Supervised Machine Learning for Recovering Implicit Implementation of Singleton Design Pattern

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: Singleton Implicit Implementation, Features, Datasets, LSTM, Supervised ML Algorithms.

Abstract: An implicit or indirect implementation of the Singleton design Pattern (SP) is a programming implementation whose purpose is to restrict the instantiation to a single object without actually using the SP. This structure may not be faulty or errant but can impact negatively the software quality especially if they are used in inappropriate contexts. To improve the quality of the source code, the injection of the SP is sometimes mandatory. In order to assuring that, a specific structure must be identified and automatically detected. However, due to their vague and abstract nature, they can be implemented in various ways, which are not conducive to automatic and accurate detection. This paper presents the first method dedicated to the automatic detection of Singleton Implicit Implementations (SII) based on supervised Machine Learning (ML) algorithms. In this work, we define the different variants of SII, then based on the detailed definition we propose relevant features and we create a dataset named **FTD** (Feature Train Data) according to the corresponding variant. Based on Long Short Term Memory (**LSTM**) models, trained by the **FTD** data we extract features values from Java program. Then we create another data named **SDTD** (Singleton Detector Train Data) containing feature combination values to train the ML classifier. We resolve the problem of automatic detection of SII with different ML algorithms like **KNN**, **SVM**, **Naive Bayes** and **Random Forest** for classification task. Based on different public Java corpus, we create and label a data named **SDED** (Singleton Detector Evaluating Data), this data is used for evaluating and choosing the appropriate **ML** model. The empirical results prove the performance of our technique to automatically detect the SII.

1 INTRODUCTION

The singleton pattern (SP) (Gamma et al., 1994) is a creational pattern. It represents an abstract solution to a commonly occurring software design problem. This pattern is useful when we need to ensure that only one object of a given class is instantiated.However, in certain cases the use of this pattern in a class that required it is absent, or oven accompanied by incomplete structure. In this case, the injection of the suitable pattern is mandatory and considered a complex form of refactoring that improve the source code quality, and code reuse.

The SP has a specific structure which usually consists of name, application, and implementation details. The use of this pattern makes possible the sharing of design information. This pattern solves a class of problems, so designers must know when to use it. In order to apply the SP, it is important to identify the type of problem resolved by it, and the existing structure in the source code that needed the use of this pattern.

In This paper, we focus on the definition and the automatic detection of SII. The definition of SII has many benefits like the reducing of errors in implementation, improving the quality of code and make code reuse.

There are many ways to implement SII and there is no formal-used structure. This variety of implementations makes the automatic detection of these structures a hard task. The existing methods dealing with pattern recovery convert source code to some intermediate representation. However, they show poor performance compared to methods used features. At the same time, the structural analysis doesn't allow a perfect recovery of the pattern intent in different representations. Semantic analysis is needed, and can perfectly improve the performance of the model in extracting needed information.

Therefore, to solve this problem, we propose to

Nacef, A., Bahroun, S., Khalfallah, A. and Ben Ahmed, S.
Supervised Machine Learning for Recovering Implicit Implementation of Singleton Design Pattern.
DOI: 10.5220/0011836100003464
In *Proceedings of the 18th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2023)*, pages 354-361
ISBN: 978-989-758-647-7; ISSN: 2184-4895
Copyright © 2023 by SCITEPRESS – Science and Technology Publications, Lda. Under CC license (CC BY-NC-ND 4.0)

define and extract features from the source code without using any intermediate representation. The relevant features are resulting from the structural and semantic analysis of the Java program. We are based on the performance of the **LSTM** to deal with source code sequence to extract feature values. We create a set of structured data named **FTD** corresponding to each feature for training **LSTM** classifiers. Based on the extracted features, we create a new structured data named **SDTD** to define the combination values defining each variant. This data is used to train a supervised ML classifier to detect an instance of SII.

In this work, we analyze a Java program using **LSTM** classifiers, then we use 4 different ML classifiers (**KNN**, **SVM**, **Naive Bayes**, and **Random Forest**) for the automatic detection of SII.

The main contributions of this paper consist on:

- Defining and analyzing the SII.
- Proposing 21 relevant features for identifying each variant.
- Constructing 3 datasets; **FTD** whose total size is, 15000 used for training **LSTM** classifier. **SDTD** containing 7000 samples used for training the Singleton Implicit Detector (**SID**). **SDED** containing 200 java files collected from the java public corpus and used for the evaluation process.
- Evaluating the created classifiers and comparing the different **ML** results in terms of precision, recall, and F1-Score.

The rest of this paper is organized as follows: In Section 2 we establish an overview of related works. In Section 3, we define and analyze the SII, and we propose a set of relevant features. Section 4 presents the proposed approach process and techniques used. In Section 5 we represent the different created data and we discuss the empirical results. In section 6, we show the conclusion and future works.

2 RELATED WORKS

The identification of SII has not been extensively discussed in the literature. For this reason, we provide an overview of relevant empirical studies.

The effect of DP on selected quality characteristics and improve software quality has been the subject of several studies. However, most DPs recognition approaches focus on the definition of a typical structure.

DP recognition methods can be classified according to different criteria and aspects. In this paper, we categorize detection approaches according to the detection methods used and the analysis styles. The Detection methods can be divided into four major categories: Database query methods, metric-based methods, UML structures, diagrams, etc. Matrix-based methods and others.

(Rasool et al., 2010), (Stencel and Wegrzynowicz, 2008) and other works use database query approaches to recover pattern. In general, this kind of approach transforms the source code into an intermediate representation, such as AST, ASG, XMI, etc. Then they use SQL queries to extract related pattern information from the source code representation. However, the use of database queries cannot recover the instances of behavioral patterns, and queries are limited to the available information existing in the intermediate representations.

Techniques proposed by (von Detten and Becker, 2011) and (Kim and Boldyreff, 2000) use program-related metrics method (e.g., aggregations, generalizations, associations, and interface hierarchies) to recover pattern. (von Detten and Becker, 2011) combine both cluster-based and pattern-based reverse engineering methods. In their approach, they show that the presence of bad smells in software system code can falsify metric-based clustering results. An advanced method, proposed by (Satoru Uchiyama, 2011) combine software metrics and machine learning to identify candidates for the roles that compose.

Other approaches transform structural and behavioral information into UML structure, graph, or matrix. (Fontana and Zanoni, 2011) exploit a combination of graph matching and ML techniques. (Yu et al., 2015) transform GoF patterns and source code into graphs with classes as nodes and relationships as edges. Moreover, to obtain final instances, the behavioral characteristics of method invocations are compared with the predefined method signature template of GoF patterns. In this approach, a structural feature model to represent GoF DPs is introduced. However, structural features cannot capture the pattern intent. Most of these methods show good precision and recall, but they cannot handle implementation variants of DP.

Another method proposed by (Chihada et al., 2015) is based on learning from DPs information for the recovering process. The DP recognizing is considered as a learning problem. Similar recent work proposed by (Hussain et al., 2018) treats the problem of DPs recovering as text categorization. Another work proposed by (Thaller et al., 2019) uses the neural networks to build a feature map for pattern instances. Recently, (Nazar et al., 2022) select a set of relevant feature codes and combine it with ML to automatically train a DP detector.

We can summarize the difference between our work and the previous works as:

- Similar to other approachs (Chihada et al., 2015), and (Hussain et al., 2018), we consider the pattern recovering as learning problem. However, we use features for reducing the learning space and focus on relevant information to improve accuracy.
- Similar to (Thaller et al., 2019) and (Nazar et al., 2022), we use ML and features. But, the use of specific and more various data makes the learning process more efficient.
- (Satoru Uchiyama, 2011) in his approach proposes a method to recover a variant of the SP. One of the recovered variants represent a specific implementation using Boolean variable (listing 1). This implementation is considered in our work as an SII.

All previous work aims to improve the quality of source code. But we are the first to define the SII and its automatic detection.

3 SII: DEFINITION AND REPORTED VARIANTS

In this section, we will give a concrete definition of SII with their reported variants. According to the implementation aspect of each variant, we propose a set of features for their automatic detection.

3.1 Definition of SII

The SP intent is to ensure that in the same application that can be only one created instance of a class at any time. To implement that, there are many different structures used depending on the context and the usefulness. These structures are almost based on common characteristics like ensuring that only one instance exists, providing a global private static instance to that access with private constructors, and providing a static method that returns a reference to this instance. However, in some cases, the use of this specific structure (SP) in the class that required it, is absent. The developer creates a class in his specific way corresponding to his knowledge and can prevent the instantiation by a specific structure. We named this specific implementation, used to prevent instantiation only to one instance, as *Singleton Implicit Implementation*. There are many ways to implement that. Based on a detailed analysis of different public projects, and developers' choices to avoid the use of SP, we identify 4 categories of SII:

- **Category 1: Non-Global Access Point**

The SP ensures instance reuse. In order to preserve the instance for future reuse, global access to the instance must be provided. However, away from SP's typical implementation, providing public and global access to the instance may not always be preserved. Single instantiation can be ensured by restricting access to certain areas of code or even making it private to a single class. There are several ways to achieve this, like the use of a Boolean variable (listing 1).

Listing 1: Example of SII using Boolean variable as feature.

```
public class SinglC
{ public static boolean test = false;
  public SinglC() throws Exeption
  {if (!test) {test = true;}
  else {throw new Exeption ("...");}}
  public static SinglC getInst() throws
      Exeption {...}
  ...
```

- **Category 2: SII with Static Properties.**
  - **Static Data.** Monostate described in (Martin, 2002) represents a kind of implementation used to create one instance of a class. This pattern includes making all data static except getters and setters.
  - **Fully Static Class.** Static classes are a solution for storing a single instance of data to be accessed globally in an application.
- **Category 3: Providing Convenient Access to an Instance**

  It consists of giving easy access to objects that need to be used in many different places. The general rule is that we want the bounds of the variables to be as tight as possible and still get the job done. There are many ways to access objects:
  - **Object as Parameter in Function.** The easiest and often best solution is to simply pass the desired object as a parameter to the function that needs it. For example, consider a function for rendering objects. Rendering requires access to an object that represents the graphics device and contains the rendering state. It is common to pass it to all render functions, usually as a parameter called context.
  - **Base Class.** The subclass sandbox contains an abstract sandbox method. Each derived Sandbox subclass implements the Sandbox methods with the provided actions. Sandbox classes work well in the case of minimizing the coupling between the derived classes and the rest of the program.
  - **Service Locator.** Consist to define a class whose sole purpose is to provide global access

to objects. This generic pattern is called Service Locator. It defines an abstract interface for a set of operations in which a specific service provider implements this interface.

- **Category 4: Other Implicit Implementations**
  It is worth noting that these implementations can be combined and other implementations to restrict instantiation, preserve the state of the instance, or share services can exist.

  The use of an enum with final fields can be considered a multi-thread safe solution. Also, the control instantiation itself (making a strict condition to limit the number of instances) can be considered a naif solution to SII. More than that, we consider an incomplete structure of Singleton's typical implementation with certain conditions (have only one used reference and only one method to create an instance) as SII which should be detected to be corrected

Listing 2: Example of Singleton Implicit Implementation using Enum class.

```
public enum SoundSystem {
    INSTANCE("Hello", "World");
    private final Object field1;
    private final Object field2;
    private SoundSystem(Object field1, Object
        field2) {
        this.field1 = field1;
        this.field2 = field2;
    }
    ...
```

3.2 Quality Impact and Trade-offs

SII aims to create a single instance of the class. However, this goal can not always be achieved, and the implementation doesn't play a good solution. Unique instantiation is not always guaranteed and therefore can cause unnecessary system overhead and inconsistent behavior.

Classes based on the implementation that doesn't provide global access, allow anyone to build it, but it will fail when we try to build multiple instances. The downside of this implementation is that the checks to prevent multiple instantiations are only performed at runtime. In contrast, due to the nature of the class structure, the SP guarantees that there is only one instance at compile time.

The benefit of using static classes is that the compiler ensures that instance methods are not accidentally added. The compiler guarantees that no instance of the class can be created. However, the downside of using static classes is that after decorating a class with the static keyword, we can never change how it behaves and the polymorphic option is restricted.

The Service Locator passes objects to the code that it needs, instead of using a global mechanism to give some code access to it. That's dead simple, and it makes the coupling completely obvious. But, some objects manually are unnecessary, or actively make the code harder to read.So, implementing a service locator instead of SP can cause scalability issues in high-concurrency environments.

SII aims to create a single instance of the class. However, sometimes it doesn't represent the perfect solution and can provoke many problems, as we have mentioned. On another side, the use of the typical implementation of SP offers a complete object-oriented solution. Only one instance of a class is required at any given time and maintains one state. In this case, the injection of a typical implementation is mandatory.

3.3 Proposed Features

To be able to detect the different variants of SII we are based on a subset of highlighted features. According to variant definition and structure, we have proposed 21 features as represented in Table 1 with the abbreviation and the description.

For all variants, common properties must be verified like; the global object declaration, the accessibility and static property of the class attribute, the accessibility of the constructor, the global and the static declaration of the access method and finally the condition to limit the number of instances; whether by using Boolean variables, numerical variables counting the number of instances, or by testing the value of the object itself.

For the implementations based on static aspects we must check that the class has many static attributes and static methods, the class has a static inner class and the static properties of getter and setter methods. For variants that provide the object reference to the block it needs, check the use of the reference as a parameter in the constructor and/or in one or more methods.

There are several ways to implement a Service Locator, but the most commonly used are: a static service locator, which uses attributes for each service to store object references, and a dynamic method that contains java.util.Map of all references of service. This can be dynamically expanded to support new services. For the base class variant (Base Class), many mechanisms can be used. The simplest solution is to pass the object reference as a parameter in the base class constructor. Another solution is to divide the initialization into two stages. The manufacturers Do not take any parameters and simply create objects. Then

Table 1: Proposed Features

No.	Abb.	Feature
1	IRE	Inheritance relationship (extends)
2	IRI	Inheritance relationship (implements)
3	EC	Enum Class
4	GD	Global class attribute declaration
5	AA	Class attribute accessibility
6	SA	Static class attribute
7	OA	Have only one class attribute
8	CA	Constructor accessibility
9	GSAM	Global Static Access Method
10	GSSM	Global Static Setter Method
11	HOGM	Have one method to generate instance
12	SIC	Use Static Inner Class
13	CSA	Class with Static attributes
14	CFA	Class with Final attributes
15	CSM	Class with Static Methods
16	COP	Constructor with Object Parameter
17	MOP	Method with Object Parameter
18	CI	Control Instantiation
19	BV	Using Boolean Variable to Instantiate
20	MSR	Create Map for serves references
21	AMDP	abstract method with data parameter

call a method that passes all the necessary data and is defined directly in the base class. Or simply make the state of the basic class private and static. Finally, in the enum class-based implementation, it is necessary to verify that a class type is declared enum, and that all variables are declared as final static. The number of class objects declared, and the number of methods that generate the class instance, play a large role in checking the consistency between the source code and the pattern's intent. Therefore, taking these conditions into account is critical to prevent the model from generating false predictions (predicting a false positive as shown in Program 1.3 and 1.4).

3.4 Features Preserving the SP Intent

The intent of the SP is to create only one instance of a class. Even if the SII doesn't represent the complete or the correct solution to resolve the problem well, an implementation with features that prohibit the pattern intent is not considered an SII. The overarching features we use to define SII consistent with the intent of the pattern are "have only one class attribute" and "have only one generating instance". For example, the implementation represented in listing 3 is not considered as SII because it's incoherent with singleton pattern.

Listing 3: Example of incorrect SP.

```
nbins=0;
public static SinglC2 getIns() {
        if (nbins == 0) {
        instance = new SinglC2();
        nbins +=1;}
        return instance; }
    public static SinglC2 getInst2 ()
          { return new SinglC2() ;}}
```

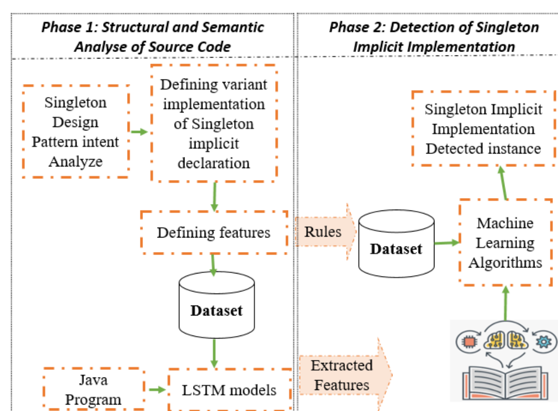

Figure 1: Proposed Approach Process.

4 DETECTION PROCESS

In this section, we will present the technical process of our proposed approach. For recovering the SII, the proposed process will be divided into two phases as shown in figure 1.

4.1 Phase1: Source Code Analysis

In this phase, we started by analyzing the intention of the SP. Based on this analysis we have determined as many as possible structures that assure this intent. These SII as we have named are subsequently studied for the purpose to extract a set of common characteristics between each variant. At the end of this analysis, we have identified 21 features and created 21 datasets **FTD** corresponding to each one. Every data is composed of a variety of implementations that can serve to good learning of the classifier.

To be able to extract feature value, we apply a structural and semantic analysis of the source code by the **LSTM** model.

We have to build 21 **LSTM** classifiers for extracting values for each feature. **LSTM** classifiers are trained by the corresponding feature's data. After analyzing the source code and extracting features value, we pass to the second phase whose purpose is to recover the SII instances.

4.2 Phase 2

The analysis of a different variant of SII makes the recovery of each one more easy. In fact, based on the previous analysis, we have created a dataset containing different features combination.

By creating structured data named **SDTD**, we have trained a set of ML classifiers. We have selected

Table 2: Data details.

Data	Size	Application
FTD	15000	training **LSTM** classifiers
SDTD	5000	training the **SID** classifier
SDED	200	Evaluate **LSTMs** and **SID**

4 ML techniques the most used on DPs and CSs detection (**Naïve Bayes** (NB), **KNN**, **SVM**, **Random Forest** (RF) and compare them based on the obtained results.

5 EVALUATION SETUP

In this section, we will present the used datasets for training and testing the models, the criteria to evaluate them, and different achieved results.

5.1 Data

We have created and labeled manually three different structured data; **FTD**, **SDTD** and **SDED** which the corresponding details are presented in table 2.

FTD. is data created from a variety of snippets of code whose total size is 15000 samples, used for extracting feature's value from Java source code. For each feature, corresponding data is created. The data contains the most needed samples that make true positive prediction and reduce the false positive (example of listing 4 5, 6 and 7) rate. Based on obtained results in each simulation, we try to correctly modify the data with the goal to obtain the most appropriate data for the right model training.

Listing 4: Example of correct control instantiation feature.

```
private static C1 instance1;
   private C1 instance1 = null ;
   private C1(){  }
   public static C1 getInstance(){
      if (instance1 == null)
         {instance1 = new C1 ();}
          return instance1;}
```

Listing 5: Example of incorrect Control instantiation feature

```
public static C1 getInstance()
  { if (C2.instance2 == null)
    { instance1 = new C1 (); }
    return instance1; }
```

Listing 6: Example of correct Boolean variable feature.

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

Table 3: True and false SII based on class Enum.

Class	Features				
	EC	**AA**	**CA**	**COP**	**CFA**
True	True	Final/ Private final	private	True	True
False	True	Final/ Private final	private	True	**False**

Table 4: True and false SII based on static class.

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

Listing 7: Example of incorrect Boolean variable feature.

```
private static C3 instance1;
private Boolean noInstance = true;
public static C3 getInstance(){
 if (noInstance)
{System.out.println("No instance");}
    instance1 = new C1();
    return instance1;}
```

SDTD. is structured data containing 5000 samples used to train the **SID**. The data contain feature combination values as input (whether Boolean and categorical types) and two categorical targets (True-SII / False-SII). To ensure fair classification, we make an equitable number of samples (2500 samples for each class). The False-SII class should contain the combination of features that is close to SII, but doesn't make an SII candidate. The goal in there is to reduce the number of false-positive and make perfect model learning. Table 3,4 and 5 shows an extract from the **SDTD**.

For a class enum, we must verify that a class type is an enum (EC) and that all variables are declared as final (CFA), otherwise, it is considered a false candidate. Static classes contain static attributes and methods (COP, CSA) and can use any type of access modifier (private, protected, public, or default) like any other static member. A static class is represented in the form of an inner class or a nested class (SIC) otherwise it is considered a faulty implementation. For SII implemented using control instantiation; independent of class attribute accessibility and constructor accessibility, a class must maintain only one generated instance (HOGM) in which there exist a block for controlling the instantiation (CI). If any of these conditions is violated, the implementation is considered as False SII.

SDED. is a structured labeled data used for evaluat-

Table 5: True and false SII based on control instantiation.

Class	Features				
	GSAM	**OA**	**HOGM**	**CI**	**MOP**
True	True	True	True	True	True
False	True	True	**False**	**False**	True

ing the **ML** models whether in phase 1 or 2. We have collected 200 Java files from a variety of GitHub public Java projects [1]. Based on the analysis of these projects, we manually select 150 files with SII and 50 with none.

5.2 The Evaluation Protocol

Precision, recall, and F1 score are standard measures for statistical evaluation of classifier effectiveness. The precision rate indicates the fraction of positive precision which was actually correct. However, the recall represents the proportion of actual positives which were identified correctly. The F1 score is a harmonic mean of precision and recall into a single number.

5.3 Experimental results

Regarding our approach being the first to define and detect SII, we haven't found closet works for comparing results. That's why, in this subsection, we restrict the comparison phase to only one relevant approach (Thaller et al., 2019).

5.3.1 Obtained Results

- **LSTM Results**

We have built 21 **LSTM** classifiers and trained them by the corresponding data to extract each feature value. The different results presented in table 6 prove that the **LSTM** is suitable to extract features value from Java program, especially when they have an elementary and simple structure (100% of F1-Sore in **RE**, **RI** and **EC**). However, the model is less efficient with complex features like **OA**, **HOGM**, and **CSM** (less than 80% of F1-Score). To deal with that, we can replace the complex features with more elementary ones.

- **SID Classifier Results**

The **SID** is a classifier using supervised learning with structured data. To select the appropriate model, we have tested a set of ML models most used on DP and CSs extraction (**KNN**, **SVM**, **RF** and **NB**). The different obtained results are represented in table 7. All **ML** models perform a good results thanks to the specific created dataset **SDTD**. The training data make any model able to automatically recover SII with higher accuracy. Except for **NB**, all models perform very good and close results (more than 96% in both precisions, recall, and F1-Score). **RF** contrarily to **NB** is the butter used model, it achieves more than 99% in different standard measures.

Table 6: LSTMs results applied on **SDED**.

Features	Precision (%)	Recall (%)	F1(%)
IRE	100	100	100
IRI	100	100	100
CSA	96.1	87.49	91.59
EC	100	100	100
CFA	94.13	90.2	92.12
GD	100	95.5	97.69
CSM	74.81	81.26	**77.9**
AA	92.6	95.12	93.84
COP	88.46	95.4	91.79
SA	94.8	91.74	93.24
MOP	89.7	87.63	88.65
OA	83.4	70.51	**76.41**
CI	92.36	97.8	95
CA	87.89	89.3	88.58
BV	96.45	91.76	94.04
GSAM	88.8	82.46	85.51
MSR	93.5	89.12	91.25
GSSM	86.23	90.21	88.17
AMDP	87.45	89.56	88.49
HOGM	70.9	77.8	**74.18**
SIC	90.56	94.78	92.62

Table 7: Comparison of SID Classifiers results.

SID	Measures		
	Precision (%)	Recall (%)	F1 (%)
RF	**99.49**	**99**	**99.23**
NB	87.90	89.95	87.63
KNN	97.77	98.29	98.01
SVM	96.48	97	96.72

5.3.2 Comparison with State-of-the-Art Approach

Though our approach is based on features and ML, we have chosen Feature Maps (FM) (Thaller et al., 2019) as a similar approach for comparison. The data used for the evaluation contain 100 Java files and is constructed from Java samples existing in DPB corpus (Fontana and Zanoni, 2011) (typical implementation) and samples from **SDED** containing base class and service locator implementation (SII). We have replicated the FM approach and evaluated it on recently created data based on the **RF** algorithm. The obtained results are detailed in table 8.

Table 8 show the difference in result between FM and SID in terms of the correct and incorrect number of predicted SP instances. Feature Maps is created to recover typical implementations, so it has the abil-

[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

Table 8: Comparing SID results with Feature Maps results.

Methods	STI		SII	
	Correct	Incorrect	Correct	Incorrect
FM	**49**	9	24	18
SID	47	11	**40**	2
Total	58		42	

ity to recover correct instances. However, similar implementations (implicit) are not fully trained by the model, so it fails to correctly predict them. Feature Maps recover a quiet number of SII as true positive SP whereas these implementations can negatively affect the source code quality. In contrast, and even if the structures were very similar, the **SID** can distinguish between the two implementations (47 correct instances from 58 SP and 40 correct instances from 42 SII).

6 CONCLUSION

In this paper, we have proposed a novel approach to define and automatically detect the SII. Based on the analysis of the SP intent, we define structure in that we need to inject the SP. By defining and analyzing different variants, we propose 21 relevant features that can make the definition of each implementation.

For extracting feature values from the Java program, we propose to use the **LSTM** models for syntactical and semantic analysis. To train the **LSTM**, we have created and labeled a data named **FTD** which is composed of 15000 snippets of code. For the automatic detection of SII, we create a classifier named **SID** which uses different ML algorithms. The **SID** is trained by a created and labeled data named **SDTD**. The data is composed of feature combination values according to the most existing implementation. The global size of the data is 5000 samples.

To evaluate the created models, we collect 200 Java files from different public projects. We manually label the data used for the evaluation process. We have selected 4 ML models for the **SID** and chosen the perfect one according to their performance. The empirical results prove that the proposed technique can correctly recover any SII with higher accuracy (more than 99% of precision) and outperforms the relevant approach in distinguishing between both types of SP.

In future work, we try to create more elementary features to improve the extraction of some complex ones. The purpose of automatic detection of the SII is to improve the source code quality by injecting the SP in the appropriate context. So, in future work, we will pass on the realization of these complex types of refactoring using ML. Beginner with the SP we try to continue the extraction of implicit implementation with other DPs.

REFERENCES

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.*

Fontana, F. A. and Zanoni, M. (2011). A tool for design pattern detection and software architecture reconstruction. *Inf. Sci.*, 181(7):1306–1324.

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

Hussain, S., Keung, J., Khan, A. A., Ahmad, A., Cuomo, S., Piccialli, F., Jeon, G., and Akhunzada, A. (2018). Implications of deep learning for the automation of design patterns organization. *J. Parallel Distributed Comput.*, pages 256–266.

Kim, H. and Boldyreff, C. (2000). A method to recover design patterns using software product metrics. In *Software Reuse: Advances in Software Reusability, 6th International Conerence*, Lecture Notes in Computer Science. Springer.

Martin, R. C. (2002). *Agile Software Development, Principles, Patterns, and Practices*. Prentice-Hall.

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

Rasool, G., Philippow, I., and Mäder, P. (2010). Design pattern recovery based on annotations. *Adv. Eng. Softw.*, 41(4):519–526.

Satoru Uchiyama, Atsuto Kubo, H. W. Y. F. (2011). 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.

Stencel, K. and Wegrzynowicz, P. (2008). Detection of diverse design pattern variants. In *15th Asia-Pacific Software Engineering Conference*, pages 25–32. IEEE Computer Society.

Thaller, H., Linsbauer, L., and Egyed, A. (2019). Feature maps: A comprehensible software representation for design pattern detection. In *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27*.

von Detten, M. and Becker, S. (2011). Combining clustering and pattern detection for the reengineering of component-based software systems. In *7th International Conference on the Quality of Software Architectures*, pages 23–32. ACM.

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.*