Features and Supervised Machine Learning-Based Method for Singleton Design Pattern Variants Detection

Abir Nacef
Laboratoire d'Informatique pour les Systèmes Industriels, Tunis El Manar University
Sahbi Bahroun (Sahbi.bahroun@isi.utm.tn)
Institut Supérieur d'Informatique (ISI), Laboratoire Limtic, Tunis El Manar University
Adel Khalfallah
Laboratoire d'Informatique pour les Systèmes Industriels, Tunis El Manar University
Samir Ben Ahmed
Laboratoire d'Informatique pour les Systèmes Industriels, Tunis El Manar University

Research Article

Keywords: Design patterns, Singleton variant, RNN-LSTM Classifier, Machine Learning, SD Classifier

Posted Date: December 19th, 2022

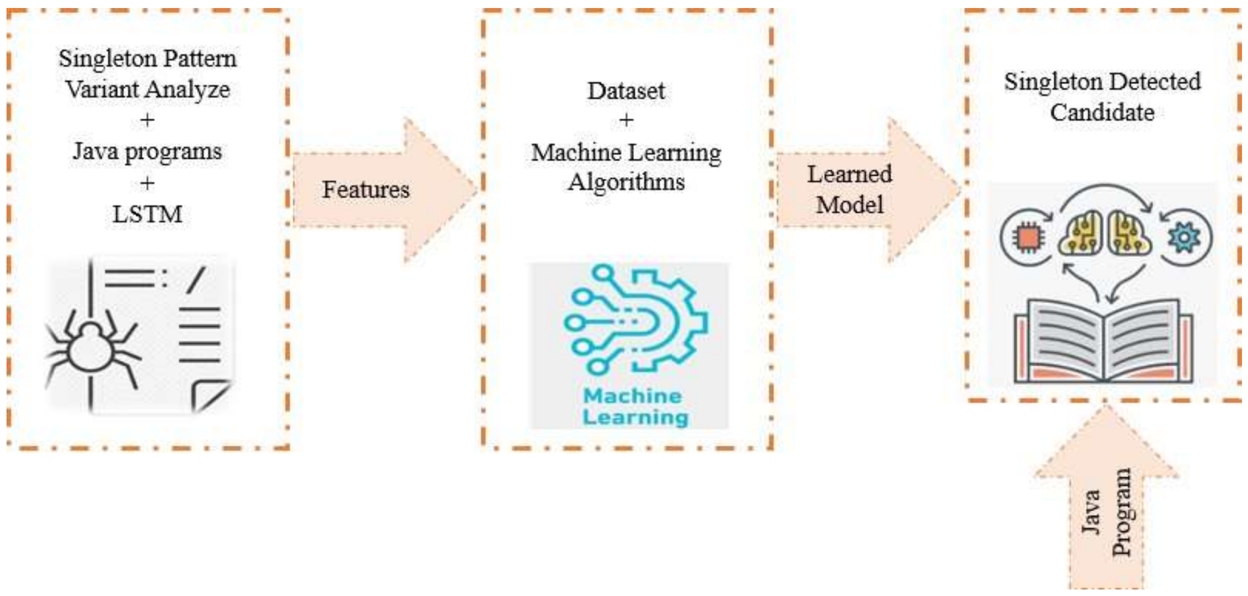
DOI: https://doi.org/10.21203/rs.3.rs-2374835/v1

Additional Declarations: No competing interests reported.

Graphical Abstract

Features and Supervised Machine Learning-Based Method for Singleton Design Pattern Variants Detection

Abir Nacef, Sahbi Bahroun, Adel Khalfallah, Samir Ben Ahmed,



Highlights

Features and Supervised Machine Learning-Based Method for Singleton Design Pattern Variants Detection

Abir Nacef, Sahbi Bahroun, Adel Khalfallah, Samir Ben Ahmed,

- Structural and semantic analysis can effectively capture the pattern intent.
- The use of features improves the accuracy of the recovered model
- Using ML based on specifically created datasets performs better results in DPs detection.

Features and Supervised Machine Learning-Based Method for Singleton Design Pattern Variants Detection

Abir Nacef[1], Sahbi Bahroun[2], Adel Khalfallah[1], Samir Ben Ahmed[1],
[1]Faculty of Mathematical, Physical and Natural Sciences of Tunis (FST), Laboratoire d'Informatique pour les Systèmes Industriels, Tunis El Manar University, TUNISIA
[2]Institut Supérieur d'Informatique (ISI), Laboratoire Limtic, Tunis El Manar University, TUNISIA
Email addresses: nacefabir91@gmail.com (Abir Nacef), Sahbi.bahroun@isi.utm.tn (Sahbi Bahroun), Adel.khalfallah@isi.utm.tn (Adel Khalfallah), samir.benahmed@fst.utm.tn (Samir Ben Ahmed)

Abstract

Design patterns codify standard solutions to common problems in software design and architecture. Given their importance in improving software quality and facilitating code reuse, many types of research are proposed on their automatic detection. In this paper, we focus on singleton pattern recovery by proposing a method that can identify orthodox implementations and non-standard variants. The recovery process is based on specific data created using a set of relevant features. These features are specific information defining each variant which is extracted from the java program by structural and semantic analysis. We are based on the singleton analysis and different proposed features presented by Nacef et al. in [1] to create structured data. This data contains a combination of feature values defining each singleton variant to train a supervised Machine Learning (ML) algorithm. The goal is not limited to detecting the singleton pattern but also the specification of the implemented variant as so as the incorrect structure that is incoherent with the pattern intent. We use different ML algorithms to create the Singleton Detector (**SD**) to compare their performance. The empirical results demonstrate that our method for automatically training the (**SD**) based on features and supervised ML, can identify any singleton implementation with the specific variant's name achieving 99% of precision, and recall. We have compared the proposed approach to similar studies namely **DPDf** and **GEML**. The results show that the **SD** outperforms the state-of-the-art approaches by more than 20% on evaluated data constructed from different repositories; **PMART**, **DPB** and **DPDf** corpus in terms of precision.

Keywords:
Design patterns, Singleton variant, RNN-LSTM Classifier, Machine Learning, SD Classifier

1. Introduction

Design patterns [2] have an important role in the software development process. The term has become commonplace among software designers, requirements engineers, and software programmers alike. The language of design patterns is now necessary to understand and work with software because it helps to understand the design intent of pre-developed software. Hence, software reverses engineering and redesign [3], [4], [5]. His recovery can make the maintenance of source code, more easily and enhance her existing analysis tools by bringing program understanding to the design level. Regarding its important role in improving program comprehension and re-engineering, design pattern detection became a more and more active research field and has observed in recent years, a continual improvement in the field of automatic detection. However, there are many difficulties in detecting practical design patterns that arise from the following:

- The non-formalization of the pattern, the variety in implementations of source codes, and the increasing complexity of software projects

Email addresses: nacefabir91@gmail.com (Abir Nacef), Sahbi.bahroun@isi.utm.tn (Sahbi Bahroun), Adel.khalfallah@isi.utm.tn (Adel Khalfallah), samir.benahmed@fst.utm.tn (Samir Ben Ahmed)
[1]Faculty of Mathematical, Physical and Natural Sciences of Tunis (FST), Laboratoire d'Informatique pour les Systèmes Industriels, Tunis El Manar University, TUNISIA
[2]Institut Supérieur d'Informatique (ISI), Laboratoire Limtic, Tunis El Manar University, TUNISIA

- The design structure does not match the intent: Even though the identified pattern instances correspond to their structure, the design intent can be broken. Which can be considered the main reason behind the increase in the false positive rate.

Most existing methods convert source code and design patterns into intermediate representations, such as rules, models, graphs, products, and languages. Using these intermediate representations makes it easier to extract structural elements such as classes, properties, methods, etc. from the source code. However, they show poor performance compared to methods based on extracted features. At the same time, structural analysis cannot recover the pattern intent of its different representations, so it is necessary to perform semantic analysis on the source code to improve the accuracy of the recovered model. On the other hand, extracting semantic information from source code is still a difficult task due to the complexity and diverse representation of the code. Therefore, to solve this problem, we are based on the work proposed by Nacef et al. in [1] to define singleton variants rules and create specific data to recover each variant from the source code. Nacef et al. [1] have proposed a set of features for identifying singleton patterns based on a detailed analysis of the variant's structure and behaviors. For analyzing the java program Nacef et al. apply structural and semantic analysis by the use of the **LSTM** model to extract semantic information (features). The **LSTM** model is trained by specifically structured data corresponding to each feature for a classification task. Since the source code is a special structured natural language, written by programmers [6], the created data in [1] contains snippets of code treated as plain text. Nacef et al. exploit the efficiency of **LSTM** models in Computer Vision and Natural Language Processing (**NLP**) [7] evidenced by many recent works, to recover semantic information and elements dependencies from java programs without using any intermediate representation.

The main goal of our work is the detection of the singleton design pattern. The **SD** role has not been limited only to the recovery of singleton variants (including non-standard variants and their combinations), but also incorrect implementations inhibit the singleton intent. We would like to be able to verify that the singleton pattern has been implemented correctly. While designers understand patterns well, developers may not have as much experience. This can lead to incorrect implementation of the pattern or the possibility of later introducing coding errors that break the pattern stage. The recovery of incorrect implementation makes possible the refactoring task. If a singleton pattern is detected, the **SD** can specify the type of the implemented variant.

In this paper, we propose a Feature-Based Singleton Design Pattern Detection approach that uses a set of features extracted from both structural and semantic analyses [1]. We create specific structured data based on the feature's combination values corresponding to each singleton variant to train a supervised ML model named **SD**. Trying to construct data containing various implementations, we ameliorate the learning process of the **SD** to recover any implementation of the singleton pattern. The proposed approach is based on the detailed analysis of the pattern realized in [1], [8], and [9] and the features proposed in [1] to extract rules for pattern identification with the goal to create the data. This dataset serves as training data to make learning the **SD** model. Then we evaluate the proposed approach with a labeled dataset collected from **PMART** [10], [11], **DPB** [12], **DPDf-corp,us** [13], and 94 Java files extracted from a publicly available GitHub Java Corpus. The detector makes a very good result, with higher accuracy compared to the state-of-the-art approaches.

The Contribution of the paper can be summarized as follows:

- We introduce a novel approach called "Singleton Design Pattern Variant Detection using features and Supervised Machine Learning" (**SD**) that uses 33 features to recover Singleton pattern variants.
- We create structured data named **DTSD** for training the **SD** classifier. the **DTSD** contains 7000 samples (a combination of feature values).
- We build two labeled data for evaluating the **SD**; the first is extracted from **DPDf** Corpus [14] (we take only files with singleton implementation) named **DPDf 2** and we insert the missing variants. The second is building from the singleton pattern file existing in **P-MART**, **DPB,** and **DPDf** Corpus.
- We proved that our approach outperforms similar existing approaches by a substantial margin in terms of standard measures.

The rest of the paper is organized as follows; we present the related work and the contribution made to them in Section 2. We indicate the reported singleton variants and the highlighted features in Section 3. In section 4, we discuss the relevant background of the related technologies and we present the proposed approach. Section 5 presents obtained results. An evaluation of different **ML** algorithms and a comparison between our study and the state-of-the-art are realized and discussed. Finally, in Section 6 a conclusion and future work is presented.

2. Related Work

In recent years, design pattern detection became a more and more active research domain. The problem of recovering design patterns from the source code has been faced and discussed in several works. Many strategies and many techniques are used.

The majority of design pattern mining approaches transform the source code and design patterns into some intermediate representations such as an abstract semantic graph, abstract syntax tree, rules, grammar, etc. The searching methods diversify also from one to another, and can be classified as metrics, constraint resolvers, database queries, eXtended Positional Grammar (XPG), etc. . . .

Several approaches [15],[16],[17], [18] use database queries as a technique for extracting patterns. They use Structured Query Language (SQL) queries to extract pattern-related information and produce an intermediate representation of the source code. In this case, the performances depend enormously on the underlying database and can be scaled very well. However, queries are limited to the available information existing in the intermediate representations. Techniques proposed by [19], [20] use program-related metrics (e.g., aggregations, generalizations, associations, and interface hierarchies) from different source code representations. The detection of DP is based on comparing DP and code source metrics values. This method is computationally efficient because it reduces the search space through filtration [21]. **FUJABA** [22] tool introduces an explicit DSL to express pattern specification declaratively.
The provided support is used to detect many variants of design patterns. An advanced step proposed by Uchiyama et al., in [23] combine software metrics and machine learning to identify candidates for the roles that compose.

Other techniques are based on graph representation. The detection approach proposed by [24] combines graph and software metrics to perform the recognition process. First, a set of candidate classes for each DP role are identified based on software metrics. These metrics were chosen based on the theoretical description of the DP, and they were used to establish clear logical rules that could lead to many false positives. In the second stage, all candidate class combinations are analyzed in detail to find DP matches. To improve the accuracy of results, machine learning methods (decision trees and artificial neural networks) are used to filter as many as possible false and distinguish similar patterns [25]. Mario di Luca et al. use a DSL-driven graph matching where DPs are modeled based on their high-level structural properties. Another work proposed by Zanoni et al.in [26] develop a **MARPLE** tool [27] that exploited a combination of graph matching and machine learning techniques.

Many works [28] [4] apply reverse engineering techniques on UML class and sequence diagrams through ontology to extract design patterns. Lucia et al., develop a **DPRE** tool [29] based also on reverse engineering through visual parsing of diagrams. Another work proposed by [30] explore the detection of design pattern on UML class diagrams by the use of first-order logic representation.

Influenced by the work given by Tsantalis et al. [31], Thaller et al., [32] propose a feature map for pattern insinstances-based neural networks. Chihada et al., in [33] propose a design pattern detector that learns based on the information extracted from each pattern instance. They treat the design pattern recognition problem as a learning problem. However, recent work proposed by [34] treats the problem as text categorization, in which they leverage deep learning algorithms for organizing and selecting DPs. Recently, Najam Nazar et al. in [13] selected 15 feature codes and use machine learning classifiers to automatically train a design pattern detector. Another recent work proposed in [35] presents a novel machine learning-based approach for DPD named GEML. Like other work, the used method explores the ML capacity, but Barbudo et al. [35] addressed their limitations by using G3P as a basis of the proposed approach.

Alhusain proposes in [36] a DPD approach based only on machine learning methods. The proposed method starts by reducing the search space with the identification of a set of candidate classes for each DP role. Then, candidates with possible role combinations are checked. An ANN is used to validate DP instances. The ML algorithm is trained by different input feature vectors using a feature selection method.

Though we base on machine learning to extract information from source code and to detect Singleton Variant, our approach differs from the other mentioned approaches in many ways. The difference that can exist is summarized as under.

- Stencel et al. in [8] and other work, detect many variants of the single pattern. However, we are the first to detect non-only the different variants, but the possible combination and incoherent implementations that inhibit the pattern intent, with their corresponding names.
- Our approach use ML like many other approaches, but it is the first to create singleton specific dataset for training the model (whether at the level of code analysis or pattern extraction), which performs the model to better training, i.e. better results, and make easy to filter false positive.
- As [34], we use deep learning algorithms for text categorization, but extracting a pattern from direct source code is a very hard task and needs an enormous number of data because there are many non-standard implementations. In our work, we consider text categorization as the first step in which we extract needed information from the source code. This information represents features that describe the pattern structure and behavior. The use of features reduces the search space, and the size of training data increases the prediction rate and decreases the false positive number.
- **DPDf** [13] uses 15 source code features to identify 12 Design Patterns, [27] utilizes code metrics, and [32] uses feature maps. However, we employ 33 features, especially for the singleton pattern. The use of specific features gives a detailed definition of the pattern and allows the identification of non-standard implementations.
- The data dedicated to the SD has a size of 7000. However, the **P-MART** corpus includes 1039 files, and **DPDf** uses a corpus with 1300 files for training the classifier to recover many design patterns with imbalanced nature. The Corpus Used in both is used for training and evaluating the model. The number of singleton patterns existing in **P-MART** and **DPDf** corpus is respectively 12, and 100 which is not entirely satisfactory to recover all implementations.
- Our SD based on extracted features from structural and semantic analysis of source code and **ML** techniques achieved approximately 98% of precision and recall and prove its capability to recover any non-standard variant and filter false positives. However, **DPDf** and **GEML** did not exceed 80% in terms of precision, recall, and F1 Score.

3. Reported variants and proposed features for Singleton design pattern detection

The singleton design pattern is used to ensure that a class has only one instance. That means restricting class instantiation to a single object (or even to a few objects only) in a system and providing a global access point to it. The recognition of instances of singleton patterns in source code is difficult, caused of the different implementations, which are also not formally defined. So that we depart from the singleton pattern analysis and their variants presented in [1].

3.1. Reported Variants

Based on the proposed singleton variants defined in [2], [8], our approach can effectively recognize the different variants represented in table 1 and their combinations form.

Table 1: Reported Singleton variants

Singleton Variants	**Description**
Eager Instantiation	Lazy Instantiation
Placeholder	Replaceable Instance
Subclassed Singleton	Delegated Construction
Different Access Point	Limiton
Social Singleton	Generic Singleton using Reflection

3.2. Features used for the singleton pattern detection

For singleton pattern detection, we have to use the 33 features proposed by [1] resulting from the singleton variants analysis. This specific analysis allowed the extraction of the essential information for each variant identification. These features are presented in table 2.

If we just look at its canonical implementation (which is quite simple), the intent of the singleton pattern seems simple. However, careful analysis of the structure may lead to further constraints being defined. These characteristics reflect information that has forty effects on keeping the singleton intent.

Table2: Used Features

No.	Abbreviation	Feature
1	IRE	Inheritance relationship (extends)
2	IRI	Inheritance relationship (implements)
3	CA	Class accessibility (public, abstract, final)
4	GOD	Global class attribute declaration
5	AA	Class attribute accessibility
6	SR	Static class attribute
7	ON	Have only one class attribute
8	COA	Constructor accessibility
9	HC	Hidden Constructor
10	ILC	Instantiate when loading class
11	GAM	Global accessor method
12	PSI	Public Static accessor method
13	GSM	Global setter method
14	PST	Public static setter method
15	INC	Use of Inner class
16	EC	Use External Class
17	RINIC	Returning instance created by the inner class
18	RINEC	Returning instance created by the External class
19	CS	Control instantiation
20	HGM	Have one method to generate the instance
21	DC	Double-check locking
22	RR	Return reference of the Singleton instance
23	CNI	Use a variable to count the number of instances
24	CII	Create an internal static read-only instance
25	DM	Use delegated method
26	GMS	Global accessor synchronized method
27	IGO	Initializing global class attribute
28	LNI	Limit the number of instances (Verify if there is a condition to instantiate limited by the number of instances (< value))
29	SCI	Use string to create an instance
30	SB	Static Block
31	AFL	Allowed Friend List
32	CAFB	Control access to friend behavior (Checking the registered allowed friend list to respond/access when one friend tries to access the behavior of another friend's social object)
33	UR	Type for generic instantiation (Using Singleton class as a type for generic instantiation)

The only way to keep a singleton instance for future reuse is to store it as a global static variable. With this, we should check the correctness of the different variants and count the number of class attributes and method-generating instances. If the number exceeds one then the structure is incorrect and the intention is inhibited (case of listing 1 and 2 .

The common conditions that must be verified in most singleton variants are:

- The global access point to get the instance.
- The Constructor modifier to restrict its accessibility.
- The Control of instantiation; verifying the existence of conditions that limit the number of created instances.
- The verification of the number of declared class attributes and the number of method-creating instances to only one.

Second, specific information for each variant should be verified. In table 3, we categorize features with corresponding variants. Relevant information needed for the identification of each variant is regrouped. This grouping strongly helps SD data creation.

Table3: Features corresponding to each variant

Singleton Variants	**Features**
Eager Implementation	ILC, SB
Lazy Instantiation	GAM, PSI, CS, HC, DC, GMS
Different Placeholder	INC, RINIC
Replaceable Instance	PST, GSM
Subclassed Singleton	CA, IRE
Different Access Point	EC, RINEC
Delegated Construction	DM
Limiton	LNI
Social Singleton	IRI, AFL, CAFB
Generic Singleton using Reflection	TUR

4. Proposed approach for singleton detection with supervised machine learning

In this section, we will represent the used techniques and discuss the singleton detection process. Finally, we are gone give details about the created data used for the training phase.

4.1. ML used Techniques

Machine learning is a subset of artificial intelligence (**AI**) that focuses on creating systems that can learn from data, identify patterns and make decisions with minimal human intervention. The power of machine learning is its capability to automatically improve performance through experience [37]. **ML** has penetrated every aspect of our lives, and made the hard task easier to resolve, with higher accuracy [38], [39].

Algorithms are the engines of machine learning. In general, two main types of machine learning algorithms are used today: supervised learning and unsupervised learning. The difference between the two is defined by the method used to process the data to make results, the type of input and output data, and the task that they intend to solve.

In our work, we use the supervised learning type, which is supplied with information about several entities whose class membership is known and which produces from this a characterization of each class. In supervised learning, there are two major types named regression and classification. In our work, tasks realized have a classification type. The first step realized by Nacef et al. in [1] analyze the java program by the use of (**RNN-LSTM**) [40], [41] to extract the value of features with binary or multi-class classification. In the second phase, we use the previous analysis of the singleton pattern and feature's value to create labeled structured data for training the **SD** classifier. The **SD** carries out a multi-class classification task; each class represents a singleton variant (if a singleton pattern is implemented) or none otherwise. For the **SD**, we built various **ML** classifiers such as **Random Forest** [42], **Gradient Boosted Tree** [43] , **SVM** [44], **KNN** [45], and **Neural Network** [46] intending to compare their results random forest provides a unique combination of prediction accuracy and model interpretability among the usage of bagging on samples that overcome

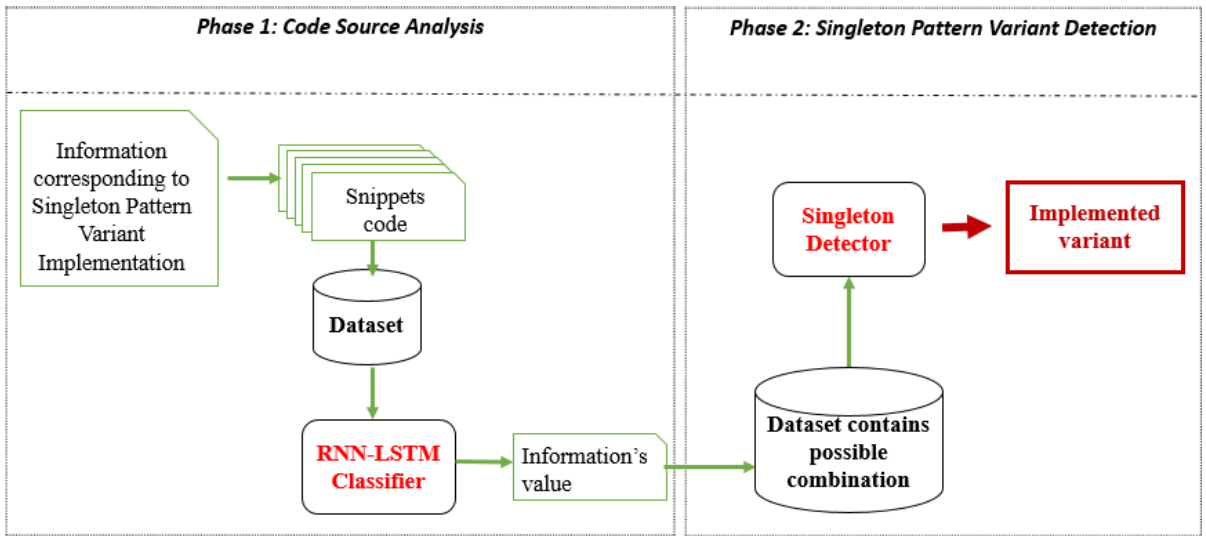

Phase 1: Code Source Analysis
Phase 2: Singleton Pattern Variant Detection
Information corresponding to Singleton Pattern Variant Implementation
Snippets code
Dataset
RNN-LSTM Classifier
Information's value
Dataset contains possible combination
Singleton Detector
Implemented variant

Figure 1: Flowchart of the proposed approach

overfitting issues. The **Gradient Boosted Tree** is generally more accurate compared to other models and trains faster especially on larger datasets. The **SVM** has high predictive accuracy and works well on smaller datasets. The **KNN** is a simple algorithm that is easy to implement and doesn't need to create a model, adjust multiple parameters or make additional assumptions. Finally, the **Neural Network** is good to model nonlinear data with a large number of inputs and has computational power enabling to process of more information. These sets of algorithms are the most used in design pattern detection.

Figure 1 illustrates the proposed approach process. In the following subsections, we will discuss these phases one by one.

4.2. *First Phase*

The detailed analyses of singleton implementation variants make easier the defining of efficient information and the construction of the training dataset according to them. In [1] a detailed analysis of the singleton pattern is realized, a set of variants is identified, and based on their behavior and structure 33 relevant features for their definition are highlighted. For extracting feature values from the java program a syntactical and semantic analysis is applied by the use of **LSTM**. The **LSTM** is a sequential model which can extract information and dependencies existing in the source code [47]. Based on the analysis and the feature's value extracted from the java program we define rules for every singleton variant and then we construct the dataset for training the **SD**.

4.3. *Second Phase*

The analysis already done was very useful in terms of defining the rules for each implementation variant, which will be the key behind the construction of the training data **DTSD**. The strength of our approach is that we use specific features, as well as the use of our own created data.

- **Dataset creation process;** As we know, the data is the essence of ML, then the more large and more diverse the dataset is, the better results will be obtained. If the classifier is trained by completed and diversified examples, it will be more able to predict correct instances, and achieve a higher accuracy result. Based on the important role of the data in building performed model, we gave importance to the creation of the training dataset. Contrarily to other approaches, we don't limit ourselves to the existing implementation extracted from the considerate benchmark corpus, because many implementations can be missed, or we can have an imbalance in their numbers (dominance of some implementations over others) which causes an unfair training process and can lead to the incapability of the model to recover all existing implementations. So for training the model, we create a dataset

DTSD based on rules extracted from singleton variants analyses, and for evaluating the model, we use other created data based on the **DPDf** Singleton corpus.

Table4: SD Classifier results

Singleton Variants	Classes	NO.
Eager Implementation	Simple Implementation	1
	Implementation With static Block	2
	Simple Implementation-Invalid Implementation	3
	Implementation With static Block-Invalid Implementation	4
Lazy Instantiation	Singleton naif	5
	Non-thread safe	6
	Thread safe with the synchronized method	7
	Lazy instantiation double lock mechanism	8
	Singleton naif-Invalid Implementation	9
	Non thread safe-Invalid Implementation	10
	Thread safe with synchronized method-Invalid Implementation	11
	Lazy instantiation double lock mechanism-Invalid Implementation	12
Different Placeholder	Different Placeholder	13
	Different Placeholder-Invalid Implementation	14
Replaceable Instance	Replaceable Instance	15
	Replaceable Instance-Invalid Implementation	16
Subclassed Singleton	Subclassed Singleton	17
	Subclassed Singleton-Invalid Implementation	18
Different Access Point	Different Access Point	19
	Different Access Point -Invalid Implementation	20
Delegated Construction	Delegated Construction	21
	Delegated Construction -Invalid Implementation	22
Limiton	Limiton	23
	Limiton-Invalid Implementation	24
Social Singleton	Social Singleton	25
Generic Singleton using Reflection	Generic Singleton using Reflection	26
No Singleton Implementation	No Singleton Implementation	27

- **Defining rules;** we have identified 27 instances according to different singleton variants. The class candidates are represented in table 4. Based on specific information about each one, we illustrate rules to define them. The rules represent a combination of values of features. The **SD** will be made by learning from information extracted from the **DTSD** data. Table 5 shows an example of important feature values that made each candidate instance true. Noting that, based on these most important features, we can create several implementations by playing on the other features' values (we must respect the identity of each variant).

  To achieve high recall (which will be explained in the next section), we need to reduce the number of false-negative predictions. For a singleton design pattern detection, this leads to creating a dataset that comports numerous implementation variants that preserve the meaning of the pattern, and those that can destroy the intent. As an example, the only way to keep a singleton instance for future reuse is to store it as a global static variable, so we should verify that there is only one declared class attribute in different variants (**ON**). In the case of different placeholder and different access point variants, the instance is held as a static attribute of an inner class or external class, so we should verify the existence of a static class attribute inside of both. Another example, as the intent of a singleton pattern is to limit the number of objects to only one (exception Limiton variant), we must verify that there is only one block for creating an Instance (**HGM**). If the two features (**ON/HGM**) are false, the implementation will be considered an incorrect singleton structure (as shown in table 11). This error inhibits the singleton intent and can provoke false-negative predictions. This type of error can be caused by a developer's unconsciousness during the implementation, so we decided to consider it, and detect similar implementations if they exist. Listings 1 and 2 represent an example of incorrect lazy and eager implementation, which inhibits the singleton pattern intent. Table 5,6,7,8,9,10 presents an example of rules defining some singleton variants and table 11 represents an example of a combination of feature values making the implementation error.

Table 5: Data extract: Example of features combination for Eager Singleton variant

Class	**Combination Values**											
	CA	GOD	AA	SR	COA	ILC	GAM	PSI	RR	SB	ON	HGM
Eager Simple Implementation	public / final	True	Private / private final	True	private	True	True	True	True	False	True	True
Eager Implementation With static Block	Public / final	True	Private /private final	True	private	True	True	True	True	True	True	True

Table 6: Data extract: Example of features combination for Lazy Instantiation Singleton variant

Class	CA	GOD	AA	SR	COA	**Combination Values**			RR	CS	DC	GMS	ON	HGM
						ILC	GAM	PSI						
Lazy instantiation non thread safe	Public / final	True	Private / private final	True	private	False	True	True	True	True	False	False	True	True
Lazy instantiation thread safe with synchronized method	Public /final	True	Private /private final	True	Private	False	True	True	True	True	False	True	True	True

Table 7: Data extract: Example of features combination for Different Placeholder Singleton variant

Class	CA	COA	ILC	**Combination Values**			INC	RINIC	ON	HGM
				GAM	PSI	RR				
Different Placeholder	public / final	Private	False	True	True	True	True	True	True	True

Table 8: Data extract: Example of features combination for Replaceable Instance Singleton variant

Class	**Combination Values**											
					COA	GAM	PSI					
Replaceable Instance	public / final	True	Private /private final	True	Private	True	True	True	True	True	True	True

Table 9: Data extract: Example of features combination for Delegated Construction Singleton variant

Class	CA	GOD	AA	COA	**Combination Values**		PSI	RR	DM	ON	HGM
					CS	GAM					
Delegated Construction	public / final	True	Private /private final	Private	True	True	True	True	True	True	True

Table 10: Data extract: Example of features combination for Social Singleton Singleton variant

Class	CA	**Combination Values**			
		COA	IRI	AFL	CAFB
Social Singleton	Public	Private	True	True	True

Table 11: Data extract: Example of features combination for Eager Invalid Implementation Singleton variant

Class	**Combination Values**											
					COA	ILC	GAM					
Simple Implementation-Invalid Implementation	public / final	True	Private / private final	True	private	True	True	True	True	False	False	False
Implementation With static Block-Invalid Implementation	Public / final	True	Private /private final	True	private	True	True	True	True	True	False	False

Listing 1: Example 1: Singleton Error Implementations

```
Public class C1 {
        Private static C1 instance = new C1 ();
        C1 () {}
        public static C1 getInstance(){
                        Return(new C1 ());}
}
```

Listing 2: Example 2: Singleton Error Implementations

```
Public class C2  {
            Private static C2 instance;
            Private C2 ()    {}
            Public static C2 getInstance1()    {
                              if(instance==null)instance=new C2;
             }
              Public static C2 getInstance2(){return new C2;}
}
```

- **Building Singleton Variants Classifier;** The use of an ML algorithm depends on the type of data to treat and generate, and the type of realized task. A singleton design pattern detection is a classification problem, with structured labeled data. To deal with this typical problem we can use different algorithms. In this case, the better model to use is the one that gives a better result. We choose to use five different algorithms the most already used in classification tasks; **Random Forest**, **Gradient Boosted Tree**, **SVM**, **KNN**, and **Neural Network**.

5. Evaluation setup

This section presents the criteria used to evaluate our **SD** and the different results made by it. We compared results generated from each model, and interpret and compared them with state-of-the-art approaches.

5.1. Evaluation protocol

The standard measures to statistically evaluate the efficacy of classifiers are Precision, Recall, and the F1-Score. Prediction: The prediction rate indicates the fraction of positive prediction which was correct. It is defined in 1:

$$Precision = \frac{TP}{TP + FP} \quad (1)$$

Recall: The recall indicates the fraction of actual positives which was identified correctly. It is defined in 2 :

$$Recall = \frac{TP}{TP + FN} \quad (2)$$

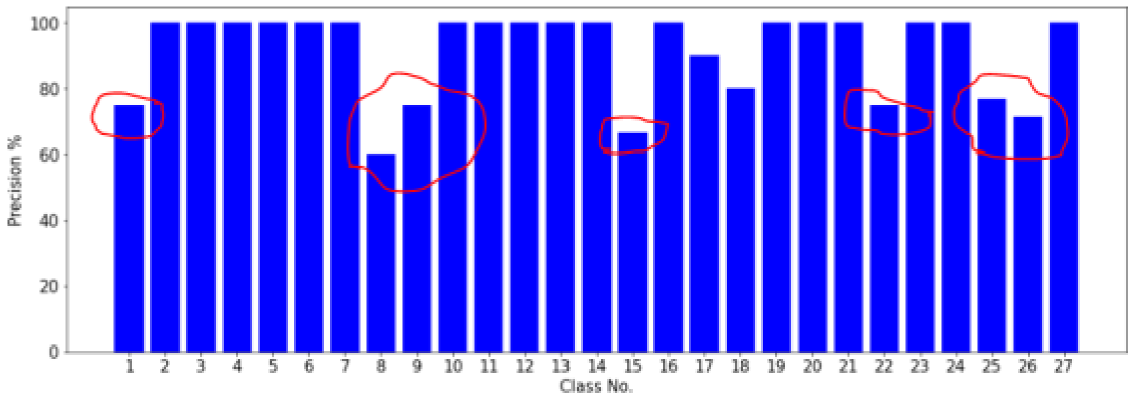


Figure 2: KNN precision results

(TP) : true positive and (TN) true negative represent the number of positive and negative classes which are correctly predicted by the classifier. However, (FP) false-positive and (FN) false-negative represent the number of positive and negative classes which are incorrectly predicted. The higher the precision and recall scores are, the performer Classifier is. However, there is a trade-off between the two values: when tuning a classifier, improves the precision score, typically reduces recall score, and vice versa, there is no free lunch. That's where F1-score are needed measure to use.

The F1 score is a way to combine both precisions and recall into a single number. It represents a harmonic mean of both scores, which is given by this simple formula 3:

$$F1Score = \frac{2 * (precision * recall)}{(precision + recall)} \tag{3}$$

5.2. Performed results

We have created a classifier based on different **ML** models. The use of specific training data **DTSD** makes the classifier more able to recognize easily every candidate instance with high performance. After training the model we have tested it with the 200 GitHub java classes referred by **DPDf** Corpus proposed by [13] 100 files contain Singleton Implementation and 100 files do not contain any type of design patterns i.e. none.

Table 12: SD Classifier results

SD Classifiers	**Precision (%)** **Measures**		
	Precision	**Recall (%)**	**F1-score (%)**
Random Forest	96.24	96.47	95.81
Gradient Boosted Tree	98.85	98.65	98.64
KNN	92.3	87.05	86.55
SVM	**99.49**	**99.42**	**99.41**
Neural Network	98.3	98.7	98.49

Unfortunately, the corpora used do not contain all singleton variants, which evaluates the **SD** not completed. To ensure the performance of the detector in recognizing each variant; we collect other GitHub java classes with missing variants. Next, we took these collected classes and injected blocks that inhibit the intention of the singleton, in order to vitality of the **SD** classifier to recognize also the incorrect implementations.

After collecting the evaluation set, we labeled them with the corresponding singleton variant name. In the first step, we analyze the existing java class with the **LSTM** classifier to determine every feature's values. In the second step, we evaluate the **SD** classifier by the use of the resulting dataset containing the feature's values. The corresponding results of the **SD** classifier are illustrated in table 12.

All used **ML** algorithms make very good results thanks to the use of specific training datasets. Comparing the results of each **ML** algorithm used in the **SD** classifier, we can see that all used algorithms are performed and make closed results. The **SVM** model has performed excellent results in terms of precision, recall, and F1-score, and can

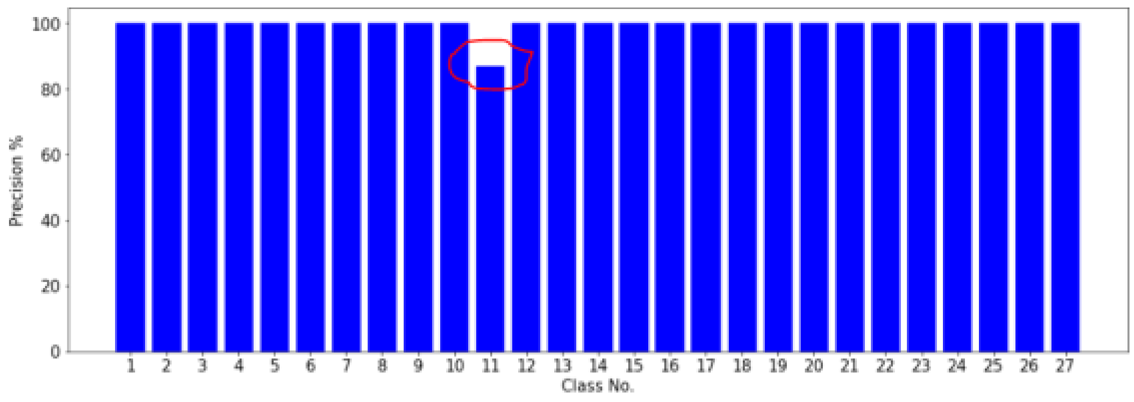


Figure 3: SVM precision results

correctly identify any Singleton candidate with 100% of precision as shown in figure 3 (except one candidate 87%). However, the KNN algorithm is the classifier that makes fewer results. Contrary to **SVM**, **KNN** fails to detect correctly some Variants like Lazy instantiation with double lock mechanism and Replaceable Instance with less than 70% of precision like showing in figure 2.

5.3. *Comparison with similar existing approaches*

In this work, we propose a machine learning-based method to recover the singleton pattern. To position our work against state-of-the-art studies, we select two recent relevant approaches working also with Machine Learning. The first is proposed by Rafael Barbudo et al. [35] named **GEML** and the second is proposed by Najam Nazar et al. [13] named **DPDf**.

5.3.1. *The Benchmark DP detection approaches*

- **GEML Approach**

  GEML is a novel DPD approach based on machine learning and grammar-guided genetic programming (G3P). By the use of software properties, GEML extracts DP characteristics formulated in terms of human-readable rules. Then a machine learning classifier is built based on the established rules to recover 5 DP roles.

  In [35] a comparison between GEML and other DPD methods, including both ML and non-ML is realized. GEML outperform MARPLE techniques [26] with high values in term of accuracy and f1 score. It's also more competitive than two other reference DPD tools which are frequently used for comparative purposes (SSA and Ptidej).

  The experimentation in GEML covers 15 roles of DP and uses implementations from two repositories to compare obtained results against other works. The singleton repository details are:

  –**DPB** : created by the authors of [12] and used to compare the results with MARPLE.

  –**JHotDraw** from PMART [10] was used to compare with non ML-based methods like DePATOS [48], MLDA [49] and SparT [50].

  Table 13 illustrate the comparison results of GEML against other DPD studies. The comparison is conducted based on common ground truth. Four methods are under study. The use of the "-" symbol, is to indicate that the particular DP is not supported by the corresponding approach. The first comparison is based on the DPB corpus. Both GEML and MARPLE perform good results in singleton detection, but GEML outperforms MARPLE by more than 7% and 3% of improves in terms of accuracy and F1 score. The second comparison is based on the JHotDraw project. GEML, MLDA, and SparT can recover all singleton instances. Their great performance is related to the reduced number of true Singleton instances in the test data.

Table 13: Comparing GEML with the state-of-the-art results

ML techniques	Singleton Corpus				
	DPB-Corpus		JHotDraw		
	Accuracy (%)	F1-score (%)	Precision (%)	Recall (%)	F1-score (%)
MARPLE	88	91	-	-	-
GEML	95.61	94.11	100	100	100
DePATOS	-	-	-	-	-
MLDA	-	-	100	100	100
SparT	-	-	100	100	100

- **DPDf Approach**

  The different research results prove that design pattern approaches based on a specific feature are more performed compared to other approaches. The approach proposed in [13] is the first to employ lexical-based code features and machine learning to recover a wide range of design patterns with higher accuracy compared to the state-of-the-art. Our approach also combines semantic analysis, features, and Machine Learning algorithms as techniques, but we have focused only on Singleton Variants recovery. Based on the relevancy measure, we chose to compare our study with the study proposed in [13].

  DPDf Developed in [13] generates a Software Syntactic and Lexical Representation (**SSLR**) by building a call graph and extracting 15 source code features. The **SSLR** is used as an input to build a word-space geometrical model by applying the Word2Vec algorithm. Then, a **DPDf** Machine Learning classifier is created and trained by a labeled dataset and geometrical model.

  DPDf reported results. Nazar has compared his study with two approaches based on code features and machine learning, which are developed by [32] and [27]. For comparing the results, Nazar uses two benchmark Corpus:

  –**P-MAR** T Corpus; containing only 12 Singleton implemented classes.

  –**DPDf** Corpus; contains 100 singleton implemented class.

  The compared results reported by [13] in the detection of the singleton design pattern, are illustrated in the table 14. **DPDf** has improved performance in recovering singleton candidates from the **DPDf-Corpus** but has fairly recovered it from the **P-MART** Corpus. These unbalanced results are caused by the number of instances of singleton existing in each corpus, which have a great impact on the learning of the classifier.

Table 14: Comparing DPDf results with MARPLE and FeatureMap

ML techniques	Singleton Corpus					
	DPDF Corpus			Labeled P-MART		
	Precision (%)	Recall (%)	F1-score (%)	Precision (%)	Recall (%)	F1-score (%)
FeatureMap	65	67	65.98	63	59	60.93
MARPLE-DPD	74.24	69.23	71	74.23	70.18	72.15
DPDf	81.6	68.22	**74.31**	43.33	40	**41.6**

5.3.2. Comparison Against GEML and DPDf methods

- **Comparison Strategy**

Results made by [13] and [35] do not refer exactly to the performance of both approaches to recover singleton pattern instances because other patterns are included. Therefore, we have tested the **DPDf** and **GEML** with Singleton specific data. The **P-MART** Corpus contains a few instances of Singleton Pattern contrarily to the **DPDf** and **DPB** corpus. Otherwise, they represent a none complete data; some variants are absent. Consequently, we construct new data for the evaluation; we bring together all singleton instances existing in **PMART**, **DPB** and **DPDf** repositories, and complete the data with missing variants. We also construct an incoherent structure by injecting correct instances, a structure that inhibits the singleton intent. We try to evaluate with complete data containing a variety of implementations. Details of constructed data are presented in table 15, wit according to the number of correct, incorrect, and incoherent samples in each repository.

Table 15: Evaluating Data Composition

	Singleton Corpus			
	PMART	**DPB**	**DPDf**	**Public project**
Correct	12	58	100	53
Incorrect	-	96	100	-
Incoherent	-	-	-	41
Total size	460			

- **Comparison Results**

We reproduce the public works of [13] and [35] with the only singleton pattern. Then, we evaluate them by the newly created data. The experiment results are conducted by the use of a Random Forest classifier in the evaluation process. Table 16 illustrate obtained results from the experiments.

Table 16: Comparing SD results with DPDf and GEML results

DP Approaches	**Measures**		
	Precision (%)	**Recall (%)**	**F1-score (%)**
DPDf	78.5	70.23	74.13
GEML	81.6	73.4	77.28
Singleton Detector (SD)	**98.96**	**97.63**	**98.29**

- **Discussion**

The comparative results illustrated in table 16 show the performance of the **SD** to recover any variant of the singleton pattern whatever the implementation is. The **SD** reaches the best result (98% of F1 Score), and outperforms **GEML** and **DPDf** by respectively 21% and 24% improvements in terms of F1 Score. Both **GEML** and **DPDf** use approximately 200 samples (including correct and incorrect implementations) for training the model, which is not enough for the best training. The classifier in this case is not able to recover all singleton implementations. His capacity enormously depends on the variants existing in the training data, new variants which not fully trained cannot be detected. However, by the use of complete data that is well created as in the case of **SD** (7000 samples), the classifier will be better trained, and always gives a great performance in the detection process. Our **SD** has the ability not only to recognize the existence of singleton patterns but also the type of implementation. It has also the capacity of recovering implementations incoherent with the singleton intent.

The imbalanced results obtained from the different DPD methods are explained by the number and the variety of implementations existing in the training data. There is another factor that strongly affects the results, which is the characteristics of the source code which are highlighted. **DPDf** approach uses only 15 features to define 12 design patterns, **GEML** propose 23 Grammar operators to describe a variety of design pattern implementations. However, in our work, we use 33 features for defining only the singleton pattern. These enormous numbers of features make a detailed description of the pattern and provide all information needed to identify any variant.

6. Conclusion and future work

We propose in this paper a new singleton design pattern approach based on features and Machine Learning techniques. We are based on the analysis presented in [1], and the different proposed features to create **DTSD** dataset. **DTSD** is used to train the **SD** classifier, in which we try to make rules defining a various number of singleton implementations. In the dataset, we give a combination of feature values to categorize each variant. We have tried to be sure that the data contain the most number of implementations to perfectly train the model. Next, we build a Singleton Detector based on different Machine Learning classifiers. We trained the supervised classifier by the labeled dataset **DTSD**, and we evaluate its performance with other data collected from the GitHub java corpus, and singleton variants existing in **DPDf**, **DPB** and **P-MART** corpus.

We apply three standard statistical measures namely precision, recall, and F1-Score to evaluate the performance of the created classifiers. A comparison between different used Machine Learning algorithms to create the **SD** is realized. The different results show that the use of the created dataset makes any classifier easily able to recover any variants of the singleton pattern although there is a slight difference in performance. The Empirical result shows that our proposed approach can recover any non-standard singleton variant, even incorrect implementation destroyed the singleton intent with approximately 99% on both precision and recall. Our approach outperforms recent relevance approaches by more than 20% improvements in terms of standard measures.

While our classifier's performance is promising, as furfur work we gonna applies it to recover a wide range of Software Design Patterns. The choice of source code as refactoring support is very important and interesting, but we should not limit ourselves only to this support, we try to switch to the model as refactoring support.

Declarations

Conflict of interest Abir Nacef declares that she has no conflict of interest.

References

[1]A. Nacef, A. Khalfallah, S. Bahroun, S. Ben Ahmed, Defining and extracting singleton design pattern information from object-oriented software program, in: C. Bădică, J. Treur, D. Benslimane, B. Hnatkowska, M. Krótkiewicz (Eds.), Advances in Computational Collective Intelligence, Springer International Publishing, Cham, 2022, pp. 713–726.

[2]E. Gamma, R. Helm, R. Johnson, J. M. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, 1st Edition, Addison-Wesley Professional, 1994.
URL http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612/ref=ntt$_{a}t_{e}$ p_{d} pi_{1} [3]G. Antoniol, R. Fiutem, L. Cristoforetti, Design pattern recovery in object-oriented software, in: 6th International Workshop on Program Comprehension (IWPC '98), June 24-26, 1998, Ischia, Italy, IEEE Computer Society, 1998, pp. 153–160. doi:10.1109/WPC.1998.693342.
URL https://doi.org/10.1109/WPC.1998.693342

[4]N. Shi, R. A. Olsson, Reverse engineering of design patterns from java source code, in: 21st IEEE /ACM International Conference on Automated Software Engineering (ASE 2006), 18-22 September 2006, Tokyo, Japan, IEEE Computer Society, 2006, pp. 123–134. doi:10.1109/ASE.2006.57.
URL https://doi.org/10.1109/ASE.2006.57

[5]D. Heuzeroth, T. Holl, G. H ögström, W. Löwe, Automatic design pattern detection, in: 11th International Workshop on Program Comprehension (IWPC 2003), May 10-11, 2003, Portland, Oregon, USA, IEEE Computer Society, 2003, pp. 94–104. doi:10.1109/WPC.2003.1199193.
URL https://doi.org/10.1109/WPC.2003.1199193

[6]A. Hindle, E. T. Barr, M. Gabel, Z. Su, P. T. Devanbu, On the naturalness of software, Commun. ACM 59 (5) (2016) 122–131. doi:10.1145/2902362.
URL https://doi.org/10.1145/2902362

[7]I. J. Goodfellow, Y. Bengio, A. C. Courville, Deep Learning, Adaptive computation and machine learning, MIT Press, 2016.
URL http://www.deeplearningbook.org/

[8]K. Stencel, P. Wegrzynowicz, Implementation variants of the singleton design pattern, in: R. Meersman, Z. Tari, P. Herrero (Eds.), On the Move to Meaningful Internet Systems: OTM 2008 Workshops, OTM Confederated International Workshops and Posters, ADI, AWeSoMe, COMBEK, EI2N, IWSSA, MONET, OnToContent + QSI, ORM, PerSys, RDDS, SEMELS, and SWWS 2008, Monterrey, Mexico, November 9-14, 2008. Proceedings, Vol. 5333 of Lecture Notes in Computer Science, Springer, 2008, pp. 396–406. doi:10.1007/978-3-540-88875-8_61.
URL https://doi.org/10.1007/978-3-540-88875-8 61 _

[9]K. Stencel, P. Wegrzynowicz, Detection of diverse design pattern variants, in: 15th Asia-Pacific Software Engineering Conference (APSEC

2008), 3-5 December 2008, Beijing, China, IEEE Computer Society, 2008, pp. 25–32. doi:10.1109/APSEC.2008.67.
URL https://doi.org/10.1109/APSEC.2008.67

[10]Y.-G. Gu éhéneuc, P-mart : Pattern-like micro architecture repository, 2007.

[11]Y. g. Gu eh eneu, "pmart: Pattern-like micro architecture repositorydoi:Available: http: //www-etud.iro.umontreal.ca/ptidej/Publications/Documents/EuroPLoP07PRa.doc.pdf.

[12]F. A. Fontana, A. Caracciolo, M. Zanoni, DPB: A benchmark for design pattern detection tools, in: T. Mens, A. Cleve, R. Ferenc (Eds.), 16th European Conference on Software Maintenance and Reengineering, CSMR 2012, Szeged, Hungary, March 27-30, 2012, IEEE Computer Society, 2012, pp. 235–244. doi:10.1109/CSMR.2012.32.
URL https://doi.org/10.1109/CSMR.2012.32

[13]N.Nazar, A. Aleti, Y. Zheng, Feature-based software design pattern detection, J. Syst. Softw. 185 (2022) 111179. doi:10.1016/j.jss.2021.111179.
URL https://doi.org/10.1016/j.jss.2021.111179

[14]M. Allamanis, C. Sutton, Mining source code repositories at massive scale using language modeling, in: T. Zimmermann, M. D. Penta, S. Kim (Eds.), Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013, IEEE Computer Society, 2013, pp. 207–216. doi:10.1109/MSR.2013.6624029.
URL https://doi.org/10.1109/MSR.2013.6624029

[15]G. Rasool, I. Philippow, P. M äder, Design pattern recovery based on annotations, Adv. Eng. Softw. 41 (4) (2010) 519–526. doi:10.1016/j.advengsoft.2009.10.014.
URL https://doi.org/10.1016/j.advengsoft.2009.10.014

[16]P. Wegrzynowicz, K. Stencel, Relaxing queries to detect variants of design patterns, in: M. Ganzha, L. A. Maciaszek, M. Paprzycki (Eds.), Proceedings of the 2013 Federated Conference on Computer Science and Information Systems, Kraków, Poland, September 8-11, 2013, 2013, pp. 1559–1566.
URL https://ieeexplore.ieee.org/document/6644226/

[17]B. Combemale, J. Kienzle, G. Mussbacher, H. Ali, D. Amyot, M. Bagherzadeh, E. Batot, N. Bencomo, B. Benni, J. Bruel, J. Cabot, B. H. C. Cheng, P. Collet, G. Engels, R. Heinrich, J. Jézéquel, A. Koziolek, S. Mosser, R. H. Reussner, H. A. Sahraoui, R. Saini, J. Sallou, S. Stinckwich, E. Syriani, M. Wimmer, A hitchhiker's guide to model-driven engineering for data-centric systems, IEEE Softw. 38 (4) (2021) 71–84. doi:10.1109/MS.2020.2995125.
URL https://doi.org/10.1109/MS.2020.2995125

[18]T. Z. Ahram (Ed.), Advances in Artificial Intelligence, Software and Systems Engineering - Proceedings of the AHFE 2020 Virtual Conferences on Software and Systems Engineering, and Artificial Intelligence and Social Computing, July 16-20, 2020, USA, Vol. 1213 of Advances in Intelligent Systems and Computing, Springer, 2021. doi:10.1007/978-3-030-51328-3.
URL https://doi.org/10.1007/978-3-030-51328-3

[19]M. von Detten, S. Becker, Combining clustering and pattern detection for the reengineering of component-based software systems, in: I. Crnkovic, J. A. Stafford, D. C. Petriu, J. Happe, P. Inverardi (Eds.), 7th International Conference on the Quality of Software Architectures, QoSA 2011 and 2nd International Symposium on Architecting Critical Systems, ISARCS 2011. Boulder, CO, USA, June 20-24, 2011, Proceedings, ACM, 2011, pp. 23–32. doi:10.1145/2000259.2000265.
URL https://doi.org/10.1145/2000259.2000265

[20]H. Kim, C. Boldyre ff, A method to recover design patterns using software product metrics, in: W. B. Frakes (Ed.), Software Reuse: Advances in Software Reusability, 6th International Conerence, ICSR-6, Vienna, Austria, June 27-29, 2000, Proceedings, Vol. 1844 of Lecture Notes in Computer Science, Springer, 2000, pp. 318–335. doi:10.1007/978-3-540-44995-9_19.
URL https://doi.org/10.1007/978-3-540-44995-9 19 _

[21]Y. Gu éhéneuc, J. Guyomarc'h, H. A. Sahraoui, Improving design-pattern identification: a new approach and an exploratory study, Softw. Qual. J. 18 (1) (2010) 145–174. doi:10.1007/s11219-009-9082-y.
URL https://doi.org/10.1007/s11219-009-9082-y

[22]J. Niere, W. Sch äfer, J. P. Wadsack, L. Wendehals, J. Welsh, Towards pattern-based design recovery, in: W. Tracz, M. Young, J. Magee (Eds.), Proceedings of the 24th International Conference on Software Engineering, ICSE 2002, 19-25 May 2002, Orlando, Florida, USA, ACM, 2002, pp. 338–348. doi:10.1145/581339.581382.
URL https://doi.org/10.1145/581339.581382

[23]H. W. Y. F. Satoru Uchiyama, Atsuto Kubo, Detecting design patterns in object-oriented program source code by using metrics and machine learning, Proceedings of the 5th International Workshop on Software Quality and Maintainability.

[24]Z. Balanyi, R. Ferenc, Mining design patterns from C ++ source code, in: 19th International Conference on Software Maintenance (ICSM 2003), The Architecture of Existing Systems, 22-26 September 2003, Amsterdam, The Netherlands, IEEE Computer Society, 2003, pp. 305–314. doi:10.1109/ICSM.2003.1235436.
URL https://doi.org/10.1109/ICSM.2003.1235436

[25]R. Ferenc, Á. Beszédes, L. J. Fülöp, J. Lele, Design pattern mining enhanced by machine learning, in: 21st IEEE International Conference on Software Maintenance (ICSM 2005), 25-30 September 2005, Budapest, Hungary, IEEE Computer Society, 2005, pp. 295–304. doi:10.1109/ICSM.2005.40.
URL https://doi.org/10.1109/ICSM.2005.40

[26]M. Zanoni, F. A. Fontana, F. Stella, On applying machine learning techniques for design pattern detection, J. Syst. Softw. 103 (2015) 102–117. doi:10.1016/j.jss.2015.01.037.
URL https://doi.org/10.1016/j.jss.2015.01.037

[27]F. A. Fontana, M. Zanoni, A tool for design pattern detection and software architecture reconstruction, Inf. Sci. 181 (7) (2011) 1306–1324. doi:10.1016/j.ins.2010.12.002.
URL https://doi.org/10.1016/j.ins.2010.12.002

[28]A. Panich, W. Vatanawood, Detection of design patterns from class diagram and sequence diagrams using ontology, in: 15th IEEE /ACIS International Conference on Computer and Information Science, ICIS 2016, Okayama, Japan, June 26-29, 2016, IEEE Computer Society, 2016, pp. 1–6. doi:10.1109/ICIS.2016.7550771.
URL https://doi.org/10.1109/ICIS.2016.7550771

[29]A. D. Lucia, V. Deufemia, C. Gravino, M. Risi, Design pattern recovery through visual language parsing and source code analysis, J. Syst. Softw. 82 (7) (2009) 1177–1193. doi:10.1016/j.jss.2009.02.012.
URL https://doi.org/10.1016/j.jss.2009.02.012

[30]B. D. Martino, A. Esposito, A rule-based procedure for automatic recognition of design patterns in UML diagrams, Softw. Pract. Exp. 46 (7) (2016) 983–1007. doi:10.1002/spe.2336.
URL https://doi.org/10.1002/spe.2336

[31]N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, S. T. Halkidis, Design pattern detection using similarity scoring, IEEE Trans. Software Eng. 32 (11) (2006) 896–909. doi:10.1109/TSE.2006.112.
URL https://doi.org/10.1109/TSE.2006.112

[32]H. Thaller, L. Linsbauer, A. Egyed, Feature maps: A comprehensible software representation for design pattern detection, in: X. Wang, D. Lo, E. Shihab (Eds.), 26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019, IEEE, 2019, pp. 207–217. doi:10.1109/SANER.2019.8667978.
URL https://doi.org/10.1109/SANER.2019.8667978

[33]A. Chihada, S. Jalili, S. M. H. Hasheminejad, M. H. Zangooei, Source code and design conformance, design pattern detection from source code by classification approach, Appl. Soft Comput. 26 (2015) 357–367. doi:10.1016/j.asoc.2014.10.027.
URL https://doi.org/10.1016/j.asoc.2014.10.027

[34]S. Hussain, J. Keung, A. A. Khan, A. Ahmad, S. Cuomo, F. Piccialli, G. Jeon, A. Akhunzada, Implications of deep learning for the automation of design patterns organization, J. Parallel Distributed Comput. 117 (2018) 256–266. doi:10.1016/j.jpdc.2017.06.022.
URL https://doi.org/10.1016/j.jpdc.2017.06.022

[35]R. Barbudo, A. Ram ́ırez, F. Servant, J. R. Romero, GEML: A grammar-based evolutionary machine learning approach for design-pattern detection, J. Syst. Softw. 175 (2021) 110919. doi:10.1016/j.jss.2021.110919.
URL https://doi.org/10.1016/j.jss.2021.110919

[36]S. Alhusain, S. Coupland, R. I. John, M. Kavanagh, Towards machine learning based design pattern recognition, in: 13th UK Workshop on Computational Intelligence, UKCI 2013, Guildford, United Kingdom, September 9-11, 2013, IEEE, 2013, pp. 244–251. doi:10.1109/UKCI.2013.6651312.
URL https://doi.org/10.1109/UKCI.2013.6651312

[37]T. M. Mitchell, Machine learning, International Edition, McGraw-Hill Series in Computer Science, McGraw-Hill, 1997.
URL https://www.worldcat.org/oclc/61321007

[38]O. Razeghi, J. A. Sol ́ıs-Lemus, A. W. C. Lee, R. Karim, C. Corrado, C. H. Roney, A. de Vecchi, S. A. Niederer, Cemrgapp: An interactive medical imaging application with image processing, computer vision, and machine learning toolkits for cardiovascular research, SoftwareX 12 (2020) 100570. doi:10.1016/j.softx.2020.100570.
URL https://doi.org/10.1016/j.softx.2020.100570

[39]X. Tian, D. Shi, K. Zhang, H. Li, L. Zhou, T. Ma, C. Wang, Q. Wen, C. Tan, Machine-learning model for prediction of martensitic transformation temperature in nimnsn-based ferromagnetic shape memory alloys, Computational Materials Science 215 (2022) 111811. doi:https://doi.org/10.1016/j.commatsci.2022.111811.
URL https://www.sciencedirect.com/science/article/pii/S0927025622005225

[40]A. Sherstinsky, Fundamentals of recurrent neural network (RNN) and long short-term memory (LSTM) network, CoRR abs /1808.03314 (2018). arXiv:1808.03314.
URL http://arxiv.org/abs/1808.03314

[41]J. Chung, Ç. G ülçehre, K. Cho, Y. Bengio, Empirical evaluation of gated recurrent neural networks on sequence modeling, CoRR abs/1412.3555 (2014). arXiv:1412.3555.
URL http://arxiv.org/abs/1412.3555

[42]S. Elmahdy, T. Ali, M. Mohamed, Regional mapping of groundwater potential in ar rub al khali, arabian peninsula using the classification and regression trees model, Remote. Sens. 13 (12) (2021) 2300. doi:10.3390/rs13122300.
URL https://doi.org/10.3390/rs13122300

[43]K. P. Murphy, Machine learning - a probabilistic perspective, Adaptive computation and machine learning series, MIT Press, 2012.

[44]D. A. M.Schnyer, Support vector machine, in: Machine Learning, ScienceDirect, 2020, pp. 101–121.
URL https://doi.org/10.1016/B978-0-12-815739-8.00006-7

[45]L. E. Peterson, K-nearest neighbor (2009). doi::10.4249 /scholarpedia.1883.

[46]K. Doya, D. Wang, Announcement of the neural networks best paper award, Neural Networks 145 (2022) xix. doi:10.1016 /S0893-6080(21)00464-0.
URL https://doi.org/10.1016/S0893-6080(21)00464-0

[47]H. Bhandari, B. Rimal, N. R. Pokhrel, R. Rimal, K. Dahal, Lstm-sdm: An integrated framework of lstm implementation for sequential data modeling, Software Impacts 14 (08 2022). doi:10.1016/j.simpa.2022.100396.

[48]D. Yu, P. Zhang, J. Yang, Z. Chen, C. Liu, J. Chen, E fficiently detecting structural design pattern instances based on ordered sequences, J. Syst. Softw. 142 (2018) 35–56. doi:10.1016/j.jss.2018.04.015.
URL https://doi.org/10.1016/j.jss.2018.04.015

[49]M. Al-Obeidallah, M. Petridis, S. Kapetanakis, A multiple phases approach for design patterns recovery based on structural and method signature features, Int. J. Softw. Innov. 6 (3) (2018) 36–52. doi:10.4018/IJSI.2018070103.
URL https://doi.org/10.4018/IJSI.2018070103

[50]R. Xiong, B. Li, Accurate design pattern detection based on idiomatic implementation matching in java language context, in: X. Wang, D. Lo, E. Shihab (Eds.), 26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019, IEEE, 2019, pp. 163–174. doi:10.1109/SANER.2019.8668031.
URL https://doi.org/10.1109/SANER.2019.8668031