to automatically learn features with syntactics and semantics from encoded token vectors extracted from source code, and then uses the attention mechanism to generate key features (i.e., embedding vectors) for training a more precise design pattern detection model. The approach is fully automated, and a tool called the DPD$_{Att}$ detector is created.
- We build a large corpus DPD$_{Att}$ corpus covering 13 GoF design patterns and consisting of 1645 labeled Java files.
- We demonstrate the effectiveness of our approach by comparing it with state-of-the-art methods in terms of precision, recall, and F1-score.

The rest of this paper is organized as follows: In Section 2, we provide background information on the concept of features in source code. Section 3 describes the proposed approach. In Section 4, we present implementation details to create our DPD$_{Att}$ detector, we discuss evaluation results, and we compare the proposed approach with the state-of-the-art methods. Section 5 overviews related work. Conclusions and future work are discussed in Section 6.

## 2 Code features

Code features are code attributes or characteristics that are extracted or computed from source code for a specific purpose. The code features used are determined by the task at hand. Code features might be syntactic or semantic. Syntactic features are the structural elements and arrangement of the code, whereas

semantic features are the meaning and behavior enclosed within those structures. In object-oriented programming, syntactic features may include class and method declarations, variable names, inheritance relationships, and the overall organization of code blocks. The inclusion of access modifiers like "public" or "private," or the use of particular phrases like "extends" for inheritance are some examples of syntactic features. On the other side, semantic characteristics examine the functionality and logic contained in the code. Important semantic features include, for instance, the creation of objects, method calls, and interactions between various classes and objects. For feature extraction, various strategies may be used. Static source code analysis, which involves examining the code without executing it, may be used to extract static features. This includes techniques such as parsing the code to construct an abstract syntax tree (AST) or using data flow analysis to understand how data moves through the program. Furthermore, advanced static analysis techniques such as symbolic execution [19], formal methods [27], or machine learning [16] can be used to derive more in-depth semantic features, revealing insights into the code's intended behavior. In this paper, we are interested in machine learning techniques to extract syntactic and semantic features from source code. In particular, we aim to automatically extract code features using an unsupervised approach.

## 3 Proposed approach

The proposed approach as depicted in Fig. 1 includes three main phases: the data collection and preparation, the feature extraction, and the classification. Each of these phases is explained in detail in the following subsections.

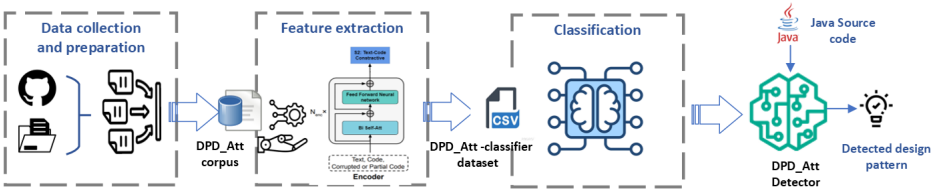


Fig. 1: Proposed approach in a nutshell

### 3.1 Data collection and preparation

The effective training of any machine learning model requires a substantial volume of high-quality data. In our specific research context and to identify GoF design patterns within Java source code, we opted for class-level detection as it is the lowest possible level of granularity that may capture design patterns. We started by conducting an extensive search for an open-source dataset. Our search

led us to Nazar et al.'s [16] corpus, $DPD_F$, one of the few open-source available design pattern detection datasets. This dataset was considered a starting point for our work and a benchmark corpus. The $DPD_F$ comprises 1300 files sourced from 216 projects, covering 12 design patterns. These files were carefully selected from the GitHub Java Corpus [1] and expertly labeled through crowd knowledge. We further processed this dataset, starting with eliminating redundant files and keeping only the ones with valid GitHub URLs and a ".java" file extension, resulting in a reduced dataset size of 943 labeled examples. This data curation step is essential since having duplicates in the dataset can lead to inflating performance metrics. Following this, we continued to work with the refined $DPD_F$, expanding it to encompass 1645 Java files and to include 13 design patterns, thereby covering a wider spectrum of design patterns and implementations. These additional files were also collected from GitHub and labeled manually by experts in the field. Subsequently, we named this refined corpus $\text{DPD}_{Att}$, which played a fundamental role in our feature extraction and the training of our multi-class classification models. As depicted in Fig. 2, the $DPD_F$ dataset encompasses 12 design patterns in addition to the "unknown" label. Whereas, our dataset extends its label set to include the "Strategy" design pattern [8], bringing the total to 14, including the "unknown" label.

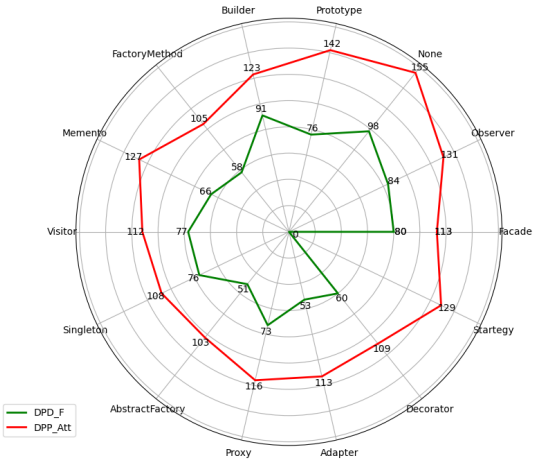


Fig. 2: Comparison between the $DPD_F$ and $\text{DPD}_{Att}$ corpora by the number of collected instances per design pattern

### 3.2 Feature Extraction

In this paper, we aim to utilize a machine learning classifier for the detection task. To do so, the classifier must be trained to distinguish between the features that represent each design pattern. As a result, we need to supply it with

data that implements these GoF pattern features so that it learns to map source code to the design pattern it implements. However, supervised learning classifiers cannot process code in its raw format, making an intermediary step of feature extraction required. The feature extraction step as depicted in Fig. 1, aims to automatically extract code features from the collected Java files ($DPD_{Att}$ corpus) and represent them as fixed-size vectors ($DPP_{Att}$-classifier dataset). The feature extraction starts with the tokenization step. This tokenization refers to the process of mapping a given input into a vector of integers based on a specific vocabulary, where each integer corresponds to a word or sub-word from the input code. An example of a Java class called "Database" to which the Singleton design pattern is applied is shown in Fig3. This code enables clients to create a connection pool for accessing the database, ensuring the reuse of the same connection across all clients. A possible tokenization process result is shown in Fig. 4. In this figure, the token "1" represents a special character initiating the start of the sentence, and the token "2" represents a special token for the end of the input. Then, the feature extraction module automatically extracts code features from the tokenized $DPD_{Att}$ corpus given as input. The attention mechanism is used to capture contextual information and perform semantic analysis. Our feature extraction module utilizes this mechanism to learn how much a token from the input sentence relates to all the other tokens from the same sentence. That way, tokens that contribute most to the context of others will be looked at with more "attention" during the learning. This contextualization is very important in the context of pattern detection since it allows distinguishing between code elements that are relevant to the implementation of the design pattern and other tokens from the source code that are irrelevant. As a final step, these learned dependencies are then mapped to an output vector, which is our embeddings. It is worth noting that the size of this vector, called embedding size, has an impact on the detection task, which will be discussed in more detail in the evaluation section. Fig. 5 depicts the embedding vector for the Java class "DataBase" with an embedding size equal to 256. Each element in the embedding vector represents a specific feature characterizing the design pattern Singleton in the source code.

```
class Database {
   private static Database dbObject;
   private Database() {
   }
   public static Database getInstance() {
      // create object if it's not already created
      if(dbObject == null) {
         dbObject = new Database();
      }
       // returns the singleton object
       return dbObject;
   }
   public void getConnection() {
       System.out.println("You are now connected to the database.");
   }
}
```

Fig. 3: An example of a Java class "DataBase" to which the Singleton pattern is applied

```
tensor([1,    203,  1106, 5130,  288,  203,  282,  3238, 760,  5130, 1319, 921,  31,   203,  282,  3238,
    5130, 1435, 288,  4202, 203,  282,  289,  203,  282,  1071, 760,  5130, 3694, 1435,  288,  203, 1377,
    368,  752,  733,  309,  518,  1807, 486,  1818, 2522,  203,  1377, 309,  12,   1966, 921,  422,  446,
    13,   288,  203,  540,  1319, 921,  273,  394,  5130, 5621, 203,  1377, 289,  203,  4202, 368,  1135,
    326,  6396, 733,  203,  4202, 327,  1319, 921,  31,   203,  282,  289,  203,  282,  1071, 918,  6742,
    1435, 288,  203,  4202, 2332, 18,   659,  18,   8222, 2932, 6225, 854, 2037, 5840, 358,  326,  2063,
    1199, 1769, 203,  282,  289,  203,  97,   203,  2])
```

Fig. 4: Tokens of the Java class "DataBase"

```
tensor([-0.0599, -0.0019, -0.1362, -0.0059, -0.1154,  0.0384,  0.0832, -0.0376, -0.0993,  0.0214,  0.0576,  0.0866,  0.0443,  0.0262,  0.0683,
         0.0198,  0.0836, -0.1733, -0.0918,  0.0761,  0.0331,  0.0234, -0.0019,  0.0564,  0.0160,  0.0435, -0.0038,  0.0191, -0.0854, -0.0252,
        -0.0348, -0.0619, -0.0560, -0.0234, -0.0905,  0.0341,  0.0479, -0.0481,  0.0293,  0.0505, -0.0272,  0.0239,  0.1233,  0.0406,  0.0708,
         0.0073, -0.0586,  0.0429,  0.0112, -0.1061,  0.0861,  0.0571,  0.0533, -0.0291, -0.0571,  0.0004,  0.0285,  0.0192,  0.0083,  0.0727,
        -0.0278, -0.0619, -0.0561, -0.0440, -0.0364,  0.0140,  0.0142, -0.0319, -0.0240, -0.0987, -0.0095,  0.0792,  0.0146, -0.0824, -0.1464,
         0.0493, -0.0365, -0.1038,  0.0748, -0.1273,  0.0096,  0.0127,  0.0376, -0.0701, -0.0534, -0.0139, -0.0733, -0.0380,  0.1140, -0.0623,
         0.0147,  0.0357,  0.0839,  0.0499, -0.0235,  0.0419,  0.0270, -0.0733,  0.0156, -0.0276, -0.0074, -0.0528, -0.0212, -0.0434,  0.0233,
         0.0309, -0.0446,  0.0289, -0.0390, -0.0726, -0.0382,  0.0447,  0.0352,  0.0756, -0.0075, -0.0513, -0.1216, -0.0088,  0.0157, -0.0249,
         0.0972,  0.0932,  0.0408,  0.1057,  0.0070, -0.0331, -0.0617,  0.0060,  0.0713, -0.0198,  0.0048, -0.0483, -0.0229,  0.0209, -0.0004,
        -0.0154, -0.0235, -0.1061,  0.1601,  0.0173, -0.0198,  0.0494,  0.0565,  0.0537, -0.0054, -0.0800,  0.0901, -0.0204, -0.0797, -0.1150,
         0.0884, -0.0690, -0.1142, -0.0339,  0.0111, -0.0533, -0.0654,  0.0312,  0.0156,  0.1225,  0.0139,  0.0977,  0.0066, -0.0013, -0.0131,
         0.0762, -0.0556,  0.0423,  0.0634,  0.1082, -0.0373,  0.0078, -0.0478,  0.0431,  0.0101, -0.1972, -0.0150, -0.0603,  0.0469, -0.0080,
         0.0867,  0.0667, -0.0568, -0.1139,  0.0926,  0.0226, -0.0058,  0.0367, -0.1321, -0.0385,  0.0581,  0.0676,  0.0019, -0.0905, -0.0394,
        -0.0797, -0.0053,  0.0525,  0.1161,  0.0251, -0.0484, -0.0360,  0.0391,  0.0485, -0.0285, -0.0750,  0.0178,  0.0649,  0.0151, -0.0130,
        -0.0383, 0.1017,  0.0646,  0.0963, -0.0029,  0.0035,  0.0368,  0.0024,  0.0705, -0.0175, -0.1066, -0.0705,  0.0345, -0.0398,  0.0171,
         0.0783, 0.0085,  0.0318,  0.0706,  0.0262,  0.1060,  0.0507, -0.0347, -0.0416,  0.1388, -0.1385, -0.0072, -0.0366,  0.0398, -0.0862,
         0.0602, 0.0041, -0.0139, -0.0036,  0.0715, -0.1764,  0.0325,  0.1186,  0.0024,  0.0252, -0.0263, -0.0007,  0.0133,  0.0036,  0.0775,
         0.0557], grad_fn=<SelectBackward0>)
```

Fig. 5: Embedding vector for the Java class "DataBase"

### 3.3 Classification

Different paradigms exist when it comes to training machine learning models. Supervised learning is one of them. Supervised learning involves training machine learning models on labeled data. During its training, the model takes inputs in addition to what it is supposed to predict given these inputs. That way, the supervised learning algorithm can learn to map like-wise data points to corresponding target values. Supervised learning can solve two types of problems by providing the correct data. These problems can be regression or classification problems. When the output we want to predict takes continuous values, we are dealing with regression, while when it takes a finite set of discrete values, it is a classification problem. In our work, we aim to detect design patterns given the extracted features from the source code. Since the target we want to predict takes a value from a discrete set of values, being the 13 design patterns our collected dataset $\text{DPD}_{Att}$ covers, we can then conclude that the design pattern detection task is a classification problem. For the classification step, we first constructed the $\text{DPD}_{Att}$-classifier dataset as shown in Fig. 1 based on the the embedding vector produced in the feature extraction step. This dataset will be used to train our classifier. Subsequently, we conducted experiments with various classifiers and assessed their performance. Notably, we trained and evaluated the quality of a Support Vector Machine (SVM), a Multi-Layer Perceptron (MLP), and a Multinomial Logistic Regression (MLR) multi-class classifiers [12]. Once

the choice of the classification model is made, dividing the data for training and testing is a crucial step. Testing data should be around 20% to 30% of the total size of the data. The training data is the one passed to the classification model with its corresponding label so that it learns the design pattern features from it. The testing data is then used to evaluate how well the model has learned from its training process. Each Java source code is passed without its corresponding label, as the trained classifier is required to predict that. Then the evaluation is done based on the number of correctly predicted patterns and the wrongly detected ones.

**Cross-validation:** One way to assure the quality of the classification and assess the model's ability to generalize is to use the cross-validation technique [13]. As we said, a sample of the data is used for testing. This sample could be situated in any part of the corpus. For example, if the testing data is 25% of the total size of the dataset, these 25% could be situated in the first Fold of the data, the second, the third, or the fourth. Cross-validation takes that into consideration by evaluating each of these scenarios individually and then taking the mean value as the final evaluation score.

## 4 Experimental results

This section first presents the implementation details. Then, an evaluation and a comparison with related work are given. At the end of this section, we discuss the computational complexity of the proposed approach.

### 4.1 Implementation

For the feature extraction step, our design pattern detector ($\text{DPD}_{Att-Detector}$) uses the CodeT5+'s [23] encoder, an open-source transformer-based code Large Language Model (LLM). To produce embedding vectors, a code embedding model called codet5p-110m-embedding model [23] made available on the Hugging Face Transformers library in May 2023 is used [4]. For the detection task, we used the Scikit-learn library in Python to train the SVM, MLP, and MLR classifiers.

### 4.2 Evaluation

In this section, we first introduce the considered evaluation criteria in this paper. Then, we describe the evaluation results and compare our findings with related work. The computational complexity is given at the end of this section.

**Evaluation criteria.** For the statistical evaluation of our approach, we report four different metrics: Precision, Recall, F1-Score, and Accuracy.

[4] https://huggingface.co/Salesforce/codet5p-110m-embedding

- **Accuracy** measures the proportion of correctly classified instances (both true positives and true negatives) out of all instances in the dataset and is calculated as:

$$\textbf{Accuracy} = \frac{\text{Correct Predictions}}{\text{Total Predictions}} \tag{1}$$

  Where Correct Predictions is the sum of true positives and true negatives, the denominator is the total number of instances in the dataset.
- **Precision** inform you on the model's ability to classify positive instances correctly while minimizing false positives. We calculate precision as:

$$\textbf{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}} \tag{2}$$

  The term True Positive represents the instances correctly identified. Whereas the term False Positives indicates the number of instances that were classified as correct but were falsely predicted.
- **Recall** indicates the fraction of instances belonging to a certain class and being identified as such by the classifier. We calculate the recall metric as:

$$\textbf{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} \tag{3}$$

  False Negatives informs about the instances classified as negative when they are actually positive.
- **F1-score** quantifies the harmonic mean between precision and recall as it balances between them. We determine F1 as:

$$\textbf{F1} - \textbf{score} = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{4}$$

**Evaluation results.** By considering the evaluation metrics described in the previous section, we conducted experiments with our proposed approach. We started by analyzing the impact of the feature extraction module on the classification process, especially the embedding size. To explore this, we adjusted the length of the extracted features (i.e., embedding size) and examined its effect on the detection task. Our findings indicated that maintaining a size of 300 produced the most favorable outcomes. Specifically, it led to superior accuracy, precision, recall, and F1 scores compared to other sizes. Fig. 7 depicts the ability of our tool to detect design patterns given different numbers of features (100, 250, 300, and 350) generated by the feature extraction step. As depicted in Fig. 7, increasing the embedding improved results. This enhancement can be attributed to the increased capacity provided to the encoder, allowing it to better capture features from the Java source code and minimize information loss. We can also see that over-increasing the size of the vector to 350 resulted in decreased performance, which can be explained by overfitting. Overfitting occurs when the model becomes too complex for the available data and starts memorizing the data instead of learning from it. In this case, when we increased the size of our learned vector to 350 features, we provided the model with more parameters to learn, which made it more capable of fitting the training data. That can be

explained by its inability to generalize to new, unseen data. As a result, we configured our model to generate vectors of size equal to 300.These vectors were represented as torch tensors, which we then used to build $DPD_{Att-Classifier}$ dataset. The $DPD_{Att-Classifier}$ dataset is the csv file that is going to be passed to the classifier for training and testing and is composed of 303 columns. The first column serves as the name of the Java project. The second is the name of the class, while the third represents the design pattern detected (label). The last 300 columns are the features extracted by CodeT5+'s encoder, capturing the necessary information about each Java source code from each project.

We also compared the different classifiers for detecting GoF design patterns in terms of accuracy, precision, recall, and F1-score. The different classifiers utilized 80% of the $DPD_{Att-Classifier}$ dataset for training and 20% for testing. To ensure that the models do not overfit the training data. We opted for the K-fold cross-validation technique with K = 5 by utilizing the K-fold module in the sklearn library. Fig. 6 shows the mean values for the defined metrics. As depicted in Fig. 6, the SVM classifier results in slightly better performance than the other classifiers. This is the reason why we opted for the SVM classifier for our detection tool ($DPD_{Att-Detector}$). Indeed, using our $DPD_{Att-Classifier}$ dataset and the SVM classifier, we obtained an accuracy score of 0.86, an average precision score of 0.87, an average recall score of 0.86, and a harmonic mean of 0.86. Table

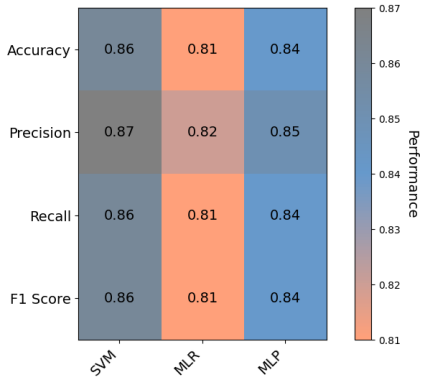


Fig. 6: Performance comparison between the used classifiers

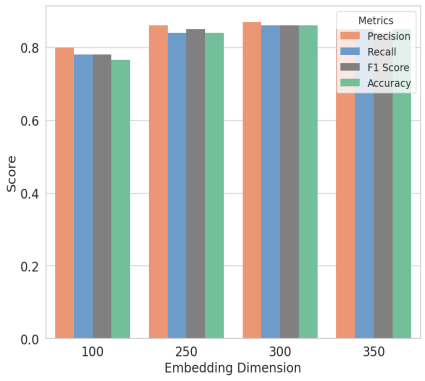


Fig. 7: Average experimental results with respect to the embedding size

1 gives the evaluation of our detection tool per label using the SVM classifier and $DPD_{Att}$-classifier dataset. As shown in Table 1, our approach detects 13 design patterns with promising scores. The strategy and proxy patterns were predicted with precision higher than 95%. Five other design patterns were detected with a precision higher than 90%.

Table 1: Evaluation per each design pattern on $DPD_{Att}$

| Class | Precision | Recall | F1-Score |
|---|---|---|---|
| Singleton | 0.88 | 0.79 | 0.84 |
| Observer | 0.94 | 0.91 | 0.92 |
| Memento | 0.87 | 0.90 | 0.89 |
| Proxy | 0.97 | 0.91 | 0.94 |
| Prototype | 0.94 | 0.97 | 0.95 |
| Builder | 0.76 | 0.73 | 0.75 |
| Abstract Factory | 0.94 | 0.94 | 0.94 |
| Factory Method | 0.90 | 0.95 | 0.92 |
| Facade | 0.94 | 0.82 | 0.87 |
| Adapter | 0.80 | 0.97 | 0.88 |
| Decorator | 0.81 | 0.83 | 0.82 |
| Visitor | 0.78 | 0.95 | 0.86 |
| Unknown | 0.64 | 0.59 | 0.61 |
| Strategy | 1.00 | 0.87 | 0.93 |

**Comparison with related work.** The objective of this section is to compare our findings with related work that deal with design pattern detection. To this end, we consider a machine learning-based approach, called DPD$_F$, that uses word embedding (i.e., word2vec) to represent their features for the classifier [16], in addition to two metric-based approaches, MARPLE-DPD [25], and FeatureMap [20]. To perform this comparison, we specifically refer to Nazar et al. [16]'s corpus (i.e., $DPD_F$ corpus). Indeed, in their paper, the authors in [16] used the $DPD_F$ corpus for comparing DPD$_F$, MARPLE-DPD, and FeatureMap approaches. As shown in Fig.8, testing the $DPD_{Att}$ approach on the same $DPD_F$ corpus yields better results in terms of mean precision, mean recall, and mean F1-score. These results indicate superior performance compared to state-of-the-art approaches [16][25][20], demonstrating the effectiveness of our approach in addressing the problem of GoF design pattern detection in source code. The comparisons of our approach with the state-of-the-art per design pattern in terms of F1-score is presented in Table 2. It is worth noting that this comparison concerns only nine design patterns, which are Singleton, Observer, Abstract Factory, Factory Method, Adapter, Builder, Decorator, Visitor, and the "unknown" target since MARPLE-DPD [25], and FeatureMap [20] deal only with these patterns. As we can see from the table, we achieve more promising results than FeatureMap [20] and MARPLE-DPD [25], in addition to DPD$_F$ [16]. Indeed, our approach, DPD$_{Att}$, outperforms them in detecting eight out of nine GoF design patterns in Java. This is due to the capacity and performance of the feature extraction method for automatically extracting contextual information that encompasses the GoF design pattern features.

**Computational complexity** In this section, we evaluate the computational complexity of the feature extraction module. LLMs are generally evaluated by the number of parameters they have [2]. The count of parameters is intricately

Table 2: Comparison of the $DPD_{Att}$ in term of F1-score per design pattern with the state-of-the-art

| **Design Pattern** | **FeatureMap** [20] | **MARPLE-DPD** [25] | **DPD**$_F$ [16] | **DPD**$_{Att}$ |
|---|---|---|---|---|
| Singleton | 0.66 | 0.72 | 0.74 | 0.83 |
| Observer | 0.49 | 0.51 | 0.85 | 0.89 |
| Builder | 0.61 | 0.55 | 0.83 | 0.58 |
| Abstract Factory | 0.52 | 0.76 | 0.93 | 1.0 |
| Factory Method | 0.55 | 0.81 | 0.78 | 0.98 |
| Adapter | 0.33 | 0.82 | 0.69 | 0.88 |
| Decorator | 0.23 | 0.59 | 0.78 | 0.83 |
| Visitor | 0.65 | 0.63 | 0.94 | 0.94 |
| Unknown | 0.72 | 0.54 | 0.73 | 0.92 |

linked to kernel sizes, input and output channels, and serves as an indicator of computational resource utilization, particularly memory, during both model training and detection processes. In our feature extraction process, we employ the encoder of the CodeT5+ model, which has 110 million frozen parameters. In addition, we evaluate the operational performance of our feature extractor using Floating Points of Operations (FLOPs). FLOPs represent the total number of multiplication and addition operations within the model, providing a measure of its computational complexity. The testing was conducted on a system equipped with an Intel Core i7 processor running at 2.30GHz, 16GB of RAM, and a NVIDIA Ge-Force GTX 1650 with Max-Q Design for video memory. Fig.9 shows the variation of FLOPs with respect to DPD$_{Att}$ corpus. The maximum number of floating operations by the encoder to extract features and construct the embedding vector is 998.92G, and it reflects on the Java class with the highest number of tokens from the DPD$_{Att}$ dataset. Whereas the minimum number of FLOPs is equal to 1.002G, and it corresponds to the Java code with the least number of tokens in DPD$_{Att}$.

## 5 Related work

This section overviews attention-based methods proposed in the literature to perform software engineering task. Then, it discusses existing design pattern detection approaches and concludes with a discussion.

### 5.1 Attention-based methods in software engineering

For software defect prediction, different attention-based approaches have been proposed in the literature. In [7], the authors employed attention mechanism to capture syntactic and semantic features of programs and use them to improve defect prediction. The proposed approach parses the abstract syntax trees (ASTs) of programs and extracts them as vectors. Then, it encodes vectors and employs the attention mechanism to further generate significant features for

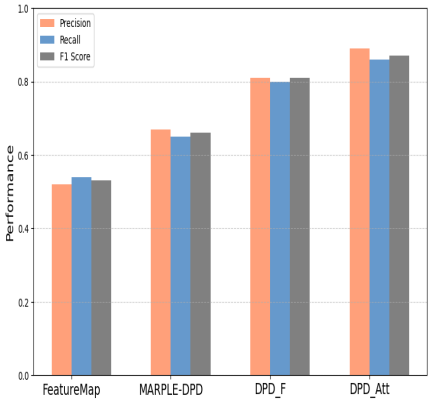


Fig. 8: Comparison of the $DPD_{Att}$ approach with the $\text{DPD}_F$ [16], MARPLE-DPD [25], and FeatureMap [20]

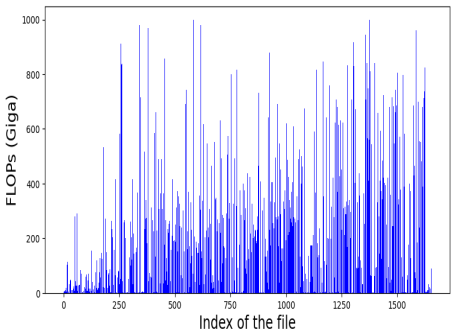


Fig. 9: FLOPs variations on the $\text{DPD}_{Att}$ dataset

accurate defect prediction. In [14], the authors proposed an attention-based approach for statement-level software defect prediction. In this work, the authors define a set of 32 statement-level metrics. Then each statement is labeled to make a three-dimensional vector for automatic learning. The attention mechanism is used to generate important features and improve accuracy. In [22], the authors proposed an attention-based method for self-admitted technical debt detection. Indeed, Self-Admitted Technical Debt (SATD) is a kind of technical debt that seeks to capture technical debts that are intentionally introduced by developers during the software development process. To extract the sequential properties of self-admitted technical debts from code comments, the proposed solution uses a positional encoder and a Bi-directional Long Short-Term Memory (Bi-LSTM) network [26]. Then, it leverages a variety of attention techniques to emphasize the importance of the automatically generated features that contributed to the detection of SATDs.

### 5.2 Machine Learning-based pattern detection

The use of machine learning techniques for pattern detection was widely addressed in the literature. In [24], the authors proposed a machine learning-based approach for the detection of security patterns in code. To be independent of the programming language, the proposed solution split the code in two vectorial representations: control flow and data flow. The control flow section is treated with word embedding to preserve the semantics of the code, and by clustering of the resulting embedding vectors to reduce the dependency on the lexical structure of the program. The authors in [10] presents a machine learning-based approach for the detection of architectural patterns, namely MVP (Model–View–Presenter)

and MVVM (Model–View–ViewModel). In the paper, the authors performed analysis using nine popular machine learning models and established a set of source code metrics that can be used to detect MVVM and MVP architectural patterns using machine learning. Various studies have been interested in GoF design pattern detection. In [4], the authors proposed a probabilistic approach for GoF design pattern detection. In this work, the authors used neural networks and regression analysis to measure the possibility of the presence of the design pattern in the source code. The authors in [4] applied a correlation feature selection method to match the system design to the design pattern. Indeed, the authors describe in the paper a graph matching algorithm that uses a correlation-based feature selection technique to identify design pattern instances in system design. Different studies aim to train machine learning classifiers to detect GoF design pattern participants [16][11]. These approaches used code features to construct a text representation of Java classes. Indeed, they extract semantic and structural features from call graphs (SCG), which are in turn embedded in a text file as a natural language in a syntactic and semantic representation (SSLR) document. Then, a text embedding algorithm on the SSLR representation is applied, such as word2vec [16][11], to produce a vector representation for the Java source code. This vector served to train a supervised machine learning classifier.

### 5.3 Discussion

In this paper, we aim to propose a fully automated approach for GoF design pattern detection. Compared to related work [16][11], which include a pre-processing step for source code embedding, the proposed solution in this paper is based on a code embedding model, which eliminates this step. Indeed, as far as we know, all the existing approaches that deal with feature-based pattern detection used neural processing language techniques for the embedding step, such as word embedding [16], which requires feature engineering efforts to increase the accuracy of the design pattern detector. In fact, code features relevant to design pattern implementations, in some cases manually performed [11][15], must be defined. This stage can be hard and time-consuming since for a single design pattern that can have multiple disjunctive variations, a set of features that describe each single variant must be defined and a method to extract them must be implemented. For instance, Nacef et al. [15] define 33 features for the singleton design pattern only.

## 6 Conclusion

In this paper, we have presented a method for automatic detection of design patterns. The proposed approach includes a transformer-based feature extraction step. Features refer to code attributes that represent design pattern implementations and will help in detecting design patterns during the classification task. To this end, we build the $\text{DPD}_{Att}$ corpus, which consists of 1645 labeled Java files covering 13 GoF design patterns. Then, following an unsupervised approach and

considering the $\text{DPD}_{Att}$ corpus as input, the feature extraction module learns from the code to automatically extract syntactic and semantic features. The features are then encoded as embedding vectors to train a classifier for the detection task. To evaluate the efficacy of the proposed approach, we apply four commonly used statistical measures, namely accuracy, precision, recall, and F1-Score. We also build a heat-map to showcase the efficiency of our chosen classifier. Empirical evaluation shows that the $\text{DPD}_{Att}$ approach shows promising results with an accuracy score of 86%, precision of 87%, recall of 86%, and F1-score of 86%. The comparison with the related work shows that the proposed approach outperforms three existing methods.

While our results are promising, as future work, we aim to extend the $\text{DPD}_{Att}$ corpus to consider more variants and extend to more design patterns. We also aim to use feature localization approaches to locate design pattern implementations among participants in a source code.

**Data Availability.** The data that supports the findings is available through Zenodo [5]

## References


1. Allamanis, M., Sutton, C.: Mining Source Code Repositories at Massive Scale using Language Modeling. In: The 10th Working Conference on Mining Software Repositories. pp. 207–216. IEEE (2013)
2. Bae, H., Deeb, A., Fleury, A., Zhu, K.: Complexitynet: Increasing llm inference efficiency by learning task complexity. arXiv preprint arXiv:2312.11511 (2023)
3. Church, K.W.: Word2vec. Natural Language Engineering **23**(1), 155–162 (2017)
4. Dewangan, S., Rao, R.S.: Design pattern detection by using correlation feature selection technique. In: 2022 IEEE 11th International Conference on Communication Systems and Network Technologies (CSNT). pp. 641–645. IEEE (2022)
5. Dong, J., Zhao, Y., Sun, Y.: A matrix-based approach to recovering design patterns. IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans **39**(6), 1271–1282 (2009)
6. Dwivedi, A.K., Tirkey, A., Rath, S.K.: Applying software metrics for the mining of design pattern. In: 2016 IEEE Uttar Pradesh Section International Conference on Electrical, Computer and Electronics Engineering (UPCON). pp. 426–431. IEEE (2016)
7. Fan, G., Diao, X., Yu, H., Yang, K., Chen, L., et al.: Software defect prediction via attention-based recurrent neural network. Scientific Programming **2019** (2019)
8. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Pearson Deutschland GmbH (1995)
9. Issaoui, I., Bouassida, N., Ben-Abdallah, H.: Using metric-based filtering to improve design pattern detection approaches. Innovations in Systems and Software Engineering **11**, 39–53 (2015)
10. Komolov, S., Dlamini, G., Megha, S., Mazzara, M.: Towards predicting architectural design patterns: A machine learning approach. Computers **11**(10), 151 (2022)


[5] https://doi.org/10.5281/zenodo.11584286

11. Kouli, M., Rasoolzadegan, A.: A feature-based method for detecting design patterns in source code. Symmetry **14**(7), 1491 (2022)
12. Li, L., Wu, Y., Ye, M.: Experimental comparisons of multi-class classifiers. Informatica **39**(1) (2015)
13. Misra, P., Yadav, A.S.: Improving the classification accuracy using recursive feature elimination with cross-validation. Int. J. Emerg. Technol **11**(3), 659–665 (2020)
14. Munir, H.S., Ren, S., Mustafa, M., Siddique, C.N., Qayyum, S.: Attention based gru-lstm for software defect prediction. Plos one **16**(3), e0247444 (2021)
15. Nacef, A., Bahroun, S., Khalfallah, A., Ahmed, S.B.: Features and supervised machine learning based method for singleton design pattern variants detection. Proceedings of the 18th International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE 2023, Prague, Czech Republic, April 24-25, 2023 pp. 226–237 (2023)
16. Nazar, N., Aleti, A., Zheng, Y.: Feature-based software design pattern detection. Journal of Systems and Software **185**, 111179 (2022)
17. Rasool, G., Philippow, I., Mäder, P.: Design pattern recovery based on annotations. Advances in Engineering Software **41**(4), 519–526 (2010)
18. Richards, M., Ford, N.: Fundamentals of software architecture: an engineering approach. O'Reilly Media (2020)
19. Ruaro, N., Zeng, K., Dresel, L., Polino, M., Bao, T., Continella, A., Zanero, S., Kruegel, C., Vigna, G.: Syml: Guiding symbolic execution toward vulnerable states through pattern learning. In: Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses. pp. 456–468 (2021)
20. Thaller, H., Linsbauer, L., Egyed, A.: Feature maps: A comprehensible software representation for design pattern detection. In: 2019 IEEE 26th international conference on software analysis, evolution and reengineering (SANER). pp. 207–217. IEEE (2019)
21. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., Polosukhin, I.: Attention is all you need. Advances in neural information processing systems **30** (2017)
22. Wang, X., Liu, J., Li, L., Chen, X., Liu, X., Wu, H.: Detecting and explaining self-admitted technical debts with attention-based neural networks. In: Proceedings of the 35th IEEE/ACM international conference on automated software engineering. pp. 871–882 (2020)
23. Wang, Y., Le, H., Gotmare, A.D., Bui, N.D., Li, J., Hoi, S.C.: Codet5+: Open code large language models for code understanding and generation. arXiv preprint arXiv:2305.07922 (2023)
24. Zaharia, S., Rebedea, T., Trausan-Matu, S.: Machine learning-based security pattern recognition techniques for code developers. Applied Sciences **12**(23), 12463 (2022)
25. Zanoni, M., Fontana, F.A., Stella, F.: On applying machine learning techniques for design pattern detection. Journal of Systems and Software **103**, 102–117 (2015)
26. Zhou, P., Shi, W., Tian, J., Qi, Z., Li, B., Hao, H., Xu, B.: Attention-based bidirectional long short-term memory networks for relation classification. In: Proceedings of the 54th annual meeting of the association for computational linguistics (volume 2: Short papers). pp. 207–212 (2016)
27. Zhu, H., Bayley, I., Shan, L., Amphlett, R.: Tool support for design pattern recognition at model level. In: 2009 33rd Annual IEEE International Computer Software and Applications Conference. vol. 1, pp. 228–233. IEEE (2009)