Investigating the Accuracy of Metric-Based and Machine Learning Approaches in Detecting Design Patterns

Author: Nils Dunlop
Academic supervisor: Jennifer Horkoff

***Abstract*—Design pattern detection approaches have evolved, with machine-learning methods gaining prominence. However, implementing machine-learning models can be challenging due to extensive training requirements and the need for large labeled design pattern datasets. We propose a simpler alternative that overcomes machine learning limitations. This study compares machine-learning design pattern detection with our metric approach, which excels with small datasets and effectively handles Java and C++ patterns. By extracting metrics from programs using scripts and evaluating patterns with predetermined thresholds, our method achieves comparable or better f-score than existing methods without relying on AI. Our findings demonstrate the potential of our metric approach as a practical alternative, simplifying design pattern analysis in software development. Future research should explore its application in industry contexts.**

***Index Terms*—Design Pattern Detection, Metrics, Thresholds, Machine Learning**

I. Introduction

A design pattern provides a general and repeatable solution to a recurring problem in software design. Utilizing a design pattern when implementing or enhancing existing code facilitates the creation of a structure that can be easily expanded, comprehended, and maintained over time. While numerous design patterns exist for various programming languages, the 23 Gang-of-Four (GoF) design patterns are particularly renowned for their extensive use, especially in object-oriented design [1]. Typically, design patterns comprise classes that outline the roles and capabilities of objects.

Understanding existing source code or programs from large corporations or open-source projects can be daunting, particularly when design patterns are developed without explicit class names, comments, or documentation. Manual detection of design patterns in existing programs often falls short, leading to potentially overlooked patterns. To address this, pattern detection research and software have been developed to effectively automate design pattern detection [2].

Kramer and Prechelt pioneered an approach to design pattern detection that has been refined and expanded upon in subsequent research papers. More recently, machine learning and deep learning models have been explored for their utility in design pattern detection [3]. However, some of these models have shown limitations. Researchers Pandey and Horkoff from Gothenburg University demonstrated last year that while pre-trained deep learning models successfully detected a range of design patterns in Java code, their detection was slightly sensitive to variations in class structure and class naming [4].

This thesis expands on the work by Pandey and Horkoff [4] to evaluate how well a metric-based design pattern detection works compared to machine learning. The primary objective is to analyze three GoF design patterns: Singleton, Adapter, and State. We use eleven code metrics for each pattern and their corresponding roles to determine the presence of design patterns. As further seen in Table I, a design pattern is detected only once per its associated question. Our approach assesses design pattern detection accuracy within eighteen open-source repositories of various sizes: nine repositories in Java from PMART [5] and nine repositories in C++ from GitHub [6–14].

Through our research, we compare how well metric-based methods and machine learning methods can find design patterns. The results will provide insight into the accuracy of metric-based approaches and a starting point for further collaborative research with Volvo and Chalmers Artificial Intelligence Research Center.

II. Related Work

Early efforts in design pattern detection primarily revolved around code structure matching. In 1996, Kramer and Prechelt suggested a method for detecting Gang-of-Four (GoF) design patterns, leveraging a Prolog representation of C++ header files for structure matching [3]. However, this technique often produced false positives i.e., patterns incorrectly identified as matches. When analysing five GoF design patterns, Kramer and Prechelt achieved a 40% detection rate with high false positives. Furthermore, some GoF patterns are small with few relationships, making it difficult to differentiate them based on structure alone. For example, the state and strategy design patterns are very similar when viewed solely in terms of structure [15].

To alleviate these issues, researchers moved towards semantic matching, analyzing method bodies and properties to reduce false positives and enhance design pattern detection accuracy. Lucia [16] extended the concept of structural matching by introducing a post-processing step that deployed semantic matching to filter out false positives, managing to recover over 80% of GoF design patterns using this approach.

Subsequently, the research focus shifted to utilizing unified modeling language (UML) representations of code and role stereotypes for design pattern analysis. More recently, methods based on machine learning and classification have been utilized to realize more robust and automated design pattern detection systems. Ho-Quang proposed the application of machine learning for role stereotype classification in two of his papers [17]. The study found the Random Forest machine learning algorithm highly effective at classifying design patterns [18].

Building upon these advances, Chihada et al. [1] explored various techniques for identifying design patterns, encompassing metric-based methods, logical reasoning, similarity scoring, ontology-based methods, and learning-based methods. Their study utilized machine learning to eliminate false positives post the structural matching phase. The code was transformed into abstract syntax trees and inputted into a pre-trained neural network to recognize patterns. Meanwhile, Tsantalis et al. [19] adopted similarity scoring between code graph vertices to detect design patterns. Their proposed methodology achieved a remarkable 95% design pattern detection accuracy across three analyzed Java repositories, demonstrating high efficacy even with patterns deviating from their standard representation.

In 2022, researchers from Chalmers and Volvo Group utilized programming language models to measure the controller-handler design pattern compliance in the automotive industry. When applied to an automotive code corpus, the resulting model accurately predicted pattern compliance with a 92% accuracy rate [20].

While metrics and machine learning have been frequently applied in design pattern detection, their combined use has been less frequent. However, Uchiyama et al. integrated these approaches to measure the effectiveness of design pattern detection within object-oriented Java programs [21]. Their methodology analyzed five of the 23 GoF design patterns: Singleton, Template Method, Adapter, State, and Strategy. They developed specific metrics for each pattern to identify particular code structures by employing the Goal Question Metric method. Subsequently, a neural network was utilized to detect the patterns and minimize false positives. Although the approach proposed in Uchiyama et al. [21] outperformed two traditional detection methods, it displayed a higher vulnerability to false positives due to the absence of strict conditions.

Building upon the established work of Uchiyama et al. [21], our study evaluates the effectiveness of metric-based approaches versus programming language models (PLMs) in identifying design patterns within open-source Java and C++ code. Our central aim is to determine the optimal metrics to detect design patterns. By contrasting these two methodologies, we seek to gain insights into which technique is more successful and offer guidance for best practices in identifying and applying design patterns in software development.

III. Research Methodology

This study aims to evaluate the effectiveness of a metric-based approach in detecting the Adapter, Singleton, and State design patterns within Java source code and to compare it to Pandey and Horkoff's machine learning-based approach [4].

Furthermore, the study aims to investigate the accuracy of metric-based design pattern detection within C++ source code and compare it to the results obtained for Java source code.

A. *Research questions*

RQ1: How accurate are metric-based approaches in detecting Adapter, Singleton, and State design patterns within C++ code compared to their detection in Java code?

RQ2: How accurate are metric-based approaches compared to machine learning techniques in detecting Adapter, Singleton, and State design patterns in Java open-source code?

Table I: Patterns and corresponding metrics. Selection gathered from Uchiyama et al. [21] and Issaoui et al. [22].

Pattern	Role	Question	Metric	Metric Description
Singleton	Singleton	Is a static field defined?	NSF	Number of static fields
		Is the constructor called from another class?	NOPC	Number of private constructors
		Does the method return a singleton instance?	NSM	Number of static methods
Adapter	Target	Are abstract methods defined?	NOAM	Number of abstract methods
		Is the class defined as interface?	NOI	Number of interfaces
	Adapter	Are override methods defined?	NORM	Number of overridden methods
		Is the Adaptee field defined?	NOF	Number of fields
			NOOF	Number of object fields
	Adaptee	Are methods used by the Adapter role defined?	NOM	Number of methods
		Is the class referred to by other classes?	NCOF	Number of other classes with field of own type
State	Context	Are methods to set states defined?	NOM	Number of methods
		Is the State field defined?	NOF	Number of fields
	State	Are abstract methods defined?	NOAM	Number of abstract methods
		Is the class referred as an interface?	NOI	Number of interfaces
		Is the class referred to by other classes?	NCOF	Number of other classes with field of own type
	ConcreteState	Number of override methods?	NORM	Number of overridden methods
		Is a method that describes the change state defined?	NMGI	Number of methods generating instances

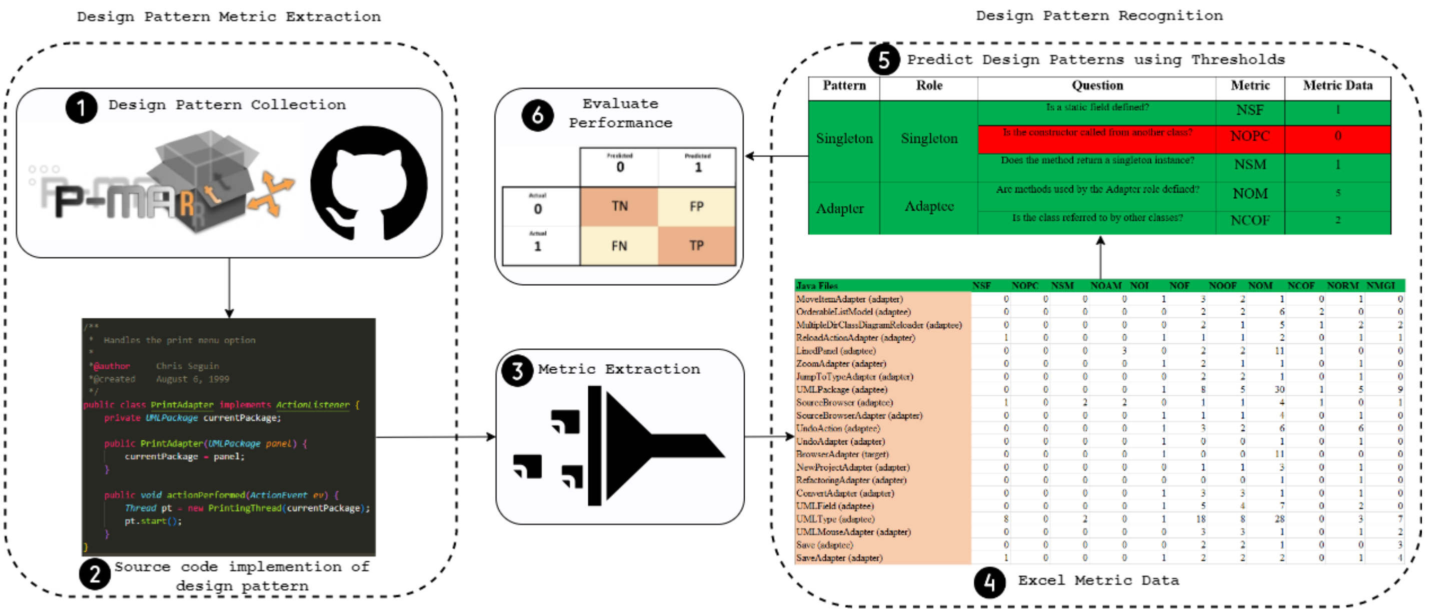
B. *Research methodology to be used*

The research methodology utilized in this study draws on Mining Software Repositories (MSR), a method to analyze data in software repositories to gain insights into strategies for software quality improvement.

This study analyzed three design patterns: Singleton, Adapter, and State. We selected these patterns to effectively represent the three distinct categories of design patterns: behavioral (State), creational (Singleton), and structural (Adapter). Design patterns generally consist of classes that define the roles and characteristics of an object. To detect design patterns, we utilized metrics and questions to evaluate whether the new code adheres to these conditions.

Table II: Java Design Patterns Source File Overview

Design Pattern	*Q-UML*	*Lexi*	*JRefactory*	*Netbeans*	*JUnit*	*JHotDraw*	*MapperXML*	*Nutch*	*PMD*	*Total*
Adapter	N/A	N/A	31	71	N/A	31	4	4	3	144
Singleton	N/A	3	11	2	2	2	3	1	1	25
State	N/A	N/A	8	N/A	N/A	16	N/A	N/A	N/A	24
Non-DP	12	5	12	8	8	10	12	7	3	77



Design Pattern Metric Extraction
Design Pattern Recognition
1 Design Pattern Collection
P-MARt
2 Source code implemention of design pattern
3 Metric Extraction
4 Excel Metric Data
5 Predict Design Patterns using Thresholds
6 Evaluate Performance

```
/**
 *  Handles the print menu option
 *
 *@author     Chris Seguin
 *@created    August 6, 1999
 */
public class PrintAdapter implements ActionListener {
    private UMLPackage currentPackage;

    public PrintAdapter(UMLPackage panel) {
        currentPackage = panel;
    }

    public void actionPerformed(ActionEvent ev) {
        Thread pt = new PrintingThread(currentPackage);
        pt.start();
    }
}
```

	Predicted 0	Predicted 1
Actual 0	TN	FP
Actual 1	FN	TP

Pattern	Role	Question	Metric	Metric Data
Singleton	Singleton	Is a static field defined?	NSF	1
		Is the constructor called from another class?	NOPC	0
		Does the method return a singleton instance?	NSM	1
Adapter	Adaptee	Are methods used by the Adapter role defined?	NOM	5
		Is the class referred to by other classes?	NCOF	2

Java Files	NSF	NOPC	NSM	NOAM	NOI	NOF	NOOF	NOM	NCOF	NORM	NMGI
MoveItemAdapter (adapter)	0	0	0	0	1	3	2	1	0	1	0
OrderableListModel (adaptee)	0	0	0	0	0	2	2	6	2	0	0
MultipleDirClassDiagramReloader (adaptee)	0	0	0	0	0	2	1	5	1	2	2
ReloadActionAdapter (adapter)	1	0	0	0	1	1	1	2	0	1	1
LinedPanel (adaptee)	0	0	0	3	0	2	2	11	1	0	0
ZoomAdapter (adapter)	0	0	0	0	1	2	1	1	0	1	0
JumpToTypeAdapter (adapter)	0	0	0	0	0	2	2	1	0	1	0
UMLPackage (adaptee)	0	0	0	0	1	8	5	30	1	5	9
SourceBrowser (adaptee)	1	0	2	2	0	1	1	4	1	0	1
SourceBrowserAdapter (adapter)	0	0	0	0	1	1	1	4	0	1	0
UndoAction (adaptee)	0	0	0	0	1	3	2	6	0	6	0
UndoAdapter (adapter)	0	0	0	0	1	0	0	1	0	1	0
BrowserAdapter (target)	0	0	0	0	1	0	0	11	0	0	0
NewProjectAdapter (adapter)	0	0	0	0	0	1	1	3	0	1	0
RefactoringAdapter (adapter)	0	0	0	0	0	0	0	1	0	1	0
ConvertAdapter (adapter)	0	0	0	0	1	3	3	1	0	1	0
UMLField (adaptee)	0	0	0	0	1	5	4	7	0	2	0
UMLType (adaptee)	8	0	2	0	1	18	8	28	0	3	7
UMLMouseAdapter (adapter)	0	0	0	0	0	3	3	1	0	1	2
Save (adaptee)	0	0	0	0	0	2	2	1	0	0	3
SaveAdapter (adapter)	1	0	0	0	1	2	2	2	0	1	4

Figure 2: Overview of proposed metric-based method.

1) Data collection

For our analysis, we collected source code and design patterns from open-source repositories. Java repositories were obtained from PMART, a comprehensive collection of projects featuring labelled design patterns. Additionally, we gathered open-source C++ repositories from GitHub that contained similarly labelled design patterns.

To analyze Java repositories from PMART, we handled nine programs: QuickUML, Lexi, JRefactory, Netbeans, JUnit, JHotDraw, MapperXML, Nutch, and PMD [5]. These programs were selected due to their clearly labelled design patterns and broad usage in the field. Details regarding the size and specific parameters of these programs can be found in Table III.

Table III: Overview of PMART repositories size and parameters

Repository	*Files*	*Classes*	*Fields*	*Methods*
QuickUML 2001	155	142	466	1264
Lexi v0.1.1 alpha	24	23	178	601
JRefactory v2.6.24	569	556	1403	4690
Netbeans v1.0.x	2558	2238	16736	25446
JUnit v3.7	79	69	188	856
JHotDraw v5.1	155	136	363	1393
MapperXML v1.9.7	217	195	819	2307
Nutch v0.4	172	149	1052	1832
PMD v1.8	447	423	1614	3752

For each chosen program, source files associated with the respective design patterns were extracted (as seen in Table II). Additionally, 77 files devoid of any design pattern were examined to prevent a common pitfall in machine learning known as overfitting, where a model performs well on the training data but poorly on unseen data.

Turning our attention to our C++ analysis, we selected source code from eight different GitHub repositories, each showcasing modern C++ implementations of the three investigated design patterns [6–13]. As with our Java analysis, to prevent overfitting, we included ten C++ files from the CPlusPlusThings GitHub repository devoid of design patterns [14]. This ensured that our C++ dataset was diverse and representative, including 11 code instances for each design pattern.

Despite the smaller size of C++ data compared to Java data, as detailed further in Table II, our C++ analysis was sufficient to validate the C++ design pattern detection effectiveness. The repositories from which we compiled our C++ dataset were selected based on various factors, including size, activity, and popularity. The primary metric in this selection process was the number of favorites on GitHub, an indicator of repository relevance and utility in the developer community.

To extract the metric data, we first obtained the source code for each design pattern from all the repositories and organized them into separate folders based on the pattern type. To automate the data collection process, we developed eleven Java scripts and eleven C++ scripts [23]. In both Java and C++, we employed abstract syntax tree (AST) libraries to extract metrics accurately by fields, methods, constructors, and interfaces being easily attainable. For Java, we utilized the JavaParser library which offers extensive documentation, wide usage, and robust capabilities for parsing industry-level code [24]. In the case of C++, we leveraged Python along with the Python binding of the C++ LLVM Clang repository [25], opting for Python due to our familiarity with the language compared to C++. Figure 1 presents a Java example illustrating how our private constructor extraction script operates. The script identifies the count, name, and location of private constructors within a Java file, which can then be retrieved by a main function.

```java
public class PrivateConstructorCounter{

public static void main(String[] args) {
try {
JavaParser javaParser = new JavaParser();
CompilationUnit cu = javaParser.parse(path).toFile()).getResult();

AtomicInteger count = new AtomicInteger();
cu.findAll(ConstructorDeclaration.class).stream().filter(c ->
c.getModifiers().contains(Modifier.privateModifier())).forEach(c -> {

count.getAndIncrement();
String constructorName = c.getNameAsString();
String constructorLocation = c.getBegin().map(Position::toString);
}
```

Figure 1: Sample source code of private constructor extraction script.

The Java and C++ extraction scripts had slight differences due to specific requirements. In C++, we compiled header files, while object fields and abstract methods were referred to as data members and virtual functions, respectively. As the thesis progressed, we refined these scripts to accurately handle override methods and classes featuring fields of their own type. To ensure this, we iterated over all files within the source folders of the respective repositories.

As seen in Figure 2, utilizing the extraction scripts, we collected metric data from all files and compiled it into an Excel spreadsheet.

2) *Data analysis*

After collecting metric data, we established threshold criteria to predict the corresponding metric roles. A minimum value of one was required for the corresponding metrics to meet the criteria for the defined questions (NSF, NOPC, NOAM, NOI, NORM, and NCOF). For the more specific questions (NSM, NOF, NOOF, NOM, and NMGI), such as analyzing present instances, at least one instance had to be detected by the extraction scripts. For roles with only two metric questions (Target, Adaptee, Context, and ConcreteState), both questions needed to be met to determine presence of the role. Meeting at least two of the three questions was sufficient for roles with three questions (Singleton, Adapter, and State).

To balance the sensitivity and specificity of the design pattern metrics, we carefully selected metric thresholds to minimize both false positives and false negatives. We used these thresholds to predict the roles of each file based on the extracted metric data. It is important to note that a single file can fulfil multiple roles, as a piece of source code can adhere to more than one design pattern.

Once the roles were predicted, we derived true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN) by contrasting the predicted roles with the actual roles observed in each file. Following this, we computed precision and recall values from these measures to calculate the accuracy of our metric-based design pattern detection for both Java and C++ code. The precision, recall, and f-score values obtained addressed our first research question (RQ1).

Finally, we compared our Java precision, recall, and f-score values to those obtained in related research by Pandey and Horkoff [4], Uchiyama et al. [21], and Tsantalis et al. [19]. Pandey and Horkoff utilized a Facebook Programming Language Model (PLM) (a language model designed to understand and generate code of a particular programming language) for Java design pattern detection. Uchiyama et al. employed a convolutional neural network with software metrics. Finally, Tsantalis et al. utilized a code graph matching algorithm. The contrasting methodologies allow us to address our second research question (RQ2) by evaluating the accuracy of our metric-based model in detecting design patterns compared to machine learning models.

IV. Results

In this study, we aimed to assess the effectiveness of our metric-based design pattern detection method. Our approach (visually depicted in Figure 2) involved extracting metric data to determine the presence of three specific design patterns: Adapter, Singleton, and State. We analyzed 270 Java source files from nine repositories on PMART and 41 C++ source files from nine GitHub repositories. In the PMART repositories, we found that the Adapter pattern was the most frequent, appearing in 144 source files. In contrast, Singleton and State patterns were less prevalent, appearing in 25 and 24 files, respectively. To predict the roles within these design patterns, we established metric thresholds. This allowed us to assess the accuracy of our approach through measures of precision, recall, and f-score.

Table IV. Precision, recall and f-score of Java pattern detection.

Java Pattern	***Number of files***	***Precision***	***Recall***	***f-score***
Adapter	144	70.12%	85.92%	77.22%
Singleton	25	86.20%	100%	92.59%
State	24	81.48 %	100%	89.79%
Non-DP	77	73.08%	100%	84.44%
Mean	270	73.37%	91.87%	81.58%

Table V. Precision, recall and f-score of Java pattern role detection.

Java Pattern	***Role***	***# of files***	***Precision***	***Recall***	***f-score***
Adapter	Target	14	100%	23.08%	37.50%
	Adapter	80	84.71%	90.00%	87.27%
	Adaptee	51	54.44%	98.00%	70.00%
Singleton	Singleton	25	86.21%	100%	92.59%
State	Context*	0			
	State	2	100%	100%	100%
	ConcreteState	22	80.00%	100%	81.97%

* Role context has no data since it was not present in PMART.

Table VI. Precision, recall and f-score of C++ pattern detection.

C++ Pattern	*Number of files*	*Precision*	*Recall*	*f-score*
Adapter	11	68.75%	100%	81.48%
Singleton	9	90.00%	100%	94.73%
State	11	91.60%	100%	95.65%
No DP	10	90.00%	100%	94.74%
Total	41	83.33%	100%	90.91%

Table VII. Comparison of our Approach with Other Related Design Pattern Detection Methods

Design Pattern	*Article*	*Precision*	*Recall*	*f-score*
Adapter	Uchiyama [20]	**100%**	90%	N/A
	Pandey [4]	N/A	N/A	N/A
	Tsantalis [18]	N/A	**100%**	N/A
	Our Approach	70.12%	85.92%	**77.22%**
Singleton	Uchiyama [20]	60%	**100%**	N/A
	Pandey [4]	**88%**	95.65%	91.67%
	Tsantalis [18]	N/A	100%	N/A
	Our Approach	86.20%	**100%**	**92.59%**
State	Uchiyama [20]	50%	**100%**	N/A
	Pandey [4]	N/A	N/A	N/A
	Tsantalis [18]	N/A	95.73%	N/A
	Our Approach	**81.48%**	**100%**	**89.79%**

Note: The use of bold font in the table denotes the highest measurement level.

Table IV presents the overall precision, recall and f-score for the detected patterns in Java, summarizing the detection performance. Table V provides a detailed breakdown of the precision and recall values for each specific pattern role. Table VI displays the precision and recall of the C++ design pattern detection. Table VII shows how our design pattern detection compares to related work methods. For further analysis of the raw data, please refer to our GitHub [23].

Table IV shows that our Java detection technique successfully identified design patterns with high precision. The Singleton design pattern had the highest precision at 86.20%, closely followed by the State design pattern at 81.48%. Although the Adapter pattern has a slightly lower precision of 70.12%, it is worth noting that it had a larger dataset than the Singleton and State patterns. Regarding recall, both the Singleton and State patterns hit the mark with a perfect 100%, while the Adapter pattern scored 85.92% due to a handful of false negatives. In terms of f-score, Singleton had the highest percentage at 92.59%, followed by State at 89.79% and Adapter at 77.22%. Furthermore, our technique showed low susceptibility to overfitting, as shown in Table IV, with precision and f-score values of 73.08% and 84.44%, respectively, for the 'no design pattern' category.

All detected design patterns exhibited higher recall values than precision values. However, as shown in Table V, the Target role had a low recall of 23.08%, mainly due to plenty of false negatives associated with the Target role. Similarly, the Adaptee role had a precision of 54.44%, primarily because of a relatively high number of false positives where the State role was predicted instead of the correct Adaptee role. Despite these challenges, our metric technique demonstrated high accuracy in identifying the Adapter, Singleton, and State patterns.

Looking at individual role detections in Table V, the roles of Target, State, Singleton, Adapter, and ConcreteState demonstrated the highest precision. However, it is important to note the small sample size for the Target and State roles of only two true positives and no false positives may affect this result. Consequently, the Singleton and Adapter roles were the most accurately detected, aligning with our overall Java design pattern detection results. The Adaptee role, on the other hand, showed lower accuracy due to a few false positives.

In our C++ results, as seen in Table VI, our metric technique effectively identified the State, Singleton, and 'no design pattern' patterns with high precisions of 91.60%, 90.00%, and 90.00%, respectively. The Adapter design pattern had a precision of 68.75%, which is only slightly lower than our Java results. Notably, the State pattern was detected more accurately in C++ (91.60%) compared to Java (81.84%). Drawing from both the Java and C++ results, we can answer our first research question (RQ1): C++ detection achieved higher precision in detecting the State and Singleton design patterns, while Java detection showed marginally better precision for the Adapter patterns. Regarding recall and f-score, our C++ outperformed our Java detection across all design patterns.

To further evaluate the effectiveness of our approach, we compared it to other relevant methods used for Java design pattern detection. Table VII presents the average precision, recall, and f-score for the three design patterns across different techniques. Our method outperformed the others regarding f-score for all design patterns and achieved the highest precision specifically for the State design pattern. Notably, when comparing datasets, it is worth mentioning that our Adapter dataset consisted of 144 files, compared to the 11 programs that Uchiyama et al. analyzed (no information about their number of files).

Remarkably, despite not utilizing machine learning models, our approach achieved a precision that was only 2% lower than the PLM work by Pandey and Horkoff [4]. Additionally, our method surpassed the work of Uchiyama et al. in detecting both Singleton and State design patterns. In comparing our results with previous research (RQ2), we conclude that our approach achieved the highest precision, recall, and f-score for Singleton and State design patterns compared to related methods. These results demonstrate the significant potential of our metric-based approach, as it outperforms other detection methods in terms of precision, recall, and f-score.

RQ1-Summary: Our results showed that C++ achieved higher precision in detecting Singleton and State design patterns. Furthermore, C++ surpassed Java in both recall and f-score. Nevertheless, given the limited quantity and nature of the C++ repositories analyzed, which lack industry-specific programs, it is clear that additional research on C++ is necessary for a more comprehensive answer to RQ1.

RQ2-Summary: The proposed metric-based approach demonstrated better performance in terms of f-score for both Singleton and State design patterns. It achieved a higher State precision of 31.48% compared to the next best approach and exhibited perfect recall for both Singleton and State design patterns. The approach performed comparably to machine learning models across all design patterns, suggesting our model as a simpler, yet effective, alternative for design pattern detection in software code.

V. Discussion

In this section, we discuss noteworthy aspects of our current work and results.

Inconsistencies in the implementation of certain roles. Our investigation of software design metrics unveiled inconsistencies in the implementation of Target and Singleton roles. More specifically, we observed that the PMART source code for the Target role primarily included either abstract methods or interfaces but rarely both. Regarding the roles with only two questions, we determined a threshold requiring correct answers to both questions, aiming to minimize false positives. However, this threshold inadvertently led to an increase in false negatives in detecting the Target role, given that only a handful of Target source code files featured both abstract methods and interfaces.

A parallel scenario appeared in the case of the Singleton metrics, where not all files contained private constructors, with some being public. To address this inconsistency, we established a threshold for roles involving three questions, requiring two of the three to be satisfied. However, this threshold resulted in certain Adaptee code being misidentified as Singleton roles due to the presence of both static fields and static methods. For future work, we aim to develop more specific metrics, questions, and thresholds to reduce Target and Singleton false positives.

Analyzing a small number of files. Our approach relied on a limited number of pre-labelled examples of the Adapter, Singleton, and State design patterns to determine the relationship between metrics and patterns. However, the size of our labelled examples is smaller than what we would typically need for training a machine-learning model. For instance, in our Java dataset from PMART, we had 144 examples of Adapters but only 24 and 25 examples of Singletons and States, respectively. Given that our C++ dataset consisted of only 41 analyzed files, which is less than a quarter of the size of the Java dataset, more research would be needed for a more comprehensive comparison.

Interestingly, we observed higher accuracy in detecting the Singleton and State patterns, possibly due to the correlation with the smaller dataset. These results suggest that our approach can provide a reasonable solution for detecting design patterns when only a few examples are available. However, further analysis is required for a more industrial context to assess its applicability on a larger scale.

The necessity of compilable code. Our current approach requires the code to be compilable for successful metric extraction using the Java and C++ AST libraries. While this was generally not an issue in the PMART examples, we encountered a few cases in the Netbeans repository where Adapter files needed to be decompiled from Class files to Java files. Consequently, a few of these decompiled files had compilation issues due to syntax errors. Analyzing intricate segments of industry code can prove to be a greater challenge when each file must be compilable. In future work, exploring techniques for extracting metrics will be necessary even for code that do not compile.

Evaluation of the optimal threshold requires further investigation. Our approach utilizes threshold logic to determine the presence of design patterns based on extracted metrics and related questions. However, in our analysis, we only considered two thresholds: for roles with two questions, all questions needed to be met; for roles with three questions, it was sufficient with two out of three. Originally, we planned to analyze all potential thresholds to find the optimal one, but due to time constraints, other research tasks took priority. Therefore, future work should focus on thoroughly analyzing the optimal threshold.

Difficulties with the NCOF metric. Challenges with the NCOF metric emerged during the extraction and generation of our results. We encountered difficulties in accurately extracting the NCOF metric (number of other classes with a field of its own type). The primary issue lies in the interpretation of this metric. Our approach focused on counting the number of other classes with a field of its own type within a single Java file. However, an alternative understanding could involve traversing an entire source folder and analyzing all classes to identify fields of a specific type. Due to time constraints, we opted against this approach. In future research, seeking clarification from experts, such as, Uchiyama et al. could provide valuable insights into the interpretation and correct implementation of the metrics.

Adjusting Metrics for C++ Analysis. When we analyzed the C++ files, we noticed that some metrics we used for Java needed to be changed to work well with C++ structures. We did not remove any metrics but had to adjust the NOAM and NOOF metrics, which relate to abstract methods and object fields, respectively. Abstract methods and object fields do not have an exact equivalent in C++. However, virtual functions and data members play similar roles, so we updated these metrics to better fit the different language structures. Although the other metrics stayed the same, we had to make significant modifications to our extraction scripts to deal with the unique syntax of C++.

C++ files do not portray real production code. One limitation of our C++ analysis is that the analyzed source code consists of C++ examples showcasing different design pattern implementations, rather than real production code adhering to a design pattern. Consequently, how well our C++ prediction would perform on industry-level code remains to be determined.

During our analysis of C++ examples without design patterns, we encountered challenges when processing large header files that contained class and field definitions. This highlighted the need for a more robust approach to accurately detect C++ design patterns in an industrial context, particularly by improving the traversal of header and CPP files. Future work in C++ design pattern detection should focus on implementing enhanced methods to effectively handle the intricacies of C++ industry-level code.

Performance comparison with Uchiyama et al. study. Our results showed better performance in detecting the Singleton and State design patterns compared to the work by Uchiyama et al., which was particularly intriguing given our similar choice of metrics. However, the application of these metrics differed: while Uchiyama et al. incorporated the metrics into a convolutional neural network for design pattern detection, we opted for threshold-based detection. Another noteworthy distinction lies in our choice of datasets: we utilized all nine programs in PMART, whereas Uchiyama et al. analyzed three programs: JUnit (also in PMART), the Java framework, and SpringSource. In comparing our results, it is evident that Uchiyama et al. structured their analysis around the number of programs while we focused on the number of files. They examined two small and seven large Adapter programs for their design pattern datasets, six small and six large Singleton programs, and two small and six large State programs.

On the other hand, we analyzed nine programs from PMART with varying sizes. However, it remains unclear which PMART's repositories would be classified as small or large under the categorization logic of Uchiyama et al. In future work, applying our methodology to their dataset would be worthwhile, enabling a more direct comparison of results.

In conclusion, our findings underscore the importance of considering implementation variations when applying software design metrics and suggest the need for more tailored metrics to accurately identify all software design patterns and roles in industry level code.

A. Threats to validity

This section discusses threats to the validity of the study. Our structure follows the validity of experimental results outlined in Cook and Campbell [26].

A. Conclusion validity

We used precision, recall, and f-score performance measurements in our study to evaluate our metric-based design pattern detection model. These measures facilitated comparisons with previous research and enabled us to answer our research questions using the data collected. A larger sample size for both C++ and Java files would be beneficial to arrive at more robust conclusions. Additionally, incorporating industry-level code could yield more nuanced understandings. Nonetheless, our conclusions are well-grounded based on our analysis of PMART and C++ GitHub repositories.

We followed a consistent methodology throughout our study, which can be readily reproduced using automated metric extraction scripts in future research. However, it should be noted that predicting design patterns based on the extracted metrics is not fully automated yet and still requires manual threshold evaluation, a step we aim to automate in future research. To use our method, one must pull our GitHub source code [23], add file paths to analyze, and run the application.

One potential area for improvement is our interpretation of the NCOF metric. Although we consider this metric as the count of other classes with the field type of the class under review, it frequently appears as zero in our extracted data. Since the extraction script runs through all classes in the source file, verifying the metric personally can be quite challenging. Further exploration into NCOF is warranted, and clarification from Uchiyama et al. [21] might be beneficial for accurate metric extraction.

Our Java results are relatively generalizable, as we analyzed over 250 PMART source files, a dataset slightly larger than those used in similar design pattern detection studies. However, more in-depth analysis with a substantially larger labelled sample size is necessary to ensure broader applicability to industry-level code.

On the other hand, our C++ results, due to the nature of the analyzed repositories, have limited generalizability and primarily serve as a preliminary assessment of our C++ design pattern detection approach. To solidify our C++ findings and enhance their generalizability, future research involving a more representative sample of C++ code is needed.

B. Internal validity

In our study, we utilized a straightforward method for design pattern detection: a fixed threshold requiring accurate responses to either 2 out of 3 or 2 out of 2 design pattern questions. This threshold approach, while simple, may have resulted in an increased number of false positives, as it does not account for potential variations in different design pattern implementations. Consequently, the precision of our results could have been affected.

An alternative and potentially more accurate approach would be to utilize an adaptive threshold that adjusts according to the specific implementation of each design pattern. Such an adaptive threshold could enhance the accuracy of our detection strategy. To implement this adaptive threshold, future studies could explore alternative detection techniques, such as classification methods. Machine learning classification methods, such as random forest used by Uchiyama et al. [21], should be considered.

Despite the potential improvements mentioned above, it is important to emphasize that using fixed thresholds was a deliberate choice to facilitate comparisons with previous studies that employed machine learning models and our metric-based approach for design pattern detection.

C. External validity

To establish a solid baseline for our study, we deliberately utilized an existing repository previously used in several studies. This approach allowed for a direct comparison of our Java performance outcomes with those from earlier studies. However, it is important to recognize that our sample selection from the Java and C++ repositories may not fully represent the broader industry.

The Java repository we selected provides high-quality code examples from various areas of Java programming, an essential factor in ensuring the severity and relevance of our findings. However, there is a potential for bias due to the specific focus of the Java repositories and their extension-based projects. This focus could limit the generalizability of our findings to other programming languages and standalone projects.

The selected C++ repositories, while providing perfect examples of C++ design patterns, have an inherent flaw of reduced generalizability compared to our Java findings. We also encountered difficulties in detecting industry-level C++ code with multiple header files. Further analysis, closely aligned with industry-level code, is required to draw more definitive conclusions regarding our C++ detection.

We contemplated analyzing industrial projects to provide a more realistic representation of current industry practices. However, this approach brought unique challenges, such as a lack of comparative studies and few known design patterns. These factors could diminish the robustness of our findings and limit their applicability to other contexts.

Ultimately, we chose the PMART repository to ensure our results were trustworthy, comparable, and as externally valid as possible given these constraints. Future studies might consider broadening the sample to include other repositories or industrial projects to address these limitations and extend the external validity of our findings.

VI. CONCLUSION

In conclusion, our study demonstrates the effectiveness of our metric-based approach for design pattern detection, showcasing strong precision, recall, and f-score values without relying on machine learning or programming language models. Our approach offers a practical and accessible alternative to enhance software quality and maintainability.

We have shown that metrics can effectively detect design patterns in both C++ and Java, achieving comparable or even higher accuracy than machine learning or programming language models. However, we acknowledge that further refinement of metrics and questions is necessary to improve accuracy and applicability. Additionally, exploring hybrid approaches that combine metrics and machine learning holds promising potential for future research.

To address our challenges, we recommend refining metrics, developing new metrics for a wider range of design patterns, and devising methods to extract information from uncompilable code. Collaboration with industry partners can provide valuable insights into real-world C++ design patterns and help tackle specific challenges in that context. Furthermore, expanding analyses to include a broader set of design patterns can deepen the understanding of using metrics in design pattern detection.

These future research directions will advance the field of design pattern detection and contribute to more sophisticated software analysis and quality assurance techniques.

REFERENCES

[1] Chihada, Abdullah & Jalili, Saeed & Hasheminejad, Seyed Mohammad Hossein & Zangooei, Mohammad Hossein. (2014). Source code and design conformance, design pattern detection from source code by classification approach. Applied Soft Computing, doi: 10.1016/j.asoc.2014.10.027.

[2] Caporuscio, Mauro & Toma, Marco & Muccini, Henry & Vaidhyanathan, Karthik. (2021). A Machine Learning Approach to Service Discovery for Microservice Architectures, doi: 10.1007/978-3-030-86044-8_5.

[3] C. Kramer and L. Prechelt, "Design recovery by automated search for structural design patterns in object-oriented software," Proceedings of WCRE '96: 4rd Working Conference on Reverse Engineering, Monterey, CA, USA, 1996, pp. 208-215, doi: 10.1109/WCRE.1996.558905.

[4] Sushant Pandey, Jennifer Horkoff, Miroslaw Staron (2023). Utilizing a Pre-Trained Language Model for Effective Design Pattern Recognition in Software (In Submission)

[5] Ptidejteam, "Ptidej team, PMART" Design Patterns - Ptidej Team, 22-Apr-2013. https://www.ptidej.net/tools/designpatterns/.

[6] A. Shvets, "Refactoringguru/design-patterns-CPP: Design pattern examples in C++," GitHub, 17-Mar-2019. https://github.com/RefactoringGuru/design-patterns-cpp.

[7] D. Nesteruk, "Apress/design-patterns-in-modern-CPP: Source code for 'design patterns in modern C++' by Dmitri Nesteruk," GitHub, 15-Apr-2018. https://github.com/Apress/design-patterns-in-modern-cpp.

[8] J. Guan, "Yogykwan/design-patterns-CPP: 《大话设计模式》C++实现," GitHub, 25-Dec-2016. https://github.com/yogykwan/design-patterns-cpp.

[9] J. Vojvoda, "Jakubvojvoda/design-patterns-CPP: C++ Design Patterns," GitHub, 04-Sep-2016. Available: https://github.com/JakubVojvoda/design-patterns-cpp.

[10] J. Wu, "Downdemo/design-patterns-in-CPP17: C++17 implementation of 23 GOF design patterns for zero memory leaks using smart pointers.," GitHub, 08-Dec-2019. https://github.com/downdemo/Design-Patterns-in-Cpp17.

[11] L. Jianhao, "Liu-Jianhao/CPP-design-patterns: C++设计模式，" GitHub, 30-Sep-2018. https://github.com/liu-jianhao/Cpp-Design-Patterns.https://github.com/downdemo/Design-Patterns-in-Cpp17

[12] Pezy, "Pezy/DesignPatterns: Design Patterns for C++.," GitHub, 30-Jul-2017. https://github.com/pezy/DesignPatterns.

[13] R. Pang, "Rhyspang/CPP-design-patterns: C++设计模式," GitHub, 30-Mar-2020. https://github.com/rhyspang/CPP-Design-Patterns.

[14] Light-City, F. *Light-city/cplusplusthings: C++那些事, GitHub*. 14-June-2019 https://github.com/Light-City/CPlusPlusThings

[15] G. Antoniol, R. Fiutem, and L. Cristoforetti. Design pattern recovery in object-oriented software, Proceedings of the 6th International Workshop on Program Comprehension, 1998, pp.153-160.

[16] Lucia, A.D., Deufemia, V., Gravino, C., & Risi, M. (2007). A Two Phase Approach to Design Pattern Recovery. 11th European Conference on Software Maintenance and Reengineering (CSMR'07), 297-306.

[17] Arif Nurwidyantoro, Truong Ho-Quang, and Michel R. V. Chaudron. 2019. Automated Classification of Class Role-Stereotypes via Machine Learning. In Proceedings of the Evaluation and Assessment on Software Engineering (EASE '19). Association for Computing Machinery, New York, NY, USA, 79–88, doi: 10.1145/3319008.3319016.

[18] T. Ho-Quang, A. Nurwidyantoro, S. A. Rukmono, M. R. Chaudron, F. Froding, and D. N. Ngoc, "Role stereotypes in software designs and their evolution," Journal of Systems and Software, vol. 189, p. 111296,2022.

[19] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides and S. T. Halkidis, "Design Pattern Detection Using Similarity Scoring," in IEEE Transactions on Software Engineering, vol. 32, no. 11, pp. 896-909, Nov. 2006, doi: 10.1109/TSE.2006.112

[20] Dhasarathy Parthasarathy, Cecilia Ekelin, Anjali Karri, Jiapeng Sun, and Panagiotis Moraitis. 2022. Measuring design compliance using neural language models: an automotive case study. In Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE 2022). Association for Computing Machinery, New York, NY, USA, 12–21, doi: 10.1145/3558489.3559067.

[21] Uchiyama, Satoru & Kubo, Atsuto & Washizaki, Hironori & Fukazawa, Yoshiaki. (2014). Detecting Design Patterns in Object-Oriented Program Source Code by Using Metrics and Machine Learning. Journal of Software Engineering and Applications. 7, doi: 10.4236/jsea.2014.712086.

[22] Issaoui, I., Bouassida, N. & Ben-Abdallah, H. Using metric-based filtering to improve design pattern detection approaches. Innovations Syst Softw Eng 11, 39–53 (2015). doi: 10.1007/s11334-014-021-3

[23] N. Dunlop "NilsDunlop/Thesis" Github, 21-04-2023. https://github.com/NilsDunlop/Thesis

[24] D. van Bruggen, "Tools for your java code," JavaParser, https://javaparser.org/, 03-05-2023.

[25] L. Jaquemet, "Clang: A C language family frontend for LLVM," Clang C Language Family Frontend for LLVM, https://clang.llvm.org/, 03-05-2023.

[26] T. D. Cook, D. T. Campbell, Quasi-Experimentation: Design and Analysis Issues for Field Settings. Houghton Mifflin, 1979.