The Journal of Systems & Software 185 (2022) 111179

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

# Feature-based software design pattern detection[☆]

Najam Nazar[*], Aldeida Aleti, Yaokun Zheng

*Faculty of Information Technology, Monash University, Australia*

ARTICLE INFO

*Article history:*
Received 30 November 2020
Received in revised form 9 October 2021
Accepted 1 December 2021
Available online 8 December 2021

*Keywords:*
Software design patterns
Code features
Word-space-model
Machine learning

ABSTRACT

Software design patterns are standard solutions to common problems in software design and architecture. Knowing that a particular module implements a design pattern is a shortcut to design comprehension. Manually detecting design patterns is a time consuming and challenging task, therefore, researchers have proposed automatic design pattern detection techniques. However, these techniques show low performance for certain design patterns. In this work, we introduce a design pattern detection approach, $DPD_F$ that improves the performance over the state-of-the-art by using code features with machine learning classifiers to automatically train a design pattern detector. $DPD_F$ creates a semantic representation of Java source code using the code features and the call graph, and applies the *Word2Vec* algorithm on the semantic representation to construct the word-space geometric model of the Java source code. $DPD_F$ then builds a machine learning classifier trained on a labelled dataset and identifies software design patterns with over 80% Precision and over 79% Recall. Additionally, we have compared $DPD_F$ with two existing design pattern detection techniques namely *FeatureMaps* & *MARPLE-DPD*. Empirical results demonstrate that our approach outperforms the existing approaches by approximately 35% and 15% respectively in terms of Precision. The run-time performance also supports the practical applicability of our classifier.



## 1. Introduction

Design pattern detection is an active research field and in recent years gained enormous attention by software engineering professionals (Mayvan and Rasoolzadegan, 2017). Kuchana (2004) defines software design patterns as *"recurring solutions to common problems in a given context and system of forces"*. Since their popularisation by the *'Gang of Four'* (GoF) (Gamma et al., 1995), design patterns have been widely adopted by software professionals to improve the quality of software, and to facilitate code reuse and refactoring. Recognising that a particular software module implements a design pattern can greatly assist in program comprehension, and consequently improve software maintenance (Prechelt et al., 2002). Due to the increasing complexity of software projects and the differences in coding styles of software developers, it is difficult to detect in code where the patterns have been implemented.

Automatic detection of design patterns has appeared to be useful in assisting software developers to quickly and correctly comprehend and maintain unfamiliar source code, ultimately leading to higher developer productivity (Walter and Alkhaeir, 2016; Scanniello et al., 2015; Gaitani et al., 2015; Christopoulou et al., 2012). The majority of existing methods reverse engineer the source code to identify the design patterns (Detten et al., 2010; Lucia et al., 2011) or build tools to detect design patterns in the source code e.g., (Hautamäki, 2005; Moreno and Marcus, 2012), or utilise code metrics e.g., (Uchiyama et al., 2011). Although it is relatively easy to obtain structural elements from the source code such as classes, attributes, methods etc. and transform them into graphs or other representations, they show low accuracy and fail to effectively predict the majority of design patterns (Yu et al., 2018). At the same time, capturing semantic (lexical) information from the source code is challenging and has not been fully attempted yet in identifying design patterns. Additionally, translating source code to natural language text has been effectively used in generating source code summaries with high accuracy (Hu et al., 2018; McBurney and McMillan, 2015, 2016; Moreno et al., 2013), including our own work on identifying summary sentences from code fragments (Nazar et al., 2016). Based on these observations, we hypothesise that the lexical-based (basic) code features extracted from the source code will increase the accuracy of design pattern detection.

In this paper, we introduce a Feature-Based Design Pattern Detection ($DPD_F$) approach that uses source code features — both structural and lexical, and employs machine learning classifiers to predict a wide range of GOF design patterns, with higher accuracy compared to the state-of-the-art. Machine learning has been applied for DPD in the past, e.g., Fontana et al. (2011), however

[☆] Editor: Leandro Minku.

[*] Corresponding author.
*E-mail address:* najam.nazar@monash.edu (N. Nazar).

https://doi.org/10.1016/j.jss.2021.111179
0164-1212/

our approach is the first to employ lexical-based code features. $\text{DPD}_F$ builds a call graph and extracts 15 source code features to generate a Software Syntactic and Lexical Representation (SSLR). The SSLR provides the lexical and syntactic information of the Java files as well as the relationships between the files' classes, methods etc., in a natural language form. Using SSLR as an input, we build a word-space geometrical model of Java files by applying the *Word2Vec* algorithm. We train a supervised machine learning classifier, $\text{DPD}_F$ on design patterns using the labelled dataset and a geometrical model for recognising twelve commonly used GOF software design patterns.

To evaluate our approach, we label a corpus of 1300 Java files extracted from a publicly available Github Java Corpus (Allamanis and Sutton, 2013), which we refer to as *$\text{DPD}_F$ Corpus*. We use an online tool *'CodeLabeller'*[1], where expert raters annotate design patterns. We statistically evaluate the performance of $\text{DPD}_F$ by calculating the Precision, Recall and F1-Score measures and compare our approach to two existing software design pattern detection approaches, namely *FeatureMaps* and *MARPLE-DPD*. Empirical results show that our approach is effective in recognising twelve GOF software design patterns with high Precision (80%) and low error rate (20%). Furthermore, $\text{DPD}_F$ outperforms the benchmark approaches (i.e. *FeatureMaps* & *MARPLE-DPD*) by 35% and 15% respectively in terms of Precision.

**Contributions:** This paper makes the following contributions:

- We introduce a novel approach called *Feature-Based Design Pattern Detector* ($\text{DPD}_F$) that uses 15 source code features to detect software design patterns.
- We build a large corpus (i.e. $\text{DPD}_F$ Corpus) consisting of 1300 Java files, which are labelled by the expert software engineers using the *Codelabeller* tool .
- We demonstrate that our approach outperforms two existing approaches with substantial margins in terms of Precision, Recall and F1-Score.

**Paper Organisation:** The remainder of this paper is organised as follows. We discuss the preliminaries and relevant background of the related technologies, in particular, design patterns, code fragments, word space models and machine learning in Section 2. Section 3 discusses our study design, including the research questions, the labelling and selection of corpora, the source code features used and selected, the application of Word2Vec for building an N-gram model and ending with a discussion of our machine learning classifier. Section 4 presents results with respect to our research questions and evaluate them. We discuss both internal and external threats to the validity of our study in Section 5 and the related work is presented in Section 6. Finally, Section 7 concludes the paper and briefly points out potential future directions.

## 2. Preliminaries

In the following subsections, we briefly discuss the relevent background and concepts used in this study — that are design patterns, code features, word-space embeddings or models and machine learning.

### 2.1. Design patterns

We consider and identify the following twelve GOF design patterns namely *Abstract Factory*, *Adapter*, *Builder*, *Decorator*, *Factory Method*, *Façade*, *Memento*, *Observer*, *Prototype*, *Proxy*, *Singleton* and *Visitor*, in our proposed approach. These patterns cover all three categories (or aspects) of GoF patterns i.e. creational, structural and behavioural patterns. Creational patterns include Builder, Abstract Factory, Factory Method, Prototype and Singleton patterns whereas, structural patterns include Adapter, Decorator, Façade and Proxy patterns. We have considered three behavioural patterns for this study and that are Memento, Observer and Visitor.

### 2.2. Code features

Code features are static source code attributes that are extracted by examining the source code (McBurney et al., 2018), such as the size of the code elements (e.g., line of code), complexity of code such as if/else blocks, object-oriented attributes in code such as inheritance etc., and source code constructs that are uniquely identifiable names for a construct, for example class name, method name etc. Zanoni et al. (2015) called features as code entities that are the names given to any code construct that is uniquely identifiable by its name and the name of its containers. In the object-oriented paradigm, code entities are classes, interfaces, enums, annotations, etc., methods and attributes. Previously, Nazar et al. (2016) used 21 textual features to identify summary lines for code fragments.

By examining aforementioned studies, we decide to use a mixture of structural and lexical object-oriented code constructs to investigate if they can be useful in identifying the design patterns from the source code. These features (as discussed in Section 3.3) capture the behavioural, structural and creational aspects of the code as needed for design patterns (as discussed in Section 2.1). In this paper, we follow the term features as we believe that they are more relevant to design patterns and depict the code structure and semantics better than low level entities discussed in relevant studies.

### 2.3. Word embeddings

Word space models are abstract representations of the meaning of words, encoded as vectors in a high dimensional space (Salton et al., 1975). A word vector space is constructed by counting co-occurrences of pairs of words in a text corpus, building a large square $n$-by-$n$ matrix where $n$ is the size of the vocabulary and the cell $(i, j)$ contains the number of times the word $i$ has been observed in co-occurrence with the word $j$ in the corpus. The $i$th row in a co-occurrence matrix is an $n$-dimensional vector that acts as a distributional representation of the $i$th word in the vocabulary.

The key to using a vector representation to compute the semantic relatedness of words lies in the *Distributional Hypothesis* (DH) (Harris, 1954). The DH states that: *"words that occur in the same contexts tend to have similar meaning"*, therefore allowing the computational linguist to approximate a measure of how much two words are related in their meaning by computing a numeric value from a vector space model.

The similarity between two words is geometrically measurable with any distance metric. The most widespread metric for this purpose is the cosine similarity, defined as the cosine of the angle between two vectors:

$$S(x, y) = \frac{x \cdot y}{\|x\| \|y\|}$$

Several techniques can be applied to reduce the dimensionality of the co-occurrence matrix. Latent Semantic Analysis (LSA), for instance, uses Singular Value Decomposition (SVD) to prune the less informative elements while preserving most of the topology of the vector space, and reducing the number of dimensions to the order of hundreds (Landauer and Dumais, 1997; Mikolov et al., 2013).

[1] www.codelabeller.org, verified on 08-10-21.

Recently, neural network based models have received increasing attention for their ability to compute dense, low-dimensional representations of words. To compute such representation, i.e., the word embeddings, several models rely on a huge amount of natural language texts from which a vector representation for each word is learned by a neural network. Their representations of the words are based on prediction as opposed to counting. Embedded vectors created using the predictive models such as *Word2Vec* have many advantages compared to LSA (Baroni et al., 2014). For instance, their ability to compute dense, low-dimensional predictive representations of words. Vector spaces created on word distributional representations have been successfully proven to encode word similarity and relatedness relations (Radinsky et al., 2011; Reisinger and Mooney, 2010; Ciobanu and Dinu, 2013; Collobert et al., 2011), and word embeddings have proven to be a useful feature in many natural language processing tasks (Collobert et al., 2011; Le and Mikolov, 2014; dos Santos and Zadrozny, 2014) in that they often encode semantically meaningful information of a word.

### 2.4. Machine learning

Machine learning (ML) is the study of computer programmes that learns from the data and improves automatically through experience (Mitchell, 1997). In ML the essential elements are data — structural such as text, unstructural or semi-structural such as source code, the model e.g., Support Vector Machines (SVM) or Random Forest (RF) (Cortes and Vapnik, 1995; Ho, 1995), and the evaluation procedure e.g., cross-validation. The types of machine learning algorithms differ in their approach, the type of data they input and output, and the type of task or problem that they intend to solve. In short, there are three major types of machine learning approaches that are supervised, unsupervised and semi-supervised. Supervised learning builds a mathematical model based on the labelled data to predict future results (Manning et al., 2008; Bishop, 2006). Unsupervised, on the other hand, learns how systems can infer a function to describe a hidden structure from unlabelled data (Bishop, 2006). Semi-supervised falls between supervised and unsupervised approaches, since it uses both labelled and unlabelled data for training — typically a small amount of labelled data and a large amount of unlabelled data (Manning et al., 2008; Bishop, 2006).

There are two major types of supervised learning namely, classification and regression. The classification methods usually use two sets, a training set and a test set. The training set is used for learning some classifiers and requires a primary group of labelled individuals, in which the category related to each individual is obvious from its label. The test set is used to measure the efficiency of the learned classifiers and includes labelled individuals which do not participate in learning classifiers. Regression, on the other hand, allows us to predict a continuous outcome of the variable we intend to find.

Design pattern detection can be categorised as a classification problem where classes containing the pattern can be labelled by expert raters. Thus, supervised classification learning corresponding to the ground truth that a class contains or does not contain a design pattern can be used to predict design patterns. In this study, RF and SVM based classifiers are considered from benchmark studies whereas our $\text{DPD}_F$ classifier builds a design pattern detection model that classifies given classes. In this paper, we have utilised supervised learning classifiers to identify design patterns from a given source code.

## 3. Study design

In this section, we describe our research questions and justify their use in applying the overall motivation of design pattern detection from source code. We discuss the methodology of identifying the patterns from source code where we talk about collection of data and features alongwith the identification of patterns using the research questions.

### 3.1. Research questions

This study seeks to identify design patterns through code features and *Word2Vec* algorithm along with the application of supervised machine learning. In doing so, we aim to examine the relationship between code features and the design patterns. Therefore, we pose the following three Research Question (RQs):

1. **RQ1**: How effective is $\text{DPD}_F$ in detecting software design patterns?
   The rationale behind *RQ1* is to determine whether our approach identifies design patterns accurately and effectively. For this purpose, we statistically evaluate our classifiers using standard statistical measures of Precision, Recall and F1-Score.
2. **RQ2**: What is the error-rate of $\text{DPD}_F$?
   For addressing *RQ2*, we build a confusion matrix of the classifier to observe how well our classifier identifies the percentage of pattern instances from the labelled data and percentage of instances missed by the classifier.
3. **RQ3**: Comparing with existing studies how well $\text{DPD}_F$ performs?
   It is important to compare our approach with the existing studies to further evaluate the efficacy of our approach. To do so, we select two studies that are *FeatureMaps* proposed by Thaller et al. (2019) & *MARPLE-DPD* proposed by Zanoni et al. (2015) respectively as a benchmark studies for comparing our approach. These studies as well as the details of the benchmark corpus are discussed in Section 3.2.

### 3.2. Methodology

Our methodology to address research questions is as follows: First, we collect the corpus containing Java projects and discuss the labelling of corpus. Next, we discuss how the selection of code features and its application to extract the syntactic and semantic representation (SSLR) of the corpus. After that, we apply the *Word2Vec* algorithm on the SSLR to create the geometric representation that can be read by the machine. In the end, we train supervised classifiers on the labelled corpus to predict design pattern instances from the geometric representation of Java files.

In the following subsections, we discuss these steps one by one.

#### 3.2.1. Data collection

Existing datasets labelled with design patterns are either too small or not publicly available. We have found one publicly available corpus of Java projects called P-MART.[2] However, we decide to use it as a benchmark corpus. Therefore, we create a new corpus *$\text{DPD}_F$-Corpus*, which we label with the respective design pattern. The aim is to increase the size of the existing corpora and make it publicly available for future researchers.

**$\text{DPD}_F$-Corpus:** We build the new corpus exclusively of design patterns from the *Github Java Corpus* (GJC) (Allamanis and Sutton,

[2] http://www.ptidej.net/tools/designpatterns/, verified on 08-10-21.

**Table 1**
The number of instances of each pattern in a labelled $DPD_F$-Corpus.

| Patterns | # | Patterns | # |
|---|---|---|---|
| Abstract Factory | 100 | None | 100 |
| Adapter | 100 | Observer | 100 |
| Builder | 100 | Prototype | 100 |
| Decorator | 100 | Proxy | 100 |
| Factory Method | 100 | Singleton | 100 |
| Façade | 100 | Visitor | 100 |
| Memento | 100 | | |

**Table 2**
The number of instances of each pattern in a P-MART corpus.

| Patterns | # | Patterns | # |
|---|---|---|---|
| Abstract Factory | 241 | Observer | 137 |
| Adapter | 241 | Memento | 15 |
| Builder | 43 | Prototype | 32 |
| Decorator | 63 | Proxy | 3 |
| Factory Method | 102 | Singleton | 13 |
| Façade | 11 | Visitor | 139 |

2013), which is the largest publicly available corpus consisting exclusively of open source Java projects collected from GitHub. In total, the GJC contains 2, 127, 357 Java files in 14,436 Java projects. As the first step, we remove unnecessary files such as unit test cases (JUnit) or user interface files such as HTML, CSS etc. from the GJC, as these files do not normally implement design patterns. From the remainder of GJC, we select a subset of the GJC projects using the correction of finite population approach,[3] to determine the size of the final corpus that can be labelled and trained by the machine learning classifier. The final corpus has 1300 files and we refer to it as *$DPD_F$-Corpus*. To ensure that sufficient instances are available for training and testing the machine learning algorithms, we select exact 100 instances for each of the tweleve design patterns (and none labels) as shown in Table 1.

**Benchmark Corpus:** In addition to the *$DPD_F$* corpus, we employ an existing dataset — P-MART used by benchmark studies (Zanoni et al., 2015; Thaller et al., 2019), which contains 4,242 files from 9 projects, which are QuickUML, Lexi, JRefactory, Netbeans, JUnit, JHotDraw, MapperXML, Apache Nutch and PMD. On exploring the P-MART we found that it contains uneven number of design patterns and requires surgery to fit for our purpose. The P-MART corpus contains 1039 files that are labelled as design patterns we plan to identify in this study. Table 2 shows the distribution of tweleve design patterns in the P-MART Corpus. On exploring the P-MART we found that it contains an uneven number of design patterns as shown in Table 2.

### 3.2.2. Data labelling

We use an online tool *CodeLabeller* that is built using Node and Angular JavaScript languages to label corpora via crowd knowledge. Each file in the corpora is labelled by at least three annotators, who have at least 2 years of programming experience in Java programming language, applied design patterns in the projects they have worked on, and taught software engineering and related units at the university level. We have asked the faculty members, postgraduate and doctoral students majoring in Software Engineering and Computer Science at the Faculty of IT, Monash University to label the corpora through our online tool. Some files are labelled as *'None'*, which means that they do not implement or contain any of the design patterns. We have added None files in our corpus solely to check how well our classifier identifies files that do not contain a design pattern. Furthermore, if a pattern is implemented in more than one file, all the files that are part of the pattern are labelled with the respective pattern. For example, a Façade pattern has two files: interface and implementation, both of which are labelled as Façade.

**Raters Agreement:** Labelling is a subjective process and there is a possibility that a file is labelled differently by different annotators. Following a similar approach to Nazar et al. (2016), we perform a Cohen's Kappa Test (Cohen, 1960) to measure the level of agreement among annotators. For the $DPD_F$ corpus, the kappa K-value is *0.74*, showing a medium to high level of agreement among raters (Cohen, 1960; Carletta, 1996). We have not calculated the Kappa value for the benchmark corpus.

## 3.3. Feature extraction

Feature extraction deals with the inherent complexity of programming languages by extracting high-level concepts that in later stages can be used to successfully find pattern instances. Structurally, design patterns describe classes and their loose arrangements and communication paths. Consequently, extracted features ideally capture these arrangements and their relationships to improve reasoning in later stages. As discussed in Section 2.2 the source code features are high level code constructs that are used to extract information from the source code. Following a similar approach with McBurney et al. (2018) &Nazar et al. (2016), we have selected 15 features that are based on the syntactic and semantic (the linguistic meanings) constructs of Java source code.

As listed in Table 3, these features are related to *class labels*, *method names*, *modifiers*, *attributes*, *n-grams*, *method return type*, *number of parameters*, *the number of lines in a method*, *incoming methods* and *outgoing methods*. We may informally categorise these features into two major groups that are class level features (4 features in total), and method level features (11 features in total).

### 3.3.1. Class-level features

A class is a user-defined blueprint or prototype from which objects are created and in the Java language, the class name begins with a capital letter so the first code feature we select is a *'ClassName'* i.e., the name of the Java class. The second feature – feature 2 – is about the access modifiers in Java language i.e., public, private, protected and default. Next two features are features 3 and 4 – implements & extends are Java keywords – and related to the inheritance principle of object-oriented languages. Inheritance is key to many patterns such as observer, abstract factory etc.; therefore, it is important to capture it. In Java language, there are two keywords for inheritance i.e. *extends* when a class inherits a class and *implements* when a class inherits an interface. An interface can extend another interface in the same way that a class can extend another class; therefore, the extends keyword can also be used to extend an interface.

### 3.3.2. Method-level features

A Java method is a collection of statements that are grouped together to perform an operation. In general, a method in Java contains a name, which begins with a small letter in a camelCase format, one or more parameters (sometimes known as attributes) enclosed in parentheses and a return type — these are features 5, 6 & 7 respectively in our case. The Feature 8 measures properties of the statements inside the method's body. These statements can be local attributes, conditional statements such as if, else, switch, or an assignment statement etc. Feature 9 measures the number of variables or attributes in the method — some methods may have local scoped variables while some may not. Feature 10 is about the number of methods called within a method, whereas

[3] https://www.surveysystem.com/sscalc.htm verified on 08-10-21.

**Table 3**
The fifteen source code features & their descriptions.

| No. | Features | Description |
|---|---|---|
| 1 | ClassName | Name of Java Class. |
| 2 | ClassModifiers | Public, Protected, Private Keywords etc. |
| 3 | ClassImplements | A binary feature (0/1) if a class implements an interface. |
| 4 | ClassExtends | A binary feature (0/1) if a class extends another class. |
| 5 | MethodName | Method name in a class. |
| 6 | MethodParam | Method parameters (arguments). |
| 7 | MethodReturnType | A method that returns something (void, int etc.) or a method having a return keyword. |
| 8 | MethodBodyLineType | Type of code in a method's body e.g. assignment statement, condition statements etc. |
| 9 | MethodNumVariables | Number of variables/attributes in a method. |
| 10 | MethodNumMethods | Number of method calls in a class. |
| 11 | MethodNumLine | Number of lines in a method. |
| 12 | MethodIncomingMethod | Number of methods a method calls. |
| 13 | MethodIncomingName | Name of methods a method calls. |
| 14 | MethodOutgoingMethod | Number of outgoing methods. |
| 15 | MethodOutgoingName | Name of outgoing methods. |

feature 11 calculates the number of lines within a method's body. 'MethodIncomingMethod' – feature 12 – is about the number of methods a given method calls. For example, Method A calls Methods B, C and D then the value for 'MethodIncomingMethod' is 3. The difference between feature 10 and 12 is that feature 10 numbers all the methods which are called within a method that can be the methods from the same class or from other classes. Whereas, feature 12 only provides the number of methods that belong to external classes. For example a method A in class A has called method B from class B and Method C from class A then the value for 'MethodIncomingMethod' will be 1. Feature 13 is related to method incoming function as it illustrates the names of methods that call Method A. Similarly, features 14 and 15 are related to each other as they mention the number of methods a given method – say method A – has been called from and their names.

### 3.4. Feature-based design pattern detector

As shown in Fig. 2, the $\mathrm{DPD}_F$ approach has three main steps namely *Preprocessing*, *Model Building* and *Machine Learning Classification*. The Preprocessing step includes building a call graph and generating the SSLR representation by parsing the Java source code with code features. Once the code is parsed into natural language representation (SSLR), we apply the *Word2Vec* algorithm on SSLR to generate a Java Embedded Model (JEM) in the model building step. The JEM is used as an input to a supervised machine learning classifier trained on design patterns to predict design patterns. In the subsequent sections, we discuss these steps in more detail.

#### 3.4.1. Preprocessing

Translation between source code and natural language is challenging due to the structure of the source code. One simple way to model source code is to just view it as plain text. However, in such a way, the structure and semantic information will be omitted, which may cause inaccuracies in the generated natural language text (Hu et al., 2020). The structural information is extracted using Static Call Graphs (SCG), which encode how methods and functions call each others (Musco et al., 2017) and build a graph (edges and nodes) of flow of code to show the relationship between classes and methods in the source code.

Furthermore, building a call graph is useful as we need to encode the caller–callee classes and methods in the corpus which is the basis of generating the natural language text from the source code. Simply having a code hierarchy may not provide the complete (or enough) information we need for building the SSLR file. Based on these arguments, SCG will be more helpful in extracting the structural, behavioural and creational aspects of design pattern compared to AST. Thus, we generate a natural language representation of the source code i.e. SSLR by parsing the source code with code features and the SCG to encode the relationships between the code elements. The code elements are classes, methods, interfaces, etc. in the *$DPD_F$-Corpus*. The Fig. 1 illustrates a small subset (example) of SSLR file generated for the Observers.java file from the Platform project that is the part of the labelled corpus.

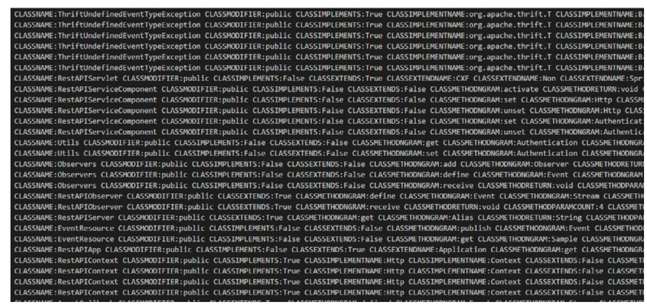

**Fig. 1.** An example (subset) of the SSLR file.

*Implementation:* We have implemented a parser in the Python programming language that parses Java files using the *Plyj*[4] library. We have written our own callgraph generator and feature extractor to build the SSLR representation of the corpus. Both files parse the input corpus and generate the SSLR representation. *Plyj* is a Java 7 parser written in Python using Ply,[5] and as the *'$DPD_F$ corpus'* contains files from the projects that were written before the year 2013, thus, the files are written prior to the release of Java 8 (pre- Java8) — using *Plyj* fulfils our requirement.

#### 3.4.2. Model building

We build a Java embedded n-gram model representation, by applying the *Word2Vec* algorithm (Baroni et al., 2014) on the SSLR representation of the $\mathrm{DPD}_F$ corpus generated in the preprocessing step. *Word2Vec* is a method to build distributed representation of words based on their contexts that works in two alternative fashions: that are the *Continuous Bag-of-Words* (CBOW)[6] model, which predicts the target word from source context words, and the Skip-Gram model, which does the inverse and predicts source context words from the target words.

The CBOW model is meant for learning relationships between pairs of words. In the CBOW model, context is represented by

[4] https://github.com/musiKk/plyj, verified on 08-10-21.
[5] https://github.com/dabeaz/ply verified on 08-10-21.
[6] also called the CBOW architecture or Vector Model

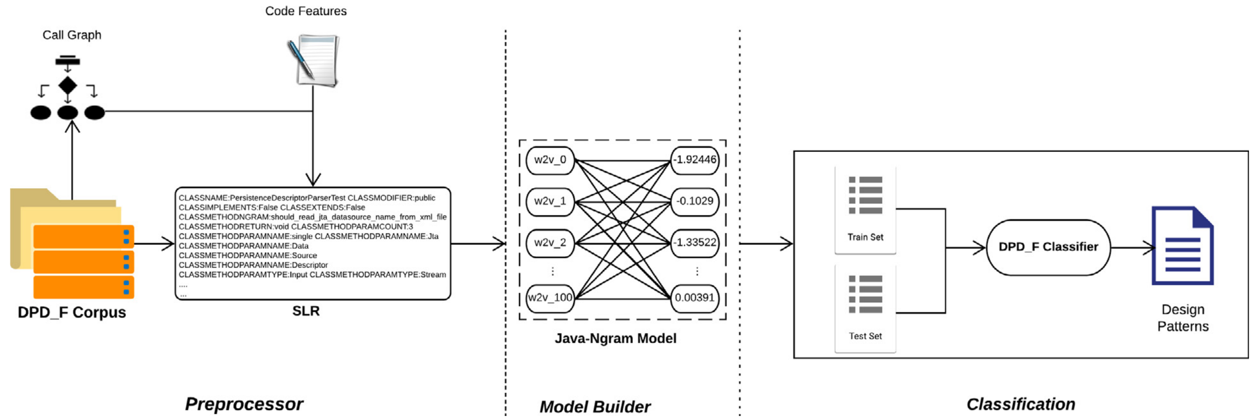


**Fig. 2.** Design Pattern Detection with Features ($DPD_F$).

multiple words for a given target word (Mitchell and Lapata, 2008). For example, we could use *'cat'* and *'tree'* as context words and *'climbed'* as the target word. The Skip-gram model architecture, on the other hand, predicts the source context words (surrounding words) given a target word (the centre word).

Hence, we create a Word Space Model of n-grams extracted from the content of the Java source code files using the source code features. We treat these files as if it was a natural language document, extracting the n-grams by segmenting the names of the most salient elements i.e., classes and methods etc., and consider them as words in the document. We then run the Word2Vec algorithm on the dataset to construct a high-dimensional representation of these n-grams, i.e., each n-gram is paired with a high-dimensional vector in a continuous dense space. The vectors representing the n-grams occurring in a Java class are composed by averaging them and the resulting vector is concatenated to the features extracted from the Java class with previous methods.

The CBOW architecture is meant for learning relationships between pairs of words, i.e., classes and methods mainly as in our case. We set a matrix of size 100 to build a 100-dimensional embedding model which is trained with *Word2Vec* that results in the vector representation of each ngram in our collection; the vector representation of a Java file will therefore be a function of the vector representations of its n-grams. Inspired by Mitchell and Lapata (2008), we produce embeddings for the Java files as a uniform linear combination of the embeddings of its constituent n-grams using Word2Vec.

In the 100-dimensional vector, each instance is associated with its project id, the class name, a 100-dimensional feature vector derived from a word embedding model built from the n-grams in the Java file, and a design pattern label where the design pattern is the target label to predict. These word embeddings provide a compact yet expressive feature representation for Java classes in a project source code. Despite being a programming language, and therefore a formal one, as opposed to natural language, several elements of the source code are natural language, including its name, the names of its methods and variables, and the comments etc.

*Implementation:* We use the *Word2vec* implementation provided by Řehůřek and Sojka (2010) from the *Python Gensim Library*[7] for generating Java n-gram model. The generated SSLR representation created in the preprocessing step is passed as an input and a Java Embedded Model is generated as a result of the model generation step.

**Table 4**
The $DPD_F$ classifier's learning parameters.

| Parameters | Values |
|---|---|
| Base Estimator | Random Forest |
| No of Estimations | 100 |
| Learning Rate | 1 |
| Algorithm | SAMME.R |

### *3.4.3. Machine classification*

By parsing the files in the labelled corpus, we are able to build a large dataset of these files paired with bags of n-grams relevant to each Java file. This structure is comparable to a natural language corpus, by drawing parallels between a Java source code (n-grams) and a text document. The augmented feature representation in Section 3.4.2 is used to train our $DPD_F$ classifier.

The $DPD_F$ classifier is an ensemble classifier that uses randomised decision trees as a base estimator. It implements a meta-estimator that fits a number of randomised decision trees on various subsamples of a dataset and uses ensembling to improve the predictive accuracy and control over-fitting. As we are dealing with the multi-class classification, we use SAMME-R as a boosting algorithm (Hastie et al., 2009).

*Implementation:* We use Python's *Scikit-learn* (Pedregosa et al., 2011) library for building our classifier and measuring the efficacy of the classification using standard statistical measures (discussed in Section 4.1). Table 4 provides a brief summary of our classifier's learning parameters.

**Cross-Validation:** Cross-Validation is a resampling procedure that is used to evaluate the machine learning models on a data sample. We have used the *Stratified K-Fold Cross-Validation* (SK-Fold CV) procedure for evaluating our machine learning model. The K-Fold Cross Validation is a type of Cross-Validation that involves randomly dividing the set of observations into K groups, or folds of approximately equal size (James et al., 2013). Normally, the first fold is treated as a validation set, and the method is fit on the remaining K minus 1 folds. A value of K=10 is the recommended value for k-fold cross-validation in the field of applied machine learning (Kuhn and Johnson, 2013), thus, we have used 10 folds per cross validation to validate our machine learning model. As there is a variation in sample size for each instance of design pattern in our corpus, it is important that each fold contains the same percentage of each design pattern instance to have fair prediction. Stratification is a variation of traditional K-Fold CV which is defined by Kuhn and Johnson (2013) as *'the splitting of data into folds ensuring that each fold has the same proportion of observations with a given categorical value'*

[7] https://radimrehurek.com/gensim_3.8.3/index.html, verified on 08-10-21.

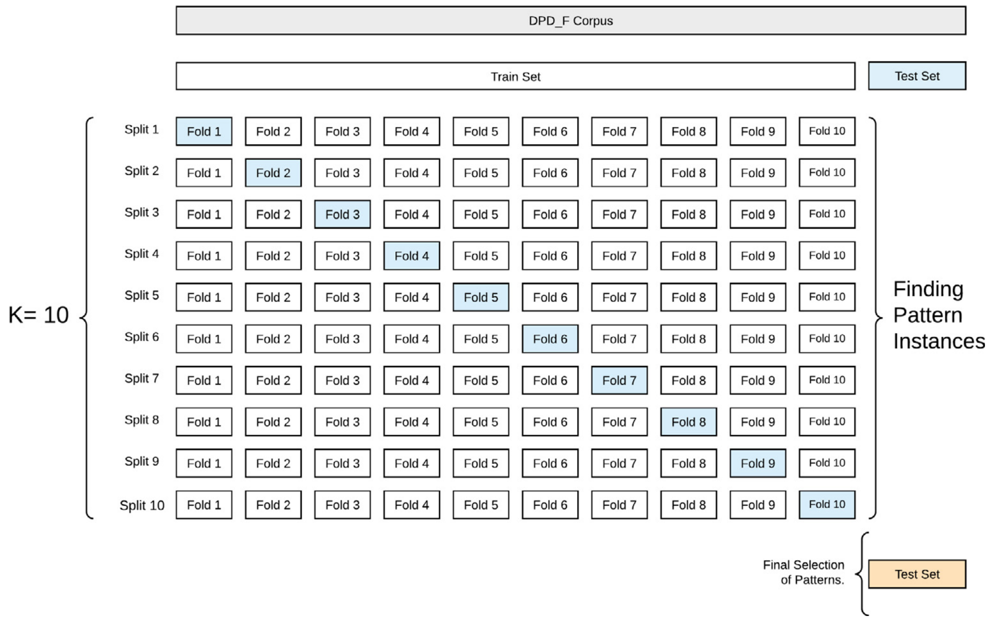


**Fig. 3.** Stratified K-Fold Cross-Validation.

that is pattern instances for our model. Fig. 3 illustrates our K-Fold Stratification procedure using 90/10 train test splits on the $DPD_F$ corpus.

*Implementation:* Our classifier used the sklearn StratifiedK-Fold,[8] implementation of cross-validation.

## 4. Results and evaluations

This section presents the results of our study where our answers for each research question are presented, evaluated and supported by our data and interpretation.

### 4.1. Evaluation criteria

We report the Precision, Recall, and F1-Score, which are the standard measures to statistically evaluate the efficacy of classifiers.

**Precision:** Precision *(P)* is defined as the fraction of instances of a classification that are correct, calculated as in Eq. (1):

$$P = \frac{TP}{TP + FP} \tag{1}$$

Where $TP$ stands for true positives and $FP$ stands for false positives.

**Recall:** Recall *(R)* is defined as the proportion of actual instances of a classification that are classified as such by the classifier as shown in Eq. (2). FN in the equations stands for false negatives.

$$R = \frac{TP}{TP + FN} \tag{2}$$

**F1-Score:** The F1-Score *(F1-S)* is measured as the harmonic mean of P and R and it is computed as follows:

$$F1 - S = 2 * \frac{P * R}{P + R} \tag{3}$$

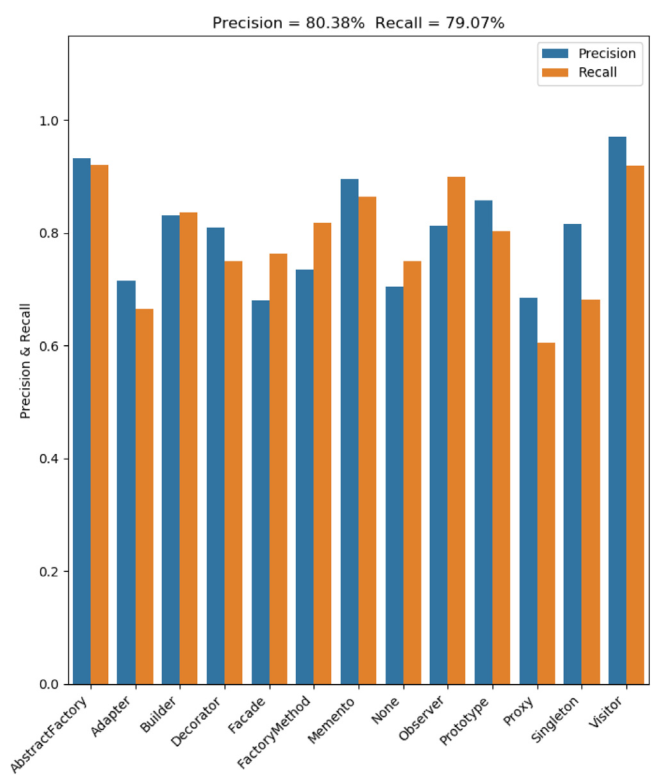


**Fig. 4.** Precision & Recall for the $DPD_F$ classifier.

*RQ1: Is $DPD_F$ effective in detecting software design patterns?*

Since the initialisation of the classifier is random, the results slightly vary at each run, although the difference is insignificant and there are no striking differences in performance, and the

[8] https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedKFold.html verified on 08-10-21.

**Table 5**
Precision, Recall and F-Score values for every label returned by the $DPD_F$ classifier.

| Classifier | $DPD_F$ | | |
|---|---|---|---|
| *Design Patterns* | *Precision (%)* | *Recall (%)* | *F1-Score (%)* |
| *Abstract Factory* | 93.27 | 92.08 | 92.46 |
| *Adapter* | 71.56 | 66.55 | 68.41 |
| *Builder* | 83.21 | 83.66 | 82.36 |
| *Decorator* | 80.99 | 75 | 77.34 |
| *Façade* | 68 | 76.27 | 71.06 |
| *Factory Method* | 89 | 83.88 | 85.79 |
| *Memento* | 89.66 | 86.44 | 87.45 |
| *Observer* | 81.26 | 90 | 85.06 |
| *Prototype* | 85.75 | 80.33 | 82.59 |
| *Proxy* | 68.51 | 60.55 | 62.86 |
| *Singleton* | 81.6 | 68.22 | 72.62 |
| *Visitor* | 97.07 | 91.88 | 93.39 |
| *None* | 70.44 | 75 | 71.91 |

overall performance is very similar for each iteration. Therefore, we calculate the weighted average of each turn to determine the final Precision, Recall and F1-Score for each label. Using the $DPD_F$ corpus, the $DPD_F$ classifier is able to predict most of the labels accurately, reaching a Precision of more than 80+% and Recall of 79+%. Fig. 4 and Table 5 shows the results obtained from the $DPD_F$ classifier divided into labels.

As evident from the results the most of the patterns are easily recognised by $DPD_F$ classifier, with *Visitor* having the highest Precision of approximately 97% and (interestingly) *Façade* with the lowest Precision of 68%. We believe the comparatively low Precision score for Facade and Proxy is due to their complex structure and usage in different context, e.g. *Proxy* can be mislead with the network proxies. Other patterns such as *Abstract Factory*, *Factory Method*, *Memento* and *Builder* are very well recognised by $DPD_F$ with over 80% Precision. In conclusion, our $DPD_F$ classifier identifies all patterns on average with over 80% of Precision and 79% of Recall.

*RQ2: What is the error rate of $DPD_F$?*

It is important to find that if the classifier misses some instances while training or predicting the instances or misidentifies these instances. We calculate the misclassification rate or error rate for this purpose.

Error rate is the ratio of how often the classifier is wrong or predicts the instances incorrectly. We compute the *confusion matrix* for $DPD_F$, which is a table that is often used to describe the performance of a classifier on a set of test data for which the true values are known. The results in Fig. 5 show that $DPD_F$ performs very well in detecting the absence of a design pattern, i.e., a 'none' — 75 out of 100 instances are true values. Similarly, 86 out of 100 instances for Visitor, 90 out of 100 instances for Observer, 85 out of 100 for Memento, 82 out of 100 for Builder and Abstract Factory are truly predicted by $DFD_F$. Looking at the confusion matrix, some of the instances are wrongly identified or missed by the classifier. For example, seven instances of Observer pattern are identified as Adapter pattern. A fair number of instances for Singleton, Proxy and Adapter are missed by the classifier and the truly predicted instances for these patterns are slightly lower than the other patterns. Fig. 5 shows the confusion matrix for $DPD_F$ classifier. Overall, the misclassification or the error rate is less than 20% and $DPD_F$ has correctly identified 80% of the instances of the all design patterns with high Precision and Recall.

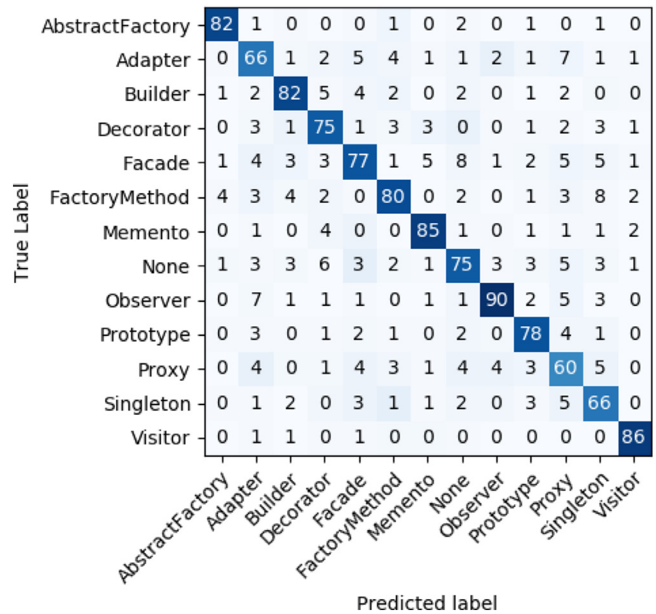

**Fig. 5.** Confusion Matrix of the $DPD_F$ classifier.

*RQ3: Comapring with existing approaches how well does the $DPD_F$ perform?*

It is observed that due to the lack of publicly available standard benchmark, the evaluation and validation of the accuracy of classifiers is difficult. Either existing approaches do not share the corpus or the source code is publicly unavailable to replicate the results.

**Selection Criteria:** Under these limitations, we select two studies from the literature that are relevant to our study. The relevancy measure is that they have either utilised code features or machine learning models to predict design patterns or a combination of both. Though they have not shared their implementation for replicating the results, we select two approaches which are clearly described, which helps us to replicate the results for comparison purposes.

**Benchmark DP detection approaches:** Based on the selection criteria discussed above we compare our study with existing approaches in design pattern detection. The benchmark approaches are developed by Thaller et al. (2019) and Fontana et al. (2011). These studies have utilised some level of code features (metrics) and machine learning classifiers to identify design patterns. Since the source code and the (part of)[9] corpus are not publicly available, we reproduce and replicate the results using the information provided in the existing studies to compare with our study.

**Comparison Strategy:** We apply the following strategy to compare our approach with state-of-the-art.

1. We apply our approach on the corpus provided by the studies and compare results. This corpus is referred to as a *P-MART Corpus*.
2. We apply selected studies to our labelled corpus *$DPD_F$-Corpus* and compare results.

**Labelling the P-MART Corpus:** We selected in total 290 files (including files not containing design patterns i.e none) from the P-MART corpus and labelled them using the same approach we use for labelling the $DPD_F$-Corpus. We refer to this corpus

[9] The DPExample project from the benchmark corpus is not publicly available.

**Table 6**
Comparison of $DPD_F$ with the state-of-the-art. The best results are presented in bold font. Precision (P), Recall (R) and F1-Score (F1-S) values of the benchmark approaches are generated for both benchmark (Labelled P-MART) and our $DFD_F$ corpus. The P-MART corpus contained some of the patterns and not all so the results are tested for the patterns mentioned in the P-MART corpus only.

| Corpora | Design Patterns | FeatureMap | | | MARPLE-DPD | | | $DPD_F$ | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | P (%) | R (%) | F1-S (%) | P (%) | R (%) | F1-S (%) | P (%) | R (%) | F1-S (%) |
| Labelled P-MART | *Abstract Factory* | 48.8 | 52.3 | 50.49 | 73.33 | 71.15 | 72.22 | 78.33 | 78.33 | 78.33 |
| | *Adapter* | 15 | 20 | 17.14 | 78.14 | 75.62 | 76.86 | 91.6 | 86.66 | 89.06 |
| | *Builder* | 55 | 45 | 49.5 | 53.45 | 48.8 | 51.02 | 77.5 | 80 | 78.73 |
| | *Decorator* | 13.22 | 12.8 | 13.01 | 54.18 | 66 | 59.51 | 60 | 36.66 | 45.51 |
| | *Factory Method* | 50.23 | 40.35 | 44.75 | 78.23 | 80.1 | 79.15 | 56.67 | 63.33 | 59.82 |
| | *Observer* | 46.12 | 44.12 | 45.1 | 57.21 | 55.23 | 56.20 | 67.5 | 76.66 | 71.79 |
| | *Singleton* | 63 | 59 | 60.93 | 74.23 | 70.18 | 72.15 | 43.33 | 40.00 | 41.6 |
| | *Visitor* | 30.3 | 35.3 | 32.61 | 45.74 | 50.25 | 47.89 | 96 | 93.3 | 94.63 |
| | *None* | 57.23 | 70.02 | 62.98 | 51.3 | 51.67 | 51.48 | 78.5 | 82.64 | 80.52 |
| | *Overall* | 42.1 | 42.1 | 41.83 | 62.87 | 63.22 | 63.04 | 72.16 | 70.84 | 71.49 |
| $DPD_F$-Corpus | *Abstract Factory* | 55.5 | 49.5 | 52.33 | 75.5 | 77 | 76.24 | 93.27 | 92.08 | 92.67 |
| | *Adapter* | 35 | 31.5 | 33.16 | 85.16 | 78.25 | 81.56 | 71.56 | 66.55 | 68.96 |
| | *Builder* | 62.2 | 60.1 | 61.13 | 58.52 | 51.23 | 54.63 | 83.21 | 83.66 | 83.43 |
| | *Decorator* | 21.28 | 24.5 | 22.78 | 60.15 | 58.23 | 59.17 | 80.99 | 75 | 77.88 |
| | *Factory Method* | 61.3 | 50.45 | 55.35 | 82.15 | 80.8 | 81.47 | 73.58 | 81.88 | 77.51 |
| | *Observer* | 50.1 | 47.65 | 48.84 | 53.25 | 48.26 | 50.63 | 81.26 | 90 | 85.41 |
| | *Singleton* | 65 | 67 | 65.98 | 74.24 | 69.23 | 71.65 | 81.6 | 68.22 | 74.31 |
| | *Visitor* | 55.1 | 80.1 | 65.29 | 60.1 | 66.25 | 63.03 | 97.07 | 91.88 | 93.93 |
| | *None* | 65.25 | 79 | 71.47 | 51.3 | 56.24 | 53.67 | 70.44 | 75 | 72.65 |
| | *Overall* | 52.30 | 54.42 | 52.93 | 66.71 | 65.05 | 65.87 | 81.44 | 80.47 | 80.75 |

**Table 7**
Labelled instances of the benchmark corpus trained by the benchmark classifiers. The red coloured instances are not identified by the $DPD_F$ classifier.

| Patterns | # | Patterns | # |
|---|---|---|---|
| Abstract Factory | 30 | Observer | 30 |
| Adapter | 30 | Prototype | 26 |
| Builder | 30 | Proxy | 0 |
| Decorator | 23 | Singleton | 12 |
| Factory Method | 30 | Visitor | 30 |
| Façade | 9 | None | 30 |
| Memento | 10 | | |

as *Labelled P-MART Corpus*. Table 7 lists the instances of design patterns used in the labelled P-MART corpus. The red coloured patterns are not identified by the classifier.

*Discussion*

Here we compare our study and results with the state-of-the-art studies.

*FeatureMaps:* Thaller et al. (2019) apply feature-maps as an input to random forest and convolutional neural networks (LeCun et al., 1998) to determine whether a given set of classes implement a particular pattern role. Their features are high-level conceptual classes such as micro-structures. As $DPD_F$ classifier's base estimator is Random Forest (RF), we only replicated and compared results with their Random Forest classifier. Though they have only identified *Decorator* and *Singleton* patterns, we replicate their study with all patterns labelled in the benchmark corpus i.e. creating feature maps for all patterns in the benchmark corpus. As shown in Table 6 our $DPD_F$ classifier outperforms the RF classifier in FeatureMaps in terms of Precision and Recall with approximately 30% improvements on benchmark corpus. With the $DPD_F$ corpus our study prevailed over theirs with approximately *30%* improvement in terms of Precision and *26%* in terms of Recall respectively.

*MARPLE-DPD:* In this study, Fontana et al. (2011) have applied basic elements and metrics to mechanically extract design patterns from the source code and further extended it in Zanoni et al. (2015). These approaches use different classifiers for the identification of design patterns, thus, we replicate their study only with the RF classifier and compare results with our $DPD_F$ classifier. We calculate the true and false positives for each pattern and calculate the weighted Precision and Recall for both classifiers. Results in Table 6 show that overall, our approach works far better than MARPLE-DPD with approximately *10%* improvements in terms of Precision on benchmark corpus and *15%* on $DPD_F$ corpus respectively.

To ensure fair predictions of labelled instances of design patterns by the classifiers, we make sure that the labelled benchmark corpus should have at least 30 instances for each pattern, which resulted in (total) seven patterns identified in the end from the benchmark corpus. The Façade, Memento and Prototype are missed because the labelled instances (after removing duplicates) are 10 (as K=10 in cross validation) or less and cannot be cross validated while training the classifiers. In the P-MART dataset many instances are labelled as decorators and prototypes so we treat them as decorator in the final labelled benchmark corpus. That is why there is no instance of Prototype selected for the labelled dataset. Both MARPLE and Featuremap identified singleton better than $DPD_F$ on the labelled P-MART Corpus. It is because very few instances of singleton are found in the corpus and thus not fully trained by the $DPD_F$ classifier. We have also generated results by setting the threshold value to 20 and the results were slightly poorer. From this we infer that the larger the dataset is, better the classifier will be trained and results with higher accuracy will be achieved.

In summary, FeatureMaps works poorly on both corpora whereas, MARPLE-DPD identifies most of the patterns reasonably well with over 62% Precision on benchmark corpus and approximately 67% on $DPD_F$ corpus. Nevertheless, it still lags behind our approach by approximately 10% on benchmark corpus and approximately 15% on our corpus in terms of Precision.

## 5. Threats to validity

In the following section, we outline the relevant threats to the validity of our work.

### 5.1. Threats to internal validity

**Counting of instances:** The first internal threat to the validity of our study is the minor difference in the counting methodologies used between our and benchmark studies. The benchmark studies counted patterns using pattern roles whereas we counted the instances of pattern contained in the file. In addition the DPExample project was not publicly available, which may slightly change the results. We alleviate this problem by labelling the benchmark corpus using our method and compare results.

**Bugs:** It is possible that there are bugs in the processing code that affect the SSLR file generation and the training data, as well as bugs in the *Word2Vec* model and the classifiers we have used, though this is less likely to be a problem given the wide use these tools have already received. We have spent some time debugging our code, found and fixed minor issues, which tend to improve the classification accuracy, so it is more likely than not that bugs will lead to lower precision and/or Recall than would otherwise be the case.

**Data Labelling:** Though we have labelled data using an online tool, there is a possibility of disagreement between the labellers, which may lead to incorrect labelling of the data. To mitigate these issues, we hire at least three labellers to label the corpus as discussed in Section 3.2.1. This process has substantially reduced the disagreement among raters as shown by the high kappa score. To further reduce the disagreement, we intend to hire more labellers in future.

### 5.2. Threats to external validity

The reference set may not reflect the totality of Java source code that may be of interest to developers, and it is unknown whether the classifier will work effectively on the source code of interest outside the reference set. Obviously, the best way to mitigate this risk is to further increase the size and diversity of the reference set, which we plan to address in future.

## 6. Related work

During the past years, with the growing amount of electronically available information, there is substantial interest and a substantial body of work among software engineers and academic researchers in design pattern detection. A majority of the approaches to the detection of design patterns transform the source code and design patterns into some intermediate representations such as rules, models, graphs, productions and languages (Yu et al., 2018). For example, Bernardi et al. (2013) exploited a meta-model which contains a set of properties characterising the structures and behaviours of the source code and design patterns and a matching algorithm is performed to identify the implemented patterns. Alnusair et al. (2014) employed semantic rules to capture the structures and behaviours of the design patterns, based on which the hidden design patterns in open-source libraries are discovered. Recently, Xiong and Li (2019) applied ontology based parser with idiomatic phrases to identify design patterns achieving high Precision.

A good number of studies developed tools that used source code or its intermediate representation to identify patterns as well as machine learning models to predict patterns. Lucia et al. (2011), Tosi et al. (2009) and Zhang and Liu (2013) used static and dynamic analysis (or a combination of both) to develop Eclipse plugins that detect design patterns from source code. Moreno and Marcus (2012) developed a tool *JStereoType* used for detecting low-level patterns (classes, interfaces etc.) to find the design intent of the source code. Zanoni et al. (2015) exploited a combination of graph matching and machine learning techniques to implement a tool called MARPLE-DPD. Niere et al. (2002) designed the FUJABA Tool Suite which provides developers with support for detecting design patterns (including their variants) and smells. (Hautamäki, 2005) used a pattern based solution and tool to teach software developers how to use development solutions in a project.

Some techniques (including tool generation) applied reverse engineering to identify design patterns from source code and UML artefacts. For instance, Lucia et al. (2011) built a tool using static analysis and applied reverse engineering through visual parsing of diagrams. Other studies such as Thongrak and Vatanawood (2014) and Panich and Vatanawood (2016), Shi and Olsson (2006) also applied reverse engineering techniques on UML class and sequence diagrams to extract design patterns using ontology. Brown (1996) proposed a method for automatic detection of design patterns by reverse-engineering the SmallTalk code.

Very few studies utilised code metrics in identification of design patterns. Uchiyama et al. (2011) have presented a software pattern detection approach by using software metrics and machine learning techniques. They have identified candidates for the roles that compose the design patterns by considering machine learning and software metrics. Lanza and Marinescu (2007) approach uses learning from the information extracted from design pattern instances which normally include variant implementations such as number of accessor methods etc. (Fontana et al., 2011) introduced the micro-structures that are regarded as the building blocks of design patterns. Thaller et al. (2019) has built a feature map for pattern instances using neural networks.

Several other approaches exploit machine learning to solve the issue of variants. For example, Chihada et al. (2015) mapped the design pattern detection problem into a learning problem. Their proposed detector is based on learning from the information extracted from design pattern instances, which normally include variant implementations. Ferenc et al. (2005) applied machine learning algorithms to filter false positives out of the results of a graph matching phase, thus providing better precision in the overall output while considering variants. A recent work by Hussain et al. (2018) leverage deep learning algorithms for the organisation and selection of DPs based on text categorisation. To reduce the size of training examples for DP detection, a clustering algorithm is proposed by Dong et al. (2008) based on decision tree learning.

Though we have used machine learning based classifier to test the efficacy of our approach, our approach is substantially different from the aforementioned studies as well as the benchmark studies in many ways. We have summarised the differences with benchmark studies as under:

- We have identified and employed 15 source code features whereas as the benchmark studies of Zanoni et al. (2015) utilised code metrics and Thaller et al. (2019) used feature maps.
- Our labelled corpus size is slightly larger than the benchmark coprus having 1300 files extracted from more than 200 projects from the GJC. Zanoni et al. (2015) and Thaller et al. (2019) used a P-MART corpus that has 1039 files containing the design patterns we intend to identify in a highly imbalanced nature of the P-MART corpus.
- Our machine learning classifier achieved approximately 80% precision whereas existing studies achieved 42% and 63% Precision on Labelled P-MART corpus and 52% and 67% on the $DPD_F$ corpus respectively.
- Our classifier identified tweleve design patterns successfully,
  whereas existing studies only recognised two and six design patterns respectively.

## 7. Conclusion & future work

In this paper, we introduce $DPD_F$, a novel approach for detecting software design patterns by using source code features and machine learning methods. $DPD_F$ builds an SSLR representation by applying call graph and source code features on the $DPD_F$ corpus extracted from the publicly available *'The Java Github Corpus'*. Next, $DPD_F$ constructs a Java n-gram model by applying the *Word2Vec* algorithm on the SSLR file. Finally, $DPD_F$ trains a supervised machine learning classifier on the labelled $DPD_F$ corpus to detect design patterns.

To statistically evaluate the efficacy of the proposed approach we apply three commonly used statistical measures namely Precision, Recall and F1-Score, and additionally build a confusion matrix. Empirical results show that our proposed approach $DFD_F$ detects software design patterns with approximately 80% on Precision and 79% on Recall. $DPD_F$ outperforms two existing approaches, by 30% and 15% respectively in terms of Precision, as well as is able to detect more patterns (12) compared to existing studies.

In future, we plan to investigate whether other useful information for software maintainers can be extracted using our code features or by using more concrete and specific code features. While our classifier's performance is promising, further improvements are clearly highly desirable for extensive practical use. While increasing the size of the training set will almost certainly improve accuracy, it is likely that adding additional code features to the SSLR files would also help, and could be targeted to improve performance on those patterns where accuracy is relatively low.

It is also important to consider how a design pattern recognition system could be integrated into the software engineering process. Scenarios for both the batch processing of large codebases - for instance, to automatically insert pattern information into Javadoc comments, and interactive use from within an IDE, are plausible. An interactive system would offer the chance to incrementally train the detector, thus improving accuracy, and could be integrated with help systems recommending best practices for classes implementing the identified design pattern, or even, conceivably, tools recommending specific refactorings for the class currently open in the IDE.

## CRediT authorship contribution statement

**Najam Nazar:** Conceptualisation, Investigation, Data curation, Writing – original draft, Revision, Original code writer, Preparation, Visualisation, Supervision. **Aldeida Aleti:** Writing – review & editing, Supervision. **Yaokun Zheng:** Data curation, Correction, Code editing and bug fixes.

## Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## Acknowledgements

We would like to extend our gratitude to the individuals who have dedicated their time and effort in performing annotation of design patterns through the online tool. We would also like to thank all anonymous reviewers for their valuable comments. This research received no internal or external funding from any public or private agency.

## Appendix. Reproducibility

For the purpose of reproducibility of our results, we have released our complete implementation with the annotated reference set and the result files to the public as an open-source project via our online appendix at https://github.com/najamnazar/designpatterndetection.

## References

Allamanis, M., Sutton, C., 2013. Mining source code repositories at massive scale using language modeling. In: The 10th Working Conference on Mining Software Repositories. IEEE, pp. 207–216.

Alnusair, A., Zhao, T., Yan, G., 2014. Rule-based detection of design patterns in program code. Int. J. Softw. Tools Technol. Transf. 16, 315–334.

Baroni, M., Dinu, G., Kruszewski, G., 2014. Don't count, predict! A systematic comparison of context-counting vs. context-predicting semantic vectors, In: Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics, ACL, pp. 238–247.

Bernardi, M.L., Cimitile, M., Di Lucca, G.A., 2013. A model-driven graph-matching approach for design pattern detection. In: 2013 20th Working Conference on Reverse Engineering, WCRE, pp. 172–181, ..

Bishop, C.M., 2006. Pattern Recognition and Machine Learning. springer.

Brown, K., 1996. Design Reverse-Engineering and Automated Design-Pattern Detection in Smalltalk. Technical Report, North Carolina State University. Dept. of Computer Science.

Carletta, J., 1996. Assessing agreement on classification tasks: the kappa statistic. Comput. Linguist. 22, 249–254.

Chihada, A., Jalili, S., Hasheminejad, S.M.H., Zangooei, M.H., 2015. Source code and design conformance, design pattern detection from source code by classification approach. Appl. Soft Comput. 26, 357–367.

Christopoulou, A., Giakoumakis, E.A., Zafeiris, V.E., Soukara, V., 2012. Automated refactoring to the strategy design pattern. Inf. Softw. Technol. 54, 1202–1214.

Ciobanu, A.M., Dinu, A., 2013. Alternative measures of word relatedness in distributional semantics. In: Joint Symposium on Semantic Processing. pp. 80.

Cohen, J., 1960. A coefficient of agreement for nominal scales. Educ. Psychol. Measur. 20, 37–46.

Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., Kuksa, P., 2011. NLP (Almost) from scratch. J. Mach. Learn. Res. 12, 2493–2537.

Cortes, C., Vapnik, V., 1995. Support-vector networks. Mach. Learn 20, 273–297.

Detten, M.V., Meyer, M., Travkin, D., 2010. Reverse engineering with the reclipse tool suite. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2, pp. 299–300.

Dong, J., Sun, Y., Zhao, Y., 2008. Compound record clustering algorithm for design pattern detection by decision tree learning. In: 2008 IEEE International Conference on Information Reuse and Integration. IEEE, pp. 226–231.

dos Santos, C.N., Zadrozny, B., 2014. Learning character-level representations for part-of-speech tagging. In: Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 2014, pp. 1818–1826.

Ferenc, R., Beszedes, A., Fulop, L., Lele, J., 2005. Design pattern mining enhanced by machine learning. In: 21st IEEE International Conference on Software Maintenance, ICSM, Vol. 05. IEEE, pp. 295–304.

Fontana, F.A., Maggioni, S., Raibulet, C., 2011. Understanding the relevance of micro-structures for design patterns detection. J. Syst. Softw. 84, 2334–2347.

Gaitani, M.A.G., Zafeiris, V.E., Diamantidis, N., Giakoumakis, E.A., 2015. Automated refactoring to the null object design pattern. Inf. Softw. Technol. 59, 33–52.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Longman Publishing Co. Inc., Boston, MA, USA.

Harris, Z.S., 1954. Distributional structure. Word 10, 146–162.

Hastie, T., Rosset, S., Zhu, J., Zou, H., 2009. Multi-class adaboost. Stat. Interface 2, 349–360.

Hautamäki, J., 2005. Pattern-based tool support for frameworks: Towards architecture-oriented software development environment.

Ho, T.K., 1995. Random decision forests. In: Proceedings of the Third International Conference on Document Analysis and Recognition (Volume 1) - Vol. 1, pp. 278–.

Hu, X., Li, G., Xia, X., Lo, D., Jin, Z., 2018. Deep code comment generation. In: 2018 IEEE/ACM 26th International Conference on Program Comprehension. ICPC, IEEE, pp. 200–20010.

Hu, X., Li, G., Xia, X., Lo, D., Jin, Z., 2020. Deep code comment generation with hybrid lexical and syntactical information. Empir. Softw. Eng. 25, 2179–2217.

Hussain, S., Keung, J., Khan, A.A., Ahmad, A., Cuomo, S., Piccialli, F., Jeon, G., Akhunzada, A., 2018. Implications of deep learning for the automation of design patterns organization. J. Parallel Distrib. Comput. 117, 256–266.

James, G., Witten, D., Hastie, T., Tibshirani, R., 2013. An Introduction To Statistical Learning, Vol. 112. Springer.

Kuchana, P., 2004. Software Architecture Design Patterns in Java. CRC Press.

Kuhn, M., Johnson, K., 2013. Applied Predictive Modeling. Vol. 26. Springer.

Landauer, T.K., Dumais, S.T., 1997. A solution to Plato's problem: The latent semantic analysis theory of acquisition, induction, and representation of knowledge. Psychol. Rev. 104, 211–240. http://dx.doi.org/10.1037/0033-295X.104.2.211.

Lanza, M., Marinescu, R., 2007. Object-Oriented Metrics in Practice: Using Software Metrics To Characterize, Evaluate, and Improve the Design of Object-Oriented Systems. Springer Science & Business Media.
Le, Q.V., Mikolov, T., 2014. Distributed representations of sentences and documents. In: Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 2014, pp. 1188–1196.
LeCun, Y., Bottou, L., Bengio, Y., Haffner, P., et al., 1998. Gradient-based learning applied to document recognition. Proc. IEEE 86, 2278–2324.
Lucia, A.D., Deufemia, V., Gravino, C., Risi, M., Tortora, G., 2011. An eclipse plug-in for the identification of design pattern variants. In: Sixth Workshop of the Italian Eclipse Community (Eclipse-IT 2011), Milan, Italy, pp. 40–51.
Manning, C.D., Raghavan, P., Schütze, H., 2008. Introduction To Information Retrieval. Cambridge University Press, USA.
Mayvan, B.B., Rasoolzadegan, A., 2017. Design pattern detection based on the graph theory. Knowl.-Based Syst. 120, 211–225.
McBurney, P.W., Jiang, S., Kessentini, M., Kraft, N.A., Armaly, A., Mkaouer, M.W., McMillan, C., 2018. Towards prioritizing documentation effort. IEEE Trans. Softw. Eng. 44, 897–913.
McBurney, P.W., McMillan, C., 2015. Automatic source code summarization of context for java methods. IEEE Trans. Softw. Eng. 42, 103–119.
McBurney, P.W., McMillan, C., 2016. An empirical study of the textual similarity between source code and source code summaries. Empir. Softw. Eng. 21, 17–42.
Mikolov, T., Sutskever, I., Chen, K., Corrado, G.S., Dean, J., 2013. Distributed representations of words and phrases and their compositionality. In: Advances in neural information processing systems, pp. 3111–3119.
Mitchell, T., 1997. Machine Learning, McGraw-Hill International Editions McGraw-Hill.
Mitchell, J., Lapata, M., 2008. Vector-based models of semantic composition. In: Proceedings of ACL, Vol. 08. HLT, pp. 236–244.
Moreno, L., Marcus, A., 2012. Jstereocode: automatically identifying method and class stereotypes in java code. In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, pp. 358–361.
Moreno, L., Marcus, A., Pollock, L., Vijay-Shanker, K., 2013. Jsummarizer: An automatic generator of natural language summaries for java classes. In: 2013 21st International Conference on Program Comprehension. ICPC, IEEE, pp. 230–232.
Musco, V., Monperrus, M., Preux, P., 2017. A large-scale study of call graph-based impact prediction using mutation testing. Softw. Qual. J. 25, 921–950.
Nazar, N., Jiang, H., Gao, G., Zhang, T., Xiaochen, X.L., Ren, Z., 2016. Source code fragment summarization with small-scale crowdsourcing based features. Front. Comput. Sci. 10, 504–517.
Niere, J., Schäfer, J.P., Wendehals, L., Welsh, J., 2002. Towards pattern-based design recovery. In: Proceedings of the 24th International Conference on Software Engineering. ACM, pp. 338–348.
Panich, A., Vatanawood, W., 2016. Detection of design patterns from class diagram and sequence diagrams using ontology. In: Computer and Information Science (ICIS), 2016 IEEE/ACIS 15th International Conference on. IEEE, pp. 1–6.
Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E., 2011. Scikit-learn: Machine learning in python. J. Mach. Learn. Res. 12, 2825–2830.
Prechelt, L., Unger-Lamprecht, B., Philippsen, M., Tichy, W.F., 2002. Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance. IEEE Trans. Softw. Eng. 28, 595–606.
Radinsky, K., Agichtein, E., Gabrilovich, E., Markovitch, S., 2011. A word at a time: computing word relatedness using temporal semantic analysis. In: Proceedings of the 20th International Conference on World Wide Web, WWW 2011, Hyderabad, India, March 28 - April 1, 2011, pp. 337–346.
Řehůřek, R., Sojka, P., 2010. Software framework for topic modelling with large corpora. In: Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks, pp. 45–50.
Reisinger, J., Mooney, R.J., 2010. Multi-prototype vector-space models of word meaning. In: Human Language Technologies: Conference of the North American Chapter of the Association of Computational Linguistics, Proceedings, June (2010) 2-4, Los Angeles, California, USA, pp. 109–117.
Salton, G., Wong, A., Yang, C.S., 1975. A vector space model for automatic indexing. Commun. ACM 18, 613–620.
Scanniello, G., Gravino, C., Risi, M., Tortora, G., Dodero, G., 2015. Documenting design-pattern instances: A family of experiments on source-code comprehensibility. ACM Trans. Softw. Eng. Methodol. (TOSEM) 24, 1–35.
Shi, N., Olsson, R.A., 2006. Reverse engineering of design patterns from java source code. In: Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on. IEEE, pp. 123–134.
Thaller, H., Linsbauer, L., Egyed, A., 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, pp. 207–217.
Thongrak, M., Vatanawood, W., 2014. Detection of design pattern in class diagram using ontology. In: Computer Science and Engineering Conference (ICSEC), 2014 International. IEEE, pp. 97–102.
Tosi, C., Zanoni, M., Maggioni, S., 2009. A design pattern detection plugin for eclipse. Eclipse-IT 2009 (89).
Uchiyama, S., Washizaki, H., Fukazawa, Y., Kubo, A., 2011. Design pattern detection using software metrics and machine learning. In: First international workshop on model-driven software migration, MDSM 2011, pp. 38.
Walter, B., Alkhaeir, T., 2016. The relationship between design patterns and code smells: An exploratory study. Inf. Softw. Technol. 74, 127–142.
Xiong, R., Li, B., 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, pp. 163–174.
Yu, D., Zhang, P., Yang, J., Chen, Z., Liu, C., Chen, J., 2018. Efficiently detecting structural design pattern instances based on ordered sequences. J. Syst. Softw. 142, 35–56.
Zanoni, M., Fontana, F.A., Stella, F., 2015. On applying machine learning techniques for design pattern detection. J. Syst. Softw. 103, 102–117.
Zhang, H., Liu, S., 2013. Java source code static check eclipse plug-in based on common design pattern. In: Software Engineering (WCSE), 2013 Fourth World Congress on. IEEE, pp. 165–170.

**Dr. Najam Nazar** is a sessional lecturer and teaching associate at the Faculty of Information Technology, Monash University in Australia. His research interests include mining software repositories (MSR), software testing, software product lines and education in software engineering.

**Dr. Aldeida Aleti** is an Associate Professor at the Faculty of Information Technology, Monash University in Australia. Aldeida's research is in the area of search-based software engineering (SBSE), with a particular focus on what makes software engineering problems (design, testing, program repair) hard to optimise and designing approaches that make it easier to apply SBSE techniques to new problems. Aldeida has served as PC member and organising committee at both SE and optimisation conferences, such as ASE, ICSE, GECCO, ESEC-FSE, and SSBSE, and is in the editorial board of JSS.

**M.Sc. Yaokun Zheng** is a teaching associate at the Monash university Australia. His research interest includes Software Engineering, Machine Learning and Natural Language Processing.