Automatic Detection of Implicit and Typical Implementation of Singleton Pattern Based on Supervised Machine Learning

Abir Nacef[1], Sahbi Bahroun[2], Adel Khalfallah[1] and Samir Ben Ahmed[1]
[1]*Faculty of Mathematical, Physical and Natural Sciences of Tunis (FST), Computer Laboratory for Industrial Systems, Tunis El Manar University, Tunisia*
[2]*Higher Institute of Computer Science (ISI), Limtic Laboratory, Tunis El Manar University, Tunisia*

Keywords: Supervised ML, Reverse engineering, Refactoring, Singleton Design Pattern.

Abstract: Reverse engineering, based on design pattern recognition and software architecture refactoring, allows practitioners to focus on the overall architecture of the system without worrying about the programming details used to implement it. In this paper, we focus on the automatization of these tasks working on Singleton design Pattern (SP). The first task is the detection of the SP in its standard form, we named the detected structure as Singleton Typical implementation (ST). The second task consists of detecting structures which need the injection of the SP (Refactoring), these structures are named Singleton Implicit implementations (SI). All SP detection methods can only recover the typical form, even if they support different variants. However, in this work, we propose an approach based on supervised Machine Learning (ML) to extract different variants of SP in both ST and SI implementations and filter out structures which are incoherent with the SP intent. Our work consists of three phases; the first phase includes SP analysis, identifying implementation variants (ST and SI), and defining features for identifying them. In the second phase, we will extract feature values from the Java program using the LSTM classifier based on structural and semantic analysis. LSTM is trained on specific created data named SFD for a classification task. The third phase is SP detection, we create an ML classifier based on different algorithms, the classifier is named SPD. For training the SPD we create a new structured data named SDD constructed from features combination values that identify each variant. The SPD reaches 97% in terms of standard measures and outperforms the state-of-the-art approaches on the DPB corpus.

1 INTRODUCTION

Design Patterns (DPs) (Gamma et al., 1994) are often reusable solutions to common software design problems. The information about intent, applicability, and motivation extracted from the pattern semantic is the key to the development and redevelopment of a system to solve a specific problem. However, several factors can lead to the absence of documentation. More than that, developers may not have much experience to understand patterns well, and this can lead to the possibility of introducing coding errors that break the DP intent or bad solution to resolve the problem. Hence, needed information about a decision, intent, and pattern implementation used is lost. Extracting this information can improve the quality of source code and help to develop, and maintain, the system. So, automatically identifying the adopted pattern (DPD) is the solution to solve these limits.

Our work is the first to identify not only different variants of the ST, but also SP in incorrect structures and different variants of SI. However, the identification process is still considered a difficult task due to the informal structure of the source code and a large number of implementations. Only the structural analysis of the source code cannot capture the pattern intent, and semantic analysis is required. Furthermore, directly processing the source code or even one of its representations (such as AST, graph, or another form) shows poor performance compared to methods using well-defined features. The use of features reduces the search space, and false positive rate, and improves accuracy.

For better results and to simplify the SP detection, we propose a based-feature approach using structural and semantic analysis of the Java program with ML techniques. First, we analyze the program with LSTM classifiers, to extract features value. LSTM classifiers are trained by a specific created dataset named SFD. Second, by analyzing the SP variant's struc-

Nacef, A., Bahroun, S., Khalfallah, A. and Ben Ahmed, S.
Automatic Detection of Implicit and Typical Implementation of Singleton Pattern Based on Supervised Machine Learning.
DOI: 10.5220/0011634000003393
In *Proceedings of the 15th International Conference on Agents and Artificial Intelligence (ICAART 2023) - Volume 3*, pages 202-210
ISBN: 978-989-758-623-1; ISSN: 2184-433X

ture and based on extracted feature's values, we construct a new dataset named SDD to train an SPD. The SDD is rule-based data guided by specific code properties of various implementations that can exist for ST and SI. For building the SPD we have used different ML algorithms the most used for DPD, evaluated and compared them based on collected data from public Java corpus SED. Finally, we compare our proposed approach with state-of-the-art approaches on **DPB** (Fontana et al., 2012) repository. Empirical results have proven that our proposed method performs others methods in SP detection. The main contributions of this paper consist on:

- Analyzing the SP intent, identifying variants, and proposing a set of features defining each one.
- Analyzing Java program with LSTM Classifier to extract feature values.
- Creating three datasets; SFD with 5000 snippets of code used to train the LSTM. SDD with 12000 samples to train the SPD. SED containing 312 Java files to evaluate the SPD.
- Demonstrating that our SPD outperforms a state-of-the-art approaches with a substantial margin in terms of standard measures.

The rest of the paper is organized as follows; In section 2 we present the related work and the contribution made to them. In section 3, we define the different declarations of the SP, we indicate variants reported by the SPD and the proposed features. In section 4, we present the method process and the related used technologies. In section 5, details about the created data are presented, and the obtained results in both phases are illustrated and discussed. In the end, a comparison with the state of the art is made. Finally, in Section 6 a conclusion and future work is presented.

2 RELATED WORKS

Working on source code, several approaches are proposed. In this section, we will discuss relevant studies, giving special attention to those using features based on ML.

Many techniques are based on similarity scoring by using graphs to represent structural information. The method proposed by (Yu et al., 2015) search a set of substructures to identifying DP implementations. Then, to optimize the result, the method signature is compared against a set of predefined templates corresponding to the DP. As an improving work, authors in (Mayvan et al., 2017) ameliorate the performance of substructures search process by conveniently partitioning the project graph.

Visual language is also used (Lucia et al., 2009) as a method to represent relations between structural properties of DPs and source code.

Features have been used in previous work to express DPs. Different work like (Rasool and Mäder, 2011) recorded it with annotations for building automated analyzers. Working with features depends enormously on the authors' acknowledgment and pattern analysis.

Given its power to automatically detect DP, ML has been used in different works. In (Chihada et al., 2015), a DP classifier is created based on a Support Vector Machine (SVM). The input data is a labeled set of manually identified implementations and a set of associated metrics. However, the method used in MARPLE (Zanoni et al., 2015) doesn't take software metrics as input, but a set of structural and behavioral properties (e.g., abstract class, abstract method invocation, and extended inheritance). These structures is used to find correlations among code elements and roles or relations within DP.

More recent work proposed by (Thaller et al., 2019) applies convolutional neural networks and random forest to learn from feature maps. The definition of these elements is based on the occurrences of certain microstructures in the code. (Nazar et al., 2022) use code features represented as code2vec purports to achieve a semantic understanding of source code. The code2vec uses paths along a method's AST to make DPs predictions.

Many approaches detect SP variants. However, we are the first to recognize not only the different variants, but also their possible combinations with their respective names and incoherent structures that destroy the pattern intent. Our approach is the first to create dataset specific to the SP (taking in consideration different forms (Typical/ Implicit), different variants, different combinations, incoherent structure, variant's name).

3 SINGLETON VARIANTS AND FEATURES

In this section, we will define SP and specify the difference between its type of implementation (typical and implicit). Next, we're going to represent the reported variants and highlight features that better describe each variant.

3.1 ST and SI Implementation

Using the SP or other structure that avoids it, is ultimately a design decision that needs to be made, and the consequences of that need to be understood and documented. We named all used structures to create an application with only one instance, as the SP. The formal structure is named Singleton's typical implementation, and the other structure is named Singleton's implicit implementation.

3.1.1 Definition of ST

The ST consists of creating a class that has only one instance and provides a global point to access it. There are two forms of implementing that. The eager instantiation allows the creation of an instance at loading time, and the lazy instantiation enables the creation when an instance is required. Using a global access point allows access to the instance of the class from anywhere in the application. More than that, to be a self-owner, the client application does not need to perform additional steps in the object creation, configuration, or destruction. To prevent other classes from instantiating it, the constructor is meant to be private, and the instance will be accessed with static properties or methods to get the preconfigured object. Listings 1 and 2 show examples of eager and lazy instantiation.

Listing 1: Example of Singleton Eager instantiation.

```
public class C1 {
   private static C1 instance = new C1();
   private C1(){}
   public static C1 getInstance(){
      return instance;}}
```

Listing 2: Example of Singleton Lazy instantiation.

```
class C2{
    private static C2 obj;
    private C2() {}
    public static C2 getInstance()
    { if (obj==null)
            obj = new C2();
        return obj; } }
```

3.1.2 Definition of SI

The SI is all used structure to create one instance of a class without using the formal and complete ST solution. Simply, it's a specification of a set of classes and methods that work together to ensure that only a single instance of a class should be created, and a single object can be used by all other classes. To ensure that, different structures can exist. At first, the class can be stateless, no global object is needed, just provide helper functions that don't require more information than parameters. Second, the use of static aspects (Mono state, static class) can be a way to ensure the creation of only one instance of a class. Also, the control of instantiation by using a Boolean variable, or a counter variable can be considered as SI as well as the use of class Enum. For sub-classing, it's possible to use a base class that represents an abstract sandbox method. Finally, developers can provide all needed services inside an interface. This interface has the goal to provide global access to the object. The implemented structure is called Service Locator.

3.2 ST and SI Reported Variants

Different variant of SP are defined and presented in (Stencel and Wegrzynowicz, 2008), (Gamma et al., 1994) and (Nacef et al., 2022). In this work, reported variants are shown in table 1.

The use of the SP is very useful to allow only one instance of a class, but a common mistake can lead to the accidental creation of multiple instances, the case of listing 3. So, we should verify that the structure preserves the SP purpose. If any error exists, the implemented structure must be analyzed and detected to eliminate inconsistency between design structure and intent.

Listing 3: Example of Singleton incorrect implementation.

```
class C3{
    private static C3 obj;
    private C3() {}
    public static C3 getInstance()  {
        if (obj==null)
            obj = new C3();
        return obj; }
    public static C3 getInstance2()
    { return(new C3());}}
```

There are different ways to implement the SI, but we try to report the most often used by Java programmers. Table 1 represent ST and SI-reported variants with ST that have incoherent structure.

3.3 Used Features

Relevant features can strongly affect the ability of the model to recognize each variant in its different representations. These features also have an impact on the prediction rate and can reduce the number of false-positive candidates. We are based on the detailed analysis of the SP realized by (Nacef et al., 2022), in which they propose 33 features for identifying ST variants with correct and incorrect structures. We propose 9 other features which seem relevant to the SI

Table 1: Reported SP variants.

Singleton Variants	
-Eager	-Eager Incorrect structure
-Lazy	-Lazy Incorrect structure
-Subclassed	-Subclassed Incorrect structure
-Replaceable Instance	-Replaceable Instance Incorrect structure
-Base Class	-Different Access Point
-Service Locator	-Different Access Point Incorrect structure
-Limiton	-Delegated Construction
-Enum Class	-Delegated Construction Incorrect structure
-Social Singleton	-Different Placeholder
-Generic Singleton	-Different Placeholder Incorrect structure
-Static data	-Object parameter in Function
-Control Instantiation	

definition. Furthermore, we try to describe SI with features that have an elementary structure to facilitate as much as possible their extraction from the source code.

Adopting the SP has the purpose to control the creation of objects and limiting the number to one. To verify the consistency between structure and intent, we must be sure that only one instance of a class is declared and created. These two features are the key to a true candidate for SP listing 3.
Table 2 represents all used features for the SP identification.

4 SUPERVISED ML FOR THE SP DETECTION

In this section, we describe our proposed method and related techniques used in SP detection. The SP recognition problem is solved in three phases, as shown in fig. 1. The first phase (P1) is dedicated to analyzing the SP intent, structure, and configuration, this analysis has the goal to highlight the pattern characteristic. The second phase (P2), consists of extract values of the proposed features from the source code by structural and semantic analysis. These features' values lead to the third phase (P3) where SP recognition is taking place. The three phases are discussed below.

Table 2: Proposed Features.

Abb.	Features
IRE	Extend inheritance
IRI	Implements inheritance
CA	Class accessibility
GOD	Global class attribute declaration
AA	Class attribute accessibility
SR	Static class attribute
ON	Have only one class attribute
COA	Constructor accessibility
HC	Hidden Constructor
ILC	Instantiate when loading class
GAM	Global accessor method
PSI	Public Static accessor method
GSM	Global setter method
PST	Public static setter method
INC	Use of Inner class
EC	Use External Class
RINIC	Returning instance in inner class
RINEC	Returning instance in External class
CS	Control instantiation
HGM	One method to generate instance
DC	Double check locking
RR	Return reference of the SP
CNI	Variable to count instance number
CII	Internal static read-only instance
DM	Use delegated method
GMS	Global synchronized method
IGO	Initialize global class attributes
LNI	Limit the number of instances
SCI	Use string to create instance
SB	Static Block
AFL	Allowed Friend List
CAFB	Control access to friend behavior
UR	Type for generic instantiation
EC	Enum Class
CSA	Class with Static attributes
CFA	Class with Final attributes
CSM	Class with Static Methods
COP	Constructor with Object Parameter
MOP	Method with Object Parameter
BV	Boolean Variable to Instantiate
MSR	Create Map for serves references
AMDP	Abstract method with data parameter

4.1 P1: SP Analysing and ST, SI Variants Identification

The goal of SP analysis is to define SP variants in both typical and implicit implementation. The purpose in this context is to identify features that hold information about each variant. In our case, we are based on the work of (Nacef et al., 2022), in which

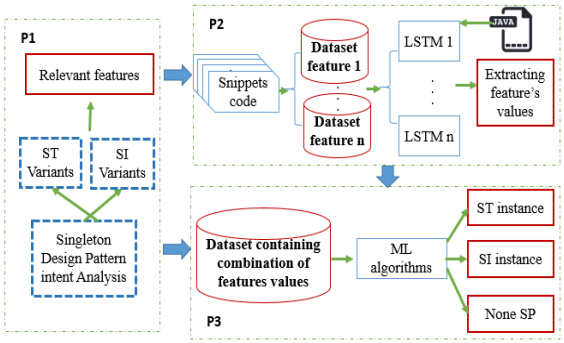

Figure 1: The SP detection process.

they analyze different variants of ST for identifying features. Based on their analysis and different public Java project analysis, we define variants for SI, and we propose additional features for their extraction.

4.2 P2: Feature's Value Extraction

To extract feature values from source code, we use the same method proposed by (Nacef et al., 2022). In their work, they create a set of data containing snippets of code. Specific data corresponding to each feature is created and used to train an LSTM classifier. For newly proposed features, we also create specific data for their extraction from the Java program. Resulting feature values extracted from the P2 are used for creating data containing the feature's combination identifying each variant.

4.3 P3: SP Detection

With the detailed analysis of different implementations realized by (Nacef et al., 2022), we construct a data named SDD based on SP properties rules. Rules in general serve as an established mechanism for encoding human knowledge, and we used them to characterize SP instances. It represents a simple and understandable way to define implementation variants.

Thereafter, we built an SPD with the use of different ML classifiers. We have selected four ML algorithms the most used on DPD (KNN, SVM, Random Forest, and Naive Bayes). The SPD is trained by the SDD data to identify candidates for ST and SI instances. The ML algorithms are created separately by training and testing each one exclusively on specific data. By learning from labeled samples, the SPD learns new rules and discovers more information defining each class.

5 EXPERIMENTAL SETUP

In this section, we will represent at first the user data to train and evaluate LSTM and SPD Classifiers. Next, we show, compare, and discuss the obtained results.

5.1 Dataset Details and Preparation

The data is the essence of ML, the better the data is constructed, the better the learning process will be. So that, to have a good model, we should carefully construct the training data. The ideal way to prepare a training dataset is to manually analyze open-source applications to discover a wide range of SP implementations. Whereas this method is time-consuming as it requires careful study and in-depth understanding, we have realized it as the first step in our method process. We have constructed three datasets for training and evaluating the used classifiers: the SFD, the SDD, and the SED, which the details are below and listed in table 3.

Table 3: Data details.

Data	Size
SFD	5000
SDD	12000
DPB	58 Correct ST, 58 Incorrect ST, 96 No SP
SEDC	100 Correct SI

5.1.1 SFD Data

The SFD is a set of data used for training the LSTM classifier to extract values for the newly added features. The SFD has a global size of 5000 samples, and it's composed of 9 data. Each data is created according to the specific properties of one of the features. We have collected a variety of implementations that can satisfy the feature property on a true/false basis. Each implementation represents snippets of code that are labeled according to the checked information. Listings 4 and 5 show examples of snippets of code that are labeled as true/false in *Boolean variable for instantiating* feature's data.

Listing 4: Example of correct implementation of the BV feature.

```
private static C1 instance1;
private Boolean noInstance = true;
public static C1 getInstance(){
 if(noInstance)
   {instance1 = new C1();
    return instance1;}
  else{System.out.println("Instance exist");}}
```

Listing 5: Example of incorrect implementation of the BV feature.

```
private static C1 instance1;
private Boolean noInstance = true;
public static C1 getInstance(){
 if (noInstance)
  {System.out.println("Nonexistent instance");}
  instance1 = new C1();
  return instance1;}
```

5.1.2 SDD Data

The SDD is structured labeled data with 12000 samples. Each sample contains a combination of feature values to satisfy an SP variant. The data has 42 categorical features and 4 classes (ST / SI / ST incorrect structure/ No SP). Table 4, 5 and 6 present an extract from the SDD dataset.
Eager implementation is characterized by a bloc that instantiates the instance when loading a class (ILC), with global static reference (GOD, AA) and private constructor (CA). However, if there are different references (ON) to the instance or their exit more than a block or method for generating this instance (HGM) the structure will be incoherent with the pattern intent. For Lazy implementation (table 5), we must verify that exists only one method to generate an instance (HGM) that is static (GAM, PSI, RR) and contains a block for controlling instantiation (CS). Otherwise, the implementation will be incorrect. For static classes (table 6), attributes and methods (COP, CSA) must be static and can use any type of access modifier (private, protected, public, or default). A static class is implemented is through an inner class or a nested class (SIC) otherwise it is considered an incorrect implementation.

Table 4: Eager variant (correct /incorrect structure).

Features	Eager	Eager Incorrect
CA	public / final	Public / final
GOD	True	True
AA	Private / Final	Private / Final
SR	True	True
COA	private	private
ILC	True	True
GAM	True	True
PSI	True	True
RR	True	True
ON	True	**False**
HGM	True	**False**

Table 5: Lazy variant (correct /incorrect structure).

Features	Lazy Thread safe	Lazy Incorrect
CA	public / final	Public / final
GOD	True	True
AA	Private / Final	Private / Final
SR	True	True
COA	private	private
ILC	False	False
GAM	True	True
PSI	True	True
RR	True	True
CS	True	**False**
DC	True	**False**
GMS	True	True
ON	True	**False**
HGM	True	**False**

Table 6: Static class (correct /incorrect structure).

Class	Features			
	COP	CSA	SIC	CSM
True SII	True	True	True	True
False SII	True	True	**False**	True

5.1.3 SED Data

The SED is a combination of two public collected data with a global size of 312. More specifically, we conduct our experiments with implementations from two repositories, namely DPB and SEDC. DPB (Fontana et al., 2012) is a peer-validated repository frequently used in DPD studies because it includes both positive and negative samples of a variety of DPs. We focus in this work on SP detection, so we have used only samples corresponding to this pattern as listed in table 3 Since the DPB repository is not claimed to be complete and doesn't contain all considered variants, we have created another data named SEDC containing files collected from GitHub public Java projects [1]. Furthermore, to test the effectiveness of our method in detecting ST instances whose structure does not match the pattern intent, we modify the correct instance at the DPB level.

[1] https://github.com/topics/service-locator?l=java / https://github.com/topics/service-locator?o=asc&s=forks / https://github.com/topics/singleton?o=desc&s=updated / https://github.com/topics/enum?l=java / https://github.com/topics/singleton?l=java / https://github.com/topics/static-class?l=java / https://github.com/topics/static-variables

5.2 Results and Evaluations

5.2.1 Validation of the Detection

The first phase of our method is to analyze the source code to extract feature values. We have built 9 LSTM classifiers and trained them by the feature's corresponding data. After the training phase using the SFD, we pass the evaluation process. We evaluate the newly created LSTM classifiers and those previously created by (Nacef et al., 2022) with the SED data. The obtained result is illustrated in table 7.
All results for the LSTM classifiers and the SPD classifier are then reported according to the following standard classification measures: *Precision* it's the exact measure of how many Singleton instances were positively identified as positive, *Recall* corresponds to the amount of SP classes retrieved over the total number of samples within the data; and the **F1 score**, which is considered as the harmonic mean between precision and recall.

All LSTM classifiers performed good results (more than 80% on F1 Score). Many features are easily extracted (+ 99%) by the model because they have a simple structure. Elementary features like **IRE**, **IRI**, **CA**, and many others, have a non-complex structure, which facilitates the creation of the corresponding training dataset, and makes the LSTM classifier more able to correctly extract them from the source code. In contrarily, features like **AFL**, **CAFB**, **HGM**, etc... with more complex structure are more difficult to extract and need extra effort in constructing training data. In the second phase, the SPD classifier is trained to identify candidates for all SP roles. Features values reported from the LSTM classifier are fed to the SPD for the evaluation process. The reported results of different algorithms are illustrated in table 8. Training the SPD classifier by specifically creating data with relevant features makes it perfectly able to recognize candidate classes that are potentially playing a particular role in SP implementation. The obtained results presented in table 8 prove that the SPD is correctly trained to detect any implementation of the SP pattern. All ML algorithms perform good results (+ 90% of F1 Score), and the best result is reached by the **SVM** (+ 99% of precision, recall, and F1 Score).

5.2.2 Performance Comparison and Discussion

The DPB corpus is a repository created by the others of MARPLE (Fontana et al., 2012) to evaluate their approach. The MARPLE method is frequently used for comparative studies in DPD, also GEML is a recent work on DPD that achieves excellent results,

Table 7: Precision, Recall and F1 Score results of the LSTM.

Abb.	Precision%	Recall%	F1%
IRE	100	100	100
IRI	100	100	100
CA	99.5	98.3	98.89
GOD	98.2	98.6	98.39
AA	97.5	98.73	98.11
SR	97.23	99.12	98.16
ON	82.6	85.96	**84.24**
COA	92.56	96.45	94.46
HC	96.86	94.36	95.59
ILC	98.56	97.85	98.2
GAM	92.33	94.15	93.24
PSI	96.28	91.85	94
GSM	93.79	89.24	91.45
PST	92.6	90.45	91.51
INC	87.36	94.26	90.67
EC	95.69	90.25	92.89
RINIC	89.69	86.26	87.94
RINEC	92.32	87.69	**89.95**
CS	96.2	98.78	97.47
HGM	82.6	88.65	**85.51**
DC	96.32	91.63	93.91
RR	97.56	94.59	96.05
CNI	93.5	89.56	91.48
CII	90.23	95.36	92.72
DM	92.56	88.56	90.51
GMS	96.23	93.56	94.87
IGO	99.23	98.36	98.79
LNI	98.36	93.26	95.74
SCI	91.23	89.26	90.23
SB	98.69	95.26	96.94
AFL	82.36	80.63	**81.48**
CAFB	88.63	82.56	**85.48**
UR	95.36	92.63	93.97
EC	100	98.36	99.44
CSA	88.36	95.36	91.72
CFA	89.23	92.56	90.86
CSM	93.26	87.67	90.37
COP	96.29	98.24	97.25
MOP	92.65	89.19	90.88
BV	98.36	94.76	96.52
MSR	89.39	94.58	91.91
AMDP	93.2	84.26	**88.5**

Table 8: SPD Classifiers results.

SPD Classifiers	Measures		
	Pre.(%)	Re.(%)	F1(%)
Random Forest	**99.24**	96.7	97.95
Naive Bayes	95.85	88.5	92.02
KNN	92.3	87.95	90.7
SVM	99	**99.62**	**99.30**

so we decide to compare our proposed method with these two studies. We use the original results reported by authors (Zanoni et al., 2015) and (Barbudo et al., 2021) to compare it with the SPD results. Given that the results in their works are obtained based on the total DPB repository size, we also use all samples in this corpus (we label all existing patterns other than SP as none). As the general purpose configuration for MARPLE corresponds to Random Forest, we use the same algorithm for the test. Comparative results can be found in Table 9.

Table 9: Comparing SPD with MARPLE and GEML results.

Classifiers	SP Corpus: Labelled DPB	
	Accuracy(%)	F1-score (%)
MARPLE	93	90
GEML	95.61	94.11
SPD	**99.86**	**99.63**

As can be observed, and even if it is just a test for recovering only ST implementation, the SPD outperforms GEML with a percentage of improvements equal to 4.35%, 5.52% and MARPLE with 6.86%, 9.63 in terms of accuracy and F1 Score. The use of specific and more complete data for training the model makes the SPD better performed in recovering any instance. MARPLE and GEML use limited data to train their classifiers, and their ability to recover SP instances depends heavily on those present in the training dataset. Furthermore, the detailed analysis of the SP and the use of relevant features make the classifier more able to identify any implementation variants even if it is in a combination form.

6 CONCLUSION

In this work, we propose a novel approach to SP detection based on features and ML techniques. This work is the first to recover non solely the typical implementation but also the incorrect structure that inhibits the SP intent, and the implicit structure of this pattern. The goal of this detection is to improve the quality of source code, correct incoherent structure, and give the possibility to automatically inject the SP by discovering the corresponding context.

Based on a detailed analysis of the SP, we identify implementation variants of SI. Thereafter, we propose 9 features for their definition, and then we create specific data for each one, containing snippets of code. We added the newly proposed features to those proposed by (Nacef et al., 2022), and we use the same method to extract their values from the Java program. In the next step, and based on a different combination of feature values, we try to create data that contains the greatest number of implementations. The data is named SDD and used to train the SPD. The SPD is built based on different ML techniques. Evaluating the SPD on collected and labeled data from GitHub Java corpus named SED and DPB corpus, prove the performance of all used techniques and achieved +99% in terms of precision, recall, and F1 Score with SVM. The proposed approach outperforms MARPLE and GEML by + 4% in terms of accuracy and F1 Score..

In this work, we have focused on the SP with a detailed analysis, and we have taken the first step toward refactoring. In future work, we try to apply the same method to recover other DPs, and as the next step, we are going to attempt the injection of them.

REFERENCES

Barbudo, R., Ramírez, A., Servant, F., and Romero, J. R. (2021). GEML: A grammar-based evolutionary machine learning approach for design-pattern detection. *J. Syst. Softw.*, 175:110919.

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

Fontana, F. A., Caracciolo, A., and Zanoni, M. (2012). DPB: A benchmark for design pattern detection tools. In *16th European Conference on Software Maintenance and Reengineering, CSMR 2012, Szeged, Hungary, March 27-30, 2012*, pages 235–244. IEEE Computer Society.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. M. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition.

Lucia, A. D., Deufemia, V., Gravino, C., and Risi, M. (2009). Design pattern recovery through visual language parsing and source code analysis. *J. Syst. Softw.*, 82(7):1177–1193.

Mayvan, B. B., Rasoolzadegan, A., and Yazdi, Z. G. (2017). The state of the art on design patterns: A systematic mapping of the literature. *J. Syst. Softw.*, 125:93–118.

Nacef, A., Khalfallah, A., Bahroun, S., and Ben Ahmed, S. (2022). Defining and extracting singleton design pattern information from object-oriented software program. In *Advances in Computational Collective Intelligence*, pages 713–726, Cham. Springer International Publishing.

Nazar, N., Aleti, A., and Zheng, Y. (2022). Feature-based software design pattern detection. *J. Syst. Softw.*, 185:111179.

Rasool, G. and Mäder, P. (2011). Flexible design pattern detection based on feature types. In *26th IEEE/ACM*

International Conference on Automated Software Engineering, pages 243–252. IEEE Computer Society.

Stencel, K. and Wegrzynowicz, P. (2008). Implementation variants of the singleton design pattern. In *On the Move to Meaningful Internet Systems*, volume 5333 of *Lecture Notes in Computer Science*, pages 396–406. Springer.

Thaller, H., Linsbauer, L., and Egyed, A. (2019). Feature maps: A comprehensible software representation for design pattern detection. In *26th International Conference on Software Analysis, Evolution and Reengineering*, pages 207–217. IEEE.

Yu, D., Zhang, Y., and Chen, Z. (2015). A comprehensive approach to the recovery of design pattern instances based on sub-patterns and method signatures. *J. Syst. Softw.*, 103:1–16.

Zanoni, M., Fontana, F. A., and Stella, F. (2015). On applying machine learning techniques for design pattern detection. *J. Syst. Softw.*, 103:102–117.