IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received March 30, 2021, accepted May 3, 2021, date of publication May 10, 2021, date of current version May 20, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3078724

Decision tree-based Design Defects Detection

MOHAMED MADDEH[1], SARRA AYOUNI[2], SULTAN ALYAHYA[1], AND FAHIMA HAJJEJ[2]
[1]Department of Information Systems, King Saud University, Riyadh 11451, Saudi Arabia
[2]Department of Information Systems, College of Computer and Information Sciences, Princess Nourah bint Abdulrahman University, Riyadh 11564, Saudi Arabia

Corresponding author: Sarra Ayouni (saayouni@pnu.edu.sa)

This research was funded by the Deanship of Scientific Research at Princess Nourah bint Abdulrahman University through the Fast-track Research Funding Program.

ABSTRACT Design defects affect project quality and hinder development and maintenance. Consequently, experts need to minimize these defects in software systems. A promising approach is to apply the concepts of refactoring at higher level of abstraction based on UML diagrams instead of code level. Unfortunately, we find in literature many defects that are described textually and there is no consensus on how to decide if a particular design violates model quality. Defects could be quantified as metrics based rules that represent a combination of software metrics. However, it is difficult to find manually the best threshold values for these metrics. In this paper, we propose a new approach to identify design defects at the model level using the ID3 decision tree algorithm. We aim to create a decision tree for each defect. We experimented our approach on four design defects: The Blob, Data class, Lazy class and Feature Envy defect, using 15 Object-Oriented metrics. The rules generated using decision tree give a very promising detection results for the four open source projects tested in this paper. In Lucene 1.4 project, we found that the precision is 67% for a recall of 100%. In general, the accuracy varies from 49%, reaching for Lucene 1.4 project 80%.

INDEX TERMS Anti-patterns, bad smells, decision tree, model refactoring, object oriented metrics.

I. INTRODUCTION

It is difficult and very expensive to identify and correct software defects especially those corresponding to large projects. The majority of researches dealing with this issue focus on code defects identification by analyzing the software source code [1], [2]. These code defects are usually, the consequence of design defects that was propagated to the code. Detecting design defects at earlier stage in the system development life cycle is a promising way to improve the process of software maintenance [3]–[6]. This will reduce the time, effort and maintenance cost.

Design defects refers to design solutions that negatively impact the development of a software like *anti-patterns* [7] and *bad smells* [8], [9]. The purpose of *anti-patterns* is to avoid or fix errors before writing any code by documenting common bad practices in software design. *Bad smells* represent symptoms of poor design and implementation choices. There is a gap between the textual defects description and the defect identification. Each designer can give his own interpretation for the same defect. For example, for a well-defined defect like *Feature Envy* referring to a method that belongs to other class, instead of the class in which it is defined. It is confusing to decide which classes are *Feature Envy* candidates because this depends on the designer's interpretation. In fact, the detection of such defect needs information like "how many times does the message communicate with a given class?". This information can be considered high in a given context and could be considered medium in another. Furthermore, a "Log" class that maintains the archive of events, used by a many classes, is a common and acceptable practice. However, referring to the definition, it can be considered as a class with abnormally high coupling. Thus, we find several techniques to detect design defects [10]. In general, authors use rules in the form of metric/threshold combinations to evaluate and identify design defects. Some studies propose to use rules-based methods that are manually identified [11]–[14]. Others propose algorithms to generate these rules [15]–[17]. Both approaches face two major difficulties. The first one is the large number of possible metrics combinations needed to find the best suitable rule. The second issue is how to find the best threshold for each metric.

In this article, we propose a method to find the best metrics' combination with the most appropriate threshold. We focus on how to define detection rules in presence of quantitative information and then how to provide a standard

The associate editor coordinating the review of this manuscript and approving it for publication was Imran Sarwar Bajwa.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

representation of defects' specification. In the context of our research, we use decision tree technique to formalize design defects. It is a predictive model, which maps observations about defects from the set of examples to detection rules. More specifically, we address the following questions:

(1) How to choose the best detection rules?

(2) How to find the best metrics threshold?

The remainder of the paper is structured as follows. Section 2 presents the related works. Section 3 gives an overview on design defects and object oriented metrics. Section 4 presents the details of our application of decision tree algorithm to the problem of detecting design defects at the model level. Section 5 validates the proposed approach. Sections 6 and 7 outline the discussion and the conclusion.

II. RELATED WORKS

In this section, we present the most relevant studies on model refactoring. We exclude works focusing on design defect detection in the code level [18]–[22]. A few existing works are based on UML model to predict defects before the implementation phase. These works could be classified into two broad categories: manual rules identification and automated rules generation. In [23], the authors used Software Architecture Analysis Tool to calculate metrics for UML diagrams. These metrics are then used to identify the flaws or the *anti-patterns*. The authors represented the structure and the behavior of the system using the class diagram and the state chart diagram, respectively. After that, they examined the metrics to identify the centralized control structure and refactor them into one that employs more delegation. This work is limited to centralized control structure. It is also based on a manual detection that is very difficult to generalize for large projects. In our current study, we overcome this problem and we propose generic detection process that can be applied for any design defect. Marinescu in [12], [24], [25] defined a list of metrics rules to detect design flaws. The limitation of his approach is that it is difficulty to identify manually the metrics thresholds. In [26], the authors proposed an automated based-approach to detect model-refactoring opportunities related to various types of design defects. Their approach uses genetic programming based on the similarity/distance measures between the studied system and a set of defect examples without defining rules for the detection. In [27], Maddeh *et al.* proposed a framework (M-RAFACTOR) to detect and correct design defects based on object oriented metrics. In this study, authors generate manually the detection rules. In [28], the authors generated rules to detect design defects at the model level using gradual rules and proposed a measurement of tendency instead of metrics thresholds. In [29], Mohamed *et al.* proposed a multi-view integrated approach to model-driven refactoring using UML models. They propose an integrated metamodel based on one model selected from each UML view, at the metamodel level. In [30], Cortellessa *et al.* proposed a model-driven solution to help designers improve the availability of their software through refactoring opportunities. They tried to improve the software availability of the system through model refactoring and model transformation. Tanhaei [31] presented a model refactoring by transforming Software Architecture (SA) to a pivot-model on which he applied the refactoring process. However, there is no automated detection of refactoring opportunities since it is based on stakeholders refactoring goals and recommendations. In [32], the authors use the machine learning approach to refactor a UML class diagram suffering from a functional decomposition problem. This solution is not generic; this work focuses on one specific problem and tries to transform the UML class diagram into a new version more compliant with the Object Oriented paradigm. In [33] Alshayeb *et al.* proposed an approach for improving sequence diagrams' security through the application of refactoring based on *bad smells* defect. The detection of these *bad smells* is based on Genetic Algorithms. In this work the authors focused on one specific refactoring problem; the security in a sequence diagram.

We note that the majority of studies related to the detection of design defects are based on the code analysis. It is more difficult to predict defects before the implementation stage, because of the lack of information comparing to the code. Some existing works defines detection rules manually; it is then difficult to find the best thresholds of metrics. Some other works are dealing with a specific refactoring or tendency evaluation.

III. DESIGN DEFECTS AND OBJECT ORIENTED METRICS

In this study, we aim to define a predictive model refactoring based on the detection of design defects, (i.e. design anomalies); namely *bad smells* and *anti-patterns*. Fowler *et al.* proposed the *bad smells*. They defined a set of 22 symptoms of common defects. Brown *et al.* introduced the *anti-patterns*.

In this study, we consider four different design defects:

The *Blob anti-pattern* also called *God class* [34]: this defect corresponds to a big controller class that uses and accesses data stored in other surrounded classes. A big class encompasses many attributes and methods with a low cohesion.

Data class bad smell: this defect corresponds to classes containing attributes without methods operating on them. They might belong to another class.

Lazy class bad smell: this defect occurs when a class is used in few number of scenario (i.e., represented by sequence diagrams). Such classes should be merged or deleted reducing the project complexity.

Feature Envy bad smell: this defect occurs when a method defined in a class C1 makes extensive use of attributes or/and methods defined in another class C2. Such method should move to the class C2.

Each defect is evaluated using a combination of metrics. Object oriented metrics are used to measure the software quality and predict design defects occurrence. In fact, we try to detect defects at an earlier stage of the software development process.

Algorithm ID3 for defect detection
function ID3 (R: set of metrics, C: target defect, S: set of examples) **return** a decision tree;
begin
1: **If** S is empty, **return** Failure;
2: **If** S consists of records with the same value for the target defect, **return** a single node with that value;
3: **If** R is empty, **return** a single node with the most frequent value of the target defect that are found records of S;
4: **Let** D be the metric with minimal entropy (D,S) among metrics in R;
5: **Let** {dj| j=1,2,…m} be the values of metric D;
6: **Let** {Sj| j=1,2,... m) be the subsets of S consisting respectively of records with value dj for the metric D;
7: **return** a tree with root labeled D and arcs labeled d1,d2.... dm
8: ID3(R-{D}, C, S1), ID3(R-{D}, C, S2),...., ID3(R-{D}, C, Sm);
end ID3;

FIGURE 1. The ID3 algorithm for defect detection.

TABLE 1. Object oriented metrics.

Metric	Name	Description
NC	Number of Classes	Number of classes in the project
PS	Package Size	Number of classes in the package
NOA	Number Of Attributes	Number of attributes in the class
NOM	Number Of Methods	Number of methods in the class
NOD	Number of Descendent	Number of classes descendent (inheritance)
NODD	Number Of Direct Descendent	Number of direct class descendent (direct inheritance)
NMSC	Number of Messages in the Same Class	Number of messages send from a class to itself (internal-messages)
NOC	Number of messages sent for Other Classes	Number of messages sent for other classes
NCC	Number of Connected Classes	Number of classes connected to the measured class
ATFD	Access To Foreigner Data	Number of classes connected with the measured class
NOP	Number Of Parameters	Number of parameters in a method
NIC	Number of Interconnected Classes	Number of classes affected by the measured method
CM	Changing Methods	Number of methods affected by the measured method
NOPM	Number Of Packages in the Model	Number of packages in the model
PUC	Package Used Classes	Number of classes used outside the measured package

This work is inspired from metrics defined in literature [35]–[37]. In our experimentation, we use fifteen metrics useful for software measurement and design flaw detection, as shown in Table 1. The measurement of metrics is based on static and dynamic diagrams. For static metrics, we use class diagram and for behavioral metrics, we use the sequence diagrams.

IV. DECISION TREE ALGORITHM FOR DETECTING DESIGN DEFECTS

A decision tree is a graphical model of potential solutions to a decision under conditional control statements. It is a method of supervised machine learning [38]. It aims to classify a set of data features into homogeneous groups in terms of the predicted variable. It takes as input a set of classified data, and its outputs is a tree that resembles to an orientation diagram. It represents a course of action that gives a possible decision represented by the tree branches. Each solution is mutually exclusive. In this work, we base our results on training examples. In fact, to predict defects we study defects occurred in older projects.

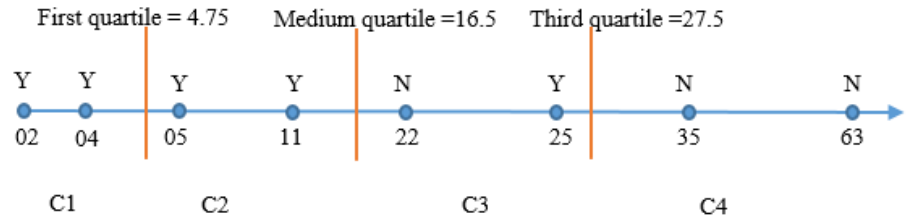
To create the design defects decision tree, we use the ID3 algorithm [39]. As presented in Figure 1, the ID3 algorithm is based on the following:

- Each non-leaf node of the decision tree corresponds to an input metric, and each arc corresponds to a possible value of that metric. A leaf node corresponds to the expected value (i.e.; Defect, No defect) of the output attribute where the input attributes are described by the path from the root node to that leaf node.
- For one problem, we can create many decision trees depending on metrics and nodes. In ID3 algorithm, the construction of the tree starts by the most informative metrics.
- Metrics information are evaluated using the Shannon Entropy described in section C.

Once the ID3 tree is constructed, designer can filter the extracted rules by fixing the heuristic N representing the minimal number of metrics in rules detection. Depending on the project and the detection strategy, the designer specifies the value of N. This value avoids the selection of rules containing a few/huge number of metrics. In fact, a small value, leads to an over detection and a high number of false positive, the number of detected defects exceeds the real existent defects. However, a high N value gives the opposite result, a high number of false negative. We detect a very small part of existent defects. The maximum value of N is the depth of the ID3 tree.

A. ADAPTATION OF DECISION TREE TO DESIGN DEFECT DETECTION

The expected rules we aim to extract using ID3 are of the form:



	<=4.75	>4.75	Total
Defect Y	2	3	5
Defect N	0	3	3
Total	2	6	8

Contingency table for cut point 4.75

	<=16.5	>16.5	Total
Defect Y	4	1	5
Defect N	0	3	3
Total	4	4	8

Contingency table for cut point 16.5

	<=27.5	>27.5	Total
Defect Y	5	0	5
Defect N	1	2	3
Total	6	2	8

Contingency table for cut point 27.5

FIGURE 2. Cut point example.

For a defect D: "***IF*** *metric*$_1$ *is higher/lower than threshold* $_1$ ***AND*** *metric* $_2$ *higher/lower than threshold* $_2$ **....** ***AND*** *metric*$_n$ *higher/lower than threshold*$_n$ ***THEN*** *defect D is suspected*".

ID3 is a recursive algorithm; lines 1-3 in Figure 1 encode the termination criteria. The algorithm stops when the set of examples is empty or when all metrics are classified. For an intermediate node, ID3 measures each metric gain based on Shannon entropy, in the line 4. ID3 chooses the metrics with the best gain as a root node; this metric is deleted from the set of metrics. For each value of the root metrics, ID3 is called with the rest of metrics and the new set of example related to the arc values, lines 5-6.

B. CHOOSING THE BEST CUT POINT

To choose the best cut point, we first discretize the metrics threshold in the set of examples. The discretization consists of transforming values into a finite number of intervals. After that we re-encode, each value for the selected attribute by associating it with its corresponding interval. It is a powerful heuristic to classify a set of training examples using the best decision tree. It is a good method to determine the most relevant attributes for the classification task. Each metric value is compared to the cut point. The idea is to transform the continuous interval into two intervals according to the cut point. We illustrate how we adapted the set of examples based on Table 2. It represents the list of observations in the set of example for the defect *"Data Class*", in different projects.

In this example, we have two projects P1 and P2. Let's consider the first metrics Access To Foreigner Data (ATFD). Metrics thresholds must be classified into two classes depending on the existence of a defect. Indeed, we have to make a supervised discretization. In this work, we adopted a bottom-up hierarchical clustering. Each item is placed in its own cluster; the next partition is created by merging the two nearest clusters.

In Figure 2, we give a cut point example. Each continuous interval is divided into three parts; first quartile, medium quartile and third quartile. We choose the cut point intervals giving the best degree of association. We use the number of phi coefficient defined in (1) to assess the degree of association between the two variables.

TABLE 2. Data class metrics.

		ATFD	NOM	NOA	NC	...	Data Class
P1	O1	22	15	08	57		NO
	O2	35	10	05	57		NO
P2	O3	04	08	10	113		YES
	O4	11	13	07	113		YES
	O5	05	14	08	113		YES
	O6	63	09	11	113		NO
	O7	25	16	13	113		YES
	O8	02	13	12	368		YES

1) $\emptyset = \sqrt{\frac{x^2}{N}}$ Where $\chi 2$ is derived from Pearson's chi-squared test and N is the total of observations.

As presented in Figure 3, $\emptyset 2$ is the best value and the best cut point is 16.5. In Table 3, we present the final discretization of the observations related to the defect *"Data Class"*. We evaluate the phi coefficient for all metrics. Therefore, the node ATFD will have two arcs one with values <= 16.5 and the other with values > 16.5.

C. CHOOSING THE BEST ROOT NODE

When constructing a decision tree, we have to solve the problem of choosing the best splitting attribute at each node. In fact, many decision trees could be created depending on a splitting point. If we limit our example to the four metrics presented in section 3, the question is: which metric will be the root? Moreover, we have to answer the same question recursively.

For each program iteration, ID3 uses information Shannon entropy to decide the root metric. It is a measure of the amount of uncertainty in a data set. We have to select the attribute that has the smallest entropy value giving the largest information gain. Each branch starts from the most informative metric and each leaf node represents a decision taken after computing all the attributes. The path from the root to a leaf represents

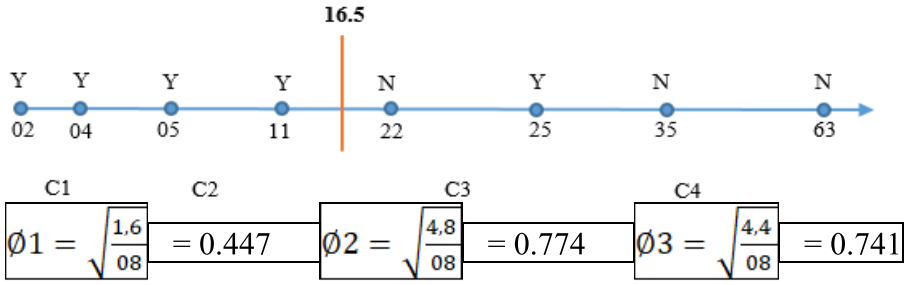


FIGURE 3. Data class metrics discrete values.

TABLE 3. Discretization of metrics values.

		ATFD	NOM	NOA	NC	...	Data Class
P1	O1	> 16.5	>14.25	<=11.25	<=251		NO
	O2	> 16.5	<=14.25	<=11.25	<=251		NO
P2	O3	<= 16.5	<=14.25	<=11.25	<=251		YES
	O4	<= 16.5	<=14.25	<=11.5	<=251		YES
	O5	<= 16.5	<=14.25	<=11.25	<=251		YES
	O6	> 16.5	<=14.25	<=11.25	<=251		NO
	O7	>16.5	>14.25	>11.25	<=251		YES
	O8	<= 16.5	<=14.25	>11.25	>251		YES

the design defect detection rules. The Shannon entropy (E) formula for a set of examples (BE) defined in (2) is:

$$(2) \quad \mathrm{E(BE/c)} = -\sum_{C=C.l} p(c)\log_2 p(c)$$

where,

BE is the set of examples
Cl is the set of classes in ***BE*** (Defect, Not Defect)
P(c) is the proportion of the number of defect values in a class c to the number of elements in the set ***BE***

In Figure 4, we present the decision tree for the example presented in Table 3. We measure the Shannon entropy for each metric.

E(Yes/ATFD) = − P(>16.5) × (P(Yes/>16.5) × log P(Yes/>16.5) + P(No/>16.5) × log P(No/>16.5)) - P(<=16.5) × (P(Yes/<=16.5) × log P(Yes/<=16.5) + P(No/<=16.5) × log P(No/<=16.5)) = 0.11

E(Yes/NOM) = 0.27

E(Yes/NOA) = 0.22

E(Yes/NC) = 0.25

ID3 selects ATFD as the root metric as it has the lowest entropy. All values for ATFD <= 16.5 belong to the class *Yes*, so it is a leaf node. For ATFD > 16.5 we have to re-evaluate the entropy for the rest of metrics using the updated set of examples (BE excluding the metric ATFD). At the NC node, all attributes are classified so ID3 stops. The final tree is shows in Figure 4.

In this example, three rules are extracted:

R1: **IF** ATFD <= 16.5 **THEN** Data class = Yes

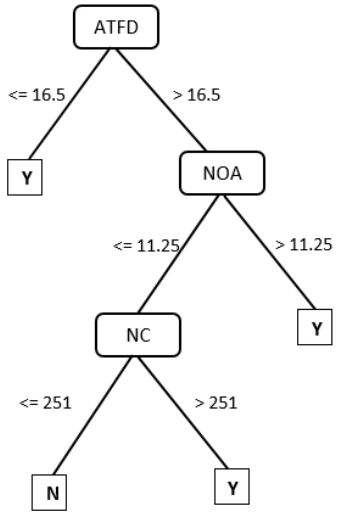


FIGURE 4. Final decision tree.

R2: **IF**ATFD > 16.5 **AND** NOA > 11.25 **THEN** Data class = Yes

R3: **IF** ATFD > 16.5 **AND** NOA <= 11.25 **AND** NC > 251 **THEN** Data class = Yes.

It is clear that R1 and R2 combine a few numbers of metrics and will generate a huge number of suspect classes. Furthermore, if the number of metrics (N) is fixed to 3 then we can extract only the rule R3, which seems to be the most appropriate rule for this illustrative example.

V. VALIDATION

The experiments concern four defects: Lazy Class (LC), Blob, Data Class (DC), and Feature Envy (FE). The validation of the results is based on two well-known indicators: the precision and recall. The precision (3) is the fraction of true design defects among the set of all detected defects. It evaluates the correctness of the approach. The precision assesses the number of true identified defects. The recall (4) indicates the fraction of correctly detected design defects among the set of expected defects. It evaluates the completeness of the approach. The number of true defects missed by ID3 algorithm is measured using the Recall.

$$(3) \quad \textbf{Precision} = \frac{\text{Detected defects} \cap \text{Expected defects}}{\text{Detected defects}}$$

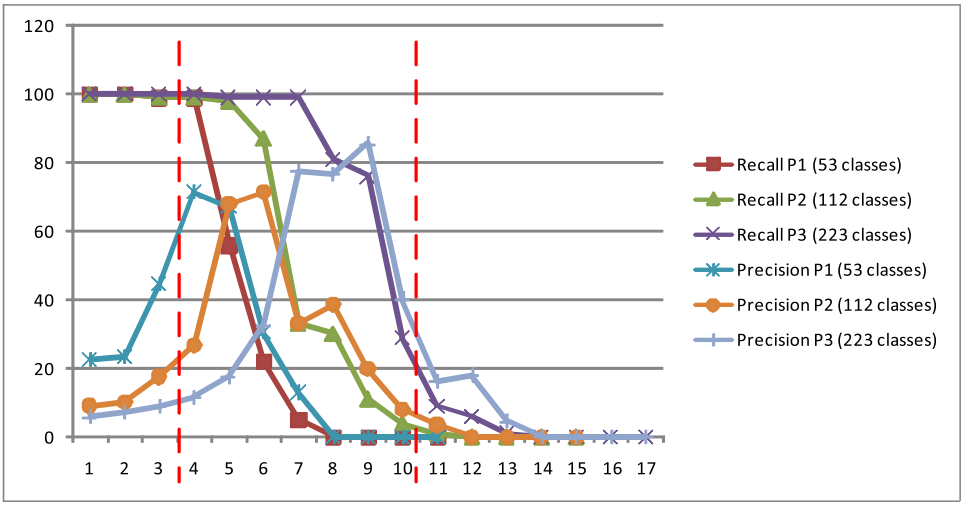


FIGURE 5. Variation of recall and precision depending on N.

$$(4)\ \textbf{Recall} = \frac{\text{Detected defects} \cap \text{Expected defects}}{\text{Expected defects}}$$

We want to validate the following assumptions:

- We detect the majority of existent defects. This assumption is validated by a high recall value. A recall of 100 percent means that we identified all existent defects.
- We generate a reasonable number of defects. Indeed, detecting a large set of defects is inconvenient for the validation process. Even if we detect all existent defects. We must guaranty a good trade-off between recall and precision. In this work, we assume that a precision over than 60 percent is acceptable regarding a recall over 80 percent.

We used the Eclipse plug-in to implement our approach. It takes as input the UML class diagram, sequences class diagrams and N the minimal number of metrics composing a detection rule. Static metrics are evaluated using class diagram while behavioral metrics are evaluated using all sequence diagrams. It generates as output a set of suspect elements in the model based on the generated detection rules. In fact, one fundamental hypothesis in this work is that we base our analysis on complete and final design. The quality of rules detection depends heavily on the completeness of sequence diagram models. Designers must ensure that they modeled all software scenarios and captured all messages between classes.

The validation is based on the reverse engineered designs of five open-source Java systems. We have tested our approach on: Xerces v2.7, ArgoUML 0.19.8, Lucene 1.4, Log4j 1.2.1 and GanttProject v1.10.2. Table 4 resumes the important information related to these projects. The choice of these projects was motivated by:

- The size, we selected a medium project and large-sized project.
- They are open-source projects.

TABLE 4. Projects information.

Project	Number of classes	Number of Blob	Number of LC	Number of DC	Number of FE
Xerces v2.7	683	29	23	17	58
ArgoUML 0.19.8	1244	15	29	15	78
Lucene 1.4	189	7	8	2	23
Log4j 1.2.1	209	11	6	5	33
GanttProject v1.10.2	241	22	12	10	46

- They are studied and results are available (even that they are based on code level while we are working on model level).

We used PMD 5.4.3 (433 classes) and Nutch 1.12 (247 classes) projects for the construction of the set of examples. We entered manually the design defects we intend to detect. The validation process was performed within two iterations. In the first, the set of examples is created using only PMD project. For the second, the set of examples is enhanced using Nutch project and hence, the number of defects is increased. Results are then compared in order to evaluate the impact of the number of defects in the set of examples on the detection quality.

The choice of N value influences the recall and precision. Minimizing this value increases the recall and decreases the precision and vice versa. It also depends on project size. N must be optimized giving the best trade-off between precision and recall. Figure 5 provides the variation of recall and precision depending on N. We tested three projects, the first two projects was implemented by four engineers specialized in software engineering and the last project was a part form an information system for the Ministry of Social Affairs. For each project we have 53, 112 and 223 classes respectively and containing 18, 25 and 34 design defects to predict the best interval of N threshold.

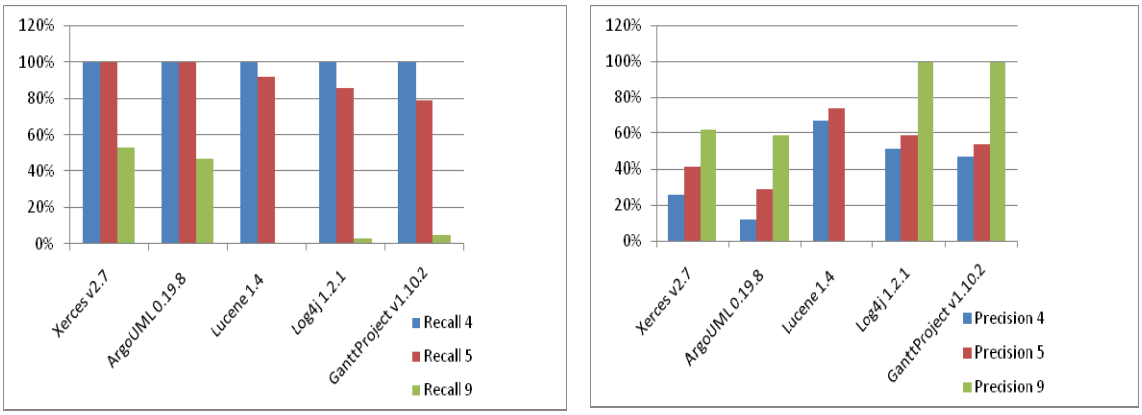


FIGURE 6. Precisions and recall results.

We found that for N between four and ten we have the best trade-off. Therefore, the best thresholds of N is 4, 5 and 7 for P1, P2 and P3 respectively.

Finding the best value for N is difficult, but using our experiment, we predict acceptable results. In general, based on our experiments, we suggest that the minimum value for N is 4 and the maximum value is 9 which is 1/3 of the number of metrics. Nevertheless, depending on the project designers may try other values for N if the results are not satisfying.

We perform all computations on a desktop computer (Intel i5 CPU running at 2.53 GHz with 4 GB of RAM). The execution time is less than 30 sec. It does not include the reverse engineered designs.

VI. RESULTS AND DISCUSSION

Figure 6 reports the precisions and recall values for the three executions using deferent values of N (4, 5 and 9). The best precision values are 62%, 59%, 74%, 100%, 100% for Xerces v2.7, ArgoUML 0.19.8, Lucene 1.4, Log4j 1.2.1 and GanttProject v1.10.2., respectively. The best recall value is 100% for all projects.

We notice that for Lucene 1.4, when assigning 9 to N, the generated rules do not detect any defect. In fact, the more the number of metrics in rule increases, the more the number of detection decreases. The recall value is inversely proportional to the precision. When N increases, the precision tends to increase (from 12% to 100%) and the recall tends to decrease (from 100 % to 3%). The best trade-off between recall and precision vary from one project to another as reported in Table 5. In our work, we assume that the precision has to be over than 50% reducing the number of false positive errors.

We have excellent results for the two projects: Lucene 1.4 and Log4j 1.2.1. Indeed, we found all expected defects having a recall value over than 50%. For the rest of the projects, expected ArgoUML 0.19.8, results are over than 50%, which is a good detection rate. For ArgoUML 0.19.8 we detect 47% percent of defects, it is an acceptable result giving a precision of 59%.

In Table 6, we detail the number of detection for each project and each defect. It shows that even if the detection rate is under 50% for ArgoUML 0.19.8, it is due to the low detection rate for FE defect (0.38). For the defects LC, Blob and DC, the recall is higher than 50% varying within 0.51, 0.6 and 0.73 respectively. We can conclude that the proposed approach is able to detect the majority of defects.

TABLE 5. Best precision and recall values.

Project	Precision	Recall	N
Xerces v2.7	62%	53%	9
ArgoUML 0.19.8	59%	47%	9
Lucene 1.4	67%	100%	4
Log4j 1.2.1	51%	100%	4
GanttProject v1.10.2	54%	79%	5

TABLE 6. The number of detection for each project and each defect.

Project	Number of Blob	Number of LC	Number of DC	Number of FE
Xerces v2.7	55%	47%	52%	51%
ArgoUML 0.19.8	60%	51%	73%	38%
Lucene 1.4	100%	100%	100%	100%
Log4j 1.2.1	100%	100%	100%	100%
GanttProject v1.10.2	54%	75%	90%	58%

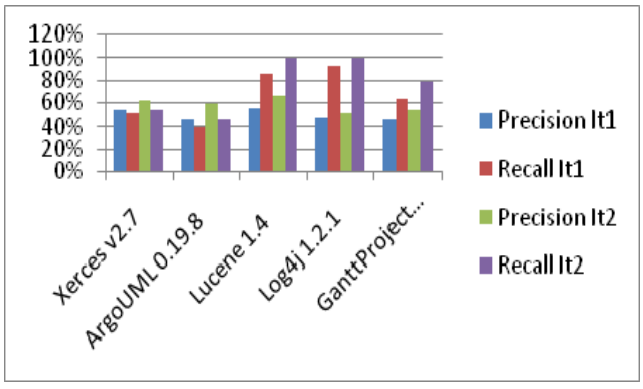


FIGURE 7. Variation of recall and precision.

TABLE 7. F1 score.

Project	F1 Score	N
Xerces v2.7	49%	9
ArgoUML 0.19.8	52%	9
Lucene 1.4	80%	4
Log4j 1.2.1	67%	4
GanttProject v1.10.2	64%	5

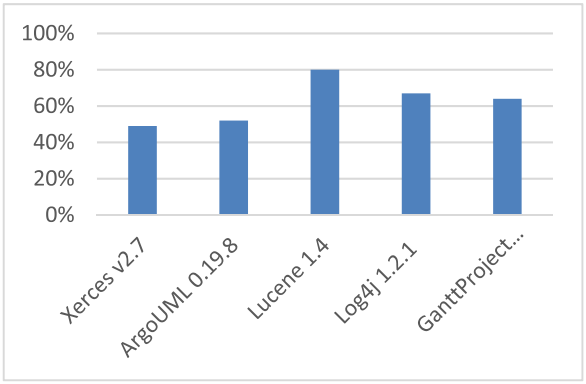


FIGURE 8. Variation of F1 score.

Figure 7 presents the variation of the recall and precision according to the size of the set of examples. Results show that the quality of the detection increases with the number of defects examples. In this work, we limit the set of examples to two open source projects that give results varying from excellent to satisfying. However, finding the optimal size of the set of examples needs further investigations, that will be discussed in future research.

Finally, results show that the use of decision tree technique is a promising way to investigate the detection of design defects at the model level. We demonstrated that the rules generated using decision tree give a good detection results (recall of 100% and Precision $>$ 50%), as shown in Figure 8. The accuracy rate of the detection is about 80%.

Giving the number of metrics combined with their respective continuous range of threshold, it is impossible for experts to find manually the best metric-based rules. Results prove that our proposed rule generation method; helps the experts to cope with time and effort waste.

As shown in table 7, F1 score (5) is the weighted average of Precision and Recall. It considers both the precision and the recall of the test to compute the score; it is a measure of the test accuracy:

$$\text{F1} = 2 * \frac{\text{Precision} * \text{Recall}}{\text{Precision} + \text{Recall}} \quad (5)$$

Furthermore, results shown in Figure 8 prove that it is possible to detect design defects at model level by only analyzing class and sequence diagrams and avoiding the propagation of defects to code. The accuracy of the detection vary from one project to another but still acceptable reaching the 80%, this is a very promising results for a predictive detection less time and effort consuming.

VII. CONCLUSION

In this paper, we presented a new approach for the design defects detection at the model level. This work leads to define a standard way for model quality quantification. We introduced an adaptation of ID3 decision tree algorithm to identify *anti-patterns* and bad smells in object-oriented design. We tested and evaluated our approach on five open source projects by measuring the precision and the recall. We showed that the efficiency and the precision of the detection vary from satisfying to excellent with a recall that reaches 100 percent. We proved that using our approach we detect the majority of design anomalies at the model level based on analyzing the class and sequence diagrams.

As future work, we plan to eliminate these defects avoiding their propagation to code. We plan also to extend the detection to other defects. Finally, we plan to implement a model-refactoring framework, integrating the detection and correction approaches.

REFERENCES

[1] M. Misbhauddin and M. Alshayeb, "UML model refactoring: A systematic literature review," *Empirical Softw. Eng.*, vol. 20, no. 1, pp. 206–251, Feb. 2015.
[2] R. Panigrahi, L. Kumar, and S. Kuanar, "An empirical study to investigate different SMOTE data sampling techniques for improving software refactoring prediction," in *Proc. ICONIP*, 2020, pp. 23–31.
[3] K. Czanecki and S. Helsen, "Classification of model transformation approaches," in *Proc. OOPSLA Workshop Generative Techn. Context (MDA)*, 2003, pp. 1–7.
[4] M. Mohamed, R. Mohamed, and G. Khaled, "Classification of model refactoring approaches," *J. Object Technol.*, vol. 8, no. 6, pp. 121–126, 2009.
[5] J. Zhang, Y. Lin, and J. Gray, "Generic and domain-specific model refactoring using a model transformation engine," in *Model-Driven Softw. Develop.* Berlin, Germany: Springer, 2005.
[6] S. Freire, A. Passos, M. Mendonca, C. Sant'Anna, and R. O. Spinola, "On the influence of UML class diagrams refactoring on code debt: A family of replicated empirical studies," in *Proc. 46th Euromicro Conf. Softw. Eng. Adv. Appl. (SEAA)*, Aug. 2020, pp. 346–353.
[7] W. J. R. C. Brown Malyeau, H. W. S. Mccormick, and T. J. Mowbray, *AntiPatterns : Refactoring Software, Architecture and Projects in Crisis*. Hoboken, NJ, USA: Wiley, 1998.
[8] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring : Improving the Design of Existing Code*. Reading, MA, USA: Addison-Wesley, 1999.
[9] M. Zhang, T. Hall, and N. Baddoo, "Code bad smells: A review of current knowledge," *J. Softw. Maintenance Evol., Res. Pract.*, vol. 23, no. 3, pp. 179–202, Oct. 2010.
[10] H. Mumtaz, M. Alshayeb, S. Mahmood, and M. Niazi, "A survey onUMLmodel smells detection techniques for software refactoring," *J. Softw., Evol. Process*, vol. 31, no. 3, Mar. 2019, Art. no. e2154.
[11] F. B. e Abreu and W. Melo, "Evaluating the impact of object-oriented design on software quality," in *Proc. 3rd Int. Softw. Metrics Symp.*, Mar. 1996, pp. 90–99.
[12] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *Proc. 20th IEEE Int. Conf. Softw. Maintenance*, Sep. 2004, pp. 350–359.
[13] M. Alzahrani, "Measuring class cohesion based on client similarities between method pairs: An improved approach that supports refactoring," *IEEE Access*, vol. 8, pp. 227901–227914, 2020.
[14] S. Mäkelä and V. Leppänen, "Client-based cohesion metrics for java programs," *Sci. Comput. Program.*, vol. 74, nos. 5–6, pp. 355–378, Mar. 2009.
[15] K. Erni and C. Lewerentz, "Applying design-metrics to object-oriented frameworks," in *Proc. 3rd Int. Softw. Metrics Symp.*, Mar. 1996, pp. 64–74.

[16] M. Kessentini, W. Kessentini, H. Sahraoui, M. Boukadoum, and A. Ouni, "Design defects detection and correction by example," in *Proc. IEEE 19th Int. Conf. Program Comprehension*, Jun. 2011, pp. 81–90.
[17] P. Tianual and A. Pohthong, "Defects detection technique of use case views during requirements engineering," in *Proc. 8th Int. Conf. Softw. Comput. Appl.*, Feb. 2019, pp. 277–281.
[18] F. Arcelli Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," *Empirical Softw. Eng.*, vol. 21, no. 3, pp. 1143–1191, Jun. 2016.
[19] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur, "DECOR: A method for the specification and detection of code and design smells," *IEEE Trans. Softw. Eng.*, vol. 36, no. 1, pp. 20–36, Jan. 2010.
[20] A. Ouni, M. Kessentini, H. Sahraoui, and M. Boukadoum, "Maintainability defects detection and correction: A multi-objective approach," *Automated Softw. Eng.*, vol. 20, no. 1, pp. 47–79, Mar. 2013.
[21] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "Detecting bad smells in source code using change history information," in *Proc. 28th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2013, pp. 268–278.
[22] M. Alenezi, M. Akour, and O. Al Qasem, "Harnessing deep learning algorithms to predict software refactoring," *TELKOMNIKA (Telecommun. Comput. Electron. Control)*, vol. 18, no. 6, p. 2977, Dec. 2020.
[23] M. V. Kempen, C. Michel, K. Derrick, and B. Andrew, "Towards proving preservation of behaviour of refactoring of UML models," *Proc. SAICSIT*, 2005, p. 252.
[24] M. Raul, "Analysis and definition of a language independent refactoring catalog," in *Proc. 17th Conf. Adv. Inf. Syst. Eng. (CAiSE)*, Jun. 2005, p. 8.
[25] M. Raul and C. Yani, "Towards a language independent refactoring framework," in *Proc. 1st Int. Conf. Softw. Data Technol. (ICSOFT)*, Setubal, Portugal, 2006, pp. 165–170.
[26] A. Ghannem, G. El Boussaidi, and M. Kessentini, "On the use of design defect examples to detect model refactoring opportunities," *Softw. Qual. J.*, vol. 24, no. 4, pp. 947–965, Dec. 2016.
[27] M. Mohamed, R. Mohamed, and G. Khaled, "M-REFACTOR: A new approach and tool for model refactoring," *ARPN J. Syst. Softw.*, vol. 1, no. 4, pp. 117–122, Jul. 2011.
[28] M. Mohamed and A. Sarra, "Extracting and modeling design defects using gradual rules and UML profile," in *Proc. 5th Int. Conf. Comput. Sci. Appl. (CIIA)*, 2015, pp. 574–583.
[29] M. Misbhauddin and M. Alshayeb, "An integrated metamodel-based approach to software model refactoring," *Softw. Syst. Model.*, vol. 18, no. 3, pp. 2013–2050, Jun. 2019.
[30] V. Cortellessa, R. Eramo, and M. Tucci, "From software architecture to analysis models and back: Model-driven refactoring aimed at availability improvement," *Inf. Softw. Technol.*, vol. 127, Nov. 2020, Art. no. 106362.
[31] M. Tanhaei, "A model transformation approach to perform refactoring on software architecture using refactoring patterns based on stakeholder requirements," *AUT J. Math. Comput.*, vol. 1, pp. 179–216, Oct. 2020.
[32] B. K. Sidhu, K. Singh, and N. Sharma, "A machine learning approach to software model refactoring," *Int. J. Comput. Appl.*, pp. 1–12, Jan. 2020.
[33] M. Alshayeb, H. Mumtaz, S. Mahmood, and M. Niazi, "Improving the security of UML sequence diagram using genetic algorithm," *IEEE Access*, vol. 8, pp. 62738–62761, 2020.
[34] A. J. Riel, *Object-Oriented Design Heuristics*. Reading, MA, USA: Addison-Wesley, 1996.
[35] B. A. Kitchenham, *Software Metrics: Measurement for Software Process Improvement*. Hoboken, NJ, USA: NCC Blackwell Publishers, 1996.
[36] N. E. Feton and A. S. L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*, 2nd ed. Boston, MA, USA: PWS Publishing Co, 1998, p. 656.
[37] A. AbuHassan and M. Alshayeb, "A metrics suite for UML model stability," *Softw. Syst. Model.*, vol. 18, no. 1, pp. 557–583, Dec. 2016.
[38] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia, "Detecting code smells using machine learning techniques: Are we there yet?" in *Proc. IEEE 25th Int. Conf. Softw. Anal., Evol. Reengineering (SANER)*, Mar. 2018, pp. 612–621.
[39] J. R. Quinlan, "Induction of decision trees," *Mach. Learn.*, vol. 1, no. 1, pp. 81–106, Mar. 1986.

MOHAMED MADDEH received the Ph.D. degree in computer science from the National School of Computer Sciences of Tunis (ENSI), in 2012/2013. He was an Assistant Professor in computer science with the Higher Institute of Finance and Taxation of Sousse (ISFF). He wor ked as an Analyst and Data Base Administrator at the Ministry of Social Affairs, from 2003 to 2009. He served in many administrative positions with King Saud University. He was a Coordinator of the Web and computer certification (moodle) in collaboration with the Virtual University of Tunis (UVT). He was the Head of the Department of Law and Finance at ISFF, in 2014. He is currently an Assistant Professor with King Saud University. His main research interests include the fields of software engineering and model driven engineering. He was a member of the Scientific Committee of ISFF, in 2011. He was also a Treasurer of the Tunisian Association of Artificial intelligence (ATIA).

SARRA AYOUNI received the M.S. degree from the Faculty of Sciences of Tunis (FST) and the Ph.D. degree in computer science from the University of Montpellier 2, France. She is currently an Assistant Professor with the Department of Information Systems, College of Computer and Information Sciences (CCIS), Princess Nourah Bint Abdulrahman University (PNU). She is also the Coordinator of Distance Education at CCIS. Her main research interests include data science, artificial intelligence, fuzzy datamining, and e-learning.

SULTAN ALYAHYA received the B.Sc. degree (Hons.) in information systems from King Saud University, the M.Sc. degree in information systems engineering from Cardiff University, U.K., and the Ph.D. degree in computer science from Cardiff University, in 2013. From 2013 to 2016, he was the Vice-Chair of the Department of Information Systems, and the Head of the College of Applied Computer Science, from 2016 to 2019. He served in many administrative positions at King Saud University. He is currently an Associate Professor with the College of Computer and Information Sciences, King Saud University. His main research interests include the fields of global software development, co-ordination in software engineering, and computer supported co-operative work (CSCW).

FAHIMA HAJJEJ received the Ph.D. degree in computer science from the Faculty of Sciences of Sfax, in 2016/2017. She is currently an Assistant Professor with the Department of Information Systems, College of Computer and Information Sciences at PNU, Saudi Arabia. She is also a member of the Research Laboratory in Technologies of Information and Communication and Electrical Engineering (LaTICE). Her research interests include the modeling concepts of e-learning, e-assessment, integration of formal and semi-formal methods, data science, and big data.