Detecting Design Patterns As Described By An Explicit Specification

Ed van Doorn
Faculteit Bètawetenschappen
Open University of the Netherlands
Heerlen, the Netherlands
ed.vandoorn@ou.nl

Sylvia Stuurman
Faculteit Bètawetenschappen
Open University of the Netherlands
Heerlen, the Netherlands
sylvia.stuurman@ou.nl

***Abstract*—Teaching design patterns in higher education enhances coding skills and improves software quality. However, manually correcting their work and detecting whether the students have correctly implemented the design patterns is time-consuming and would benefit from automated tools. There exist tools that can automatically detect design patterns in code. However, these tools rarely provide an explicit specification of what each pattern should look like. As a consequence, these tools are not usable in an educational environment. When using a tool to detect design patterns in an educational environment, there are several important considerations to keep in mind. Firstly, the tool should use an explicit specification of each design pattern. Secondly, as a teacher, you should be able to adjust the specifications. Third, the tool should be accurate, meaning it should produce few to no false positives or false negatives when identifying patterns in the students' code. Finally, the tool should do this computationally efficiently. In this article, we present a new tool for detecting design patterns in code. Our tool is capable of recognizing 20 of the most commonly used design patterns, including those described by the 'Gang of Four'.**

***Index Terms*—Design Patterns, Detection, Specification**

I. Introduction

Design patterns have shown a positive impact on the maintainability of software [14]. As a result, teaching design patterns has become an important topic in software engineering education at the university level.

Design patterns were originally described in the book written by 'the Gang of Four' (GoF) [5]. However, the book primarily uses natural language, class diagrams, and examples in C^{++} source code to describe the patterns, without providing a clear and unambiguous specification. This can make it challenging for teachers to assess assignments that require the use of design patterns, and can lead to time-consuming discussions with students over the correct implementation of these patterns. As a result, there is a need for tools that can help automate the process of detecting design patterns in code, and provide a more objective way of evaluating students' work. Although automated tools for detecting design patterns exist, many of them lack transparency in their detection methodology. In an educational environment, it is essential for teachers to have access to tools that can accurately detect design patterns used in student code while being customizable to only identify specific patterns. Additionally, the tool should have a low rate of false positives and false negatives to ensure accurate pattern detection. Finally, the tool should be computationally efficient to enable teachers to efficiently evaluate large amounts of code.

To address this issue, we have developed a more precise specification of the 'GoF' patterns [4], which can be customized to suit the preferences of individual teachers. In this article, we will demonstrate the accuracy of our tool, which is based on this specification. We assess our tool using a test set that includes all the GoF design patterns, and Java sources that were found 'in the wild'. By using this approach, we can ensure that our tool accurately detects design patterns in a variety of contexts and scenarios.

The contributions of this article are:

- An algorithm based on explicit specifications that can detect 20 GoF design patterns.
- A reusable test set for the GoF design patterns.
- An assessment where we demonstrate the efficiency and accuracy of our approach.

The remainder of this paper is organized as follows. Section II describes related work. In Section III, we outline the specifications we use and describe our tool in detail. In Section IV, we present the results of our tool's verification and validation. Finally, Section V contains our conclusions, a discussion of our work, and suggestions for future research.

II. Related Work

Hadis Yarahmadi et al. have written a comprehensive systematic review of design pattern detection methods that are published in 112 articles between 2008 - 2019 [13].

None of the authors describe the specifications of design patterns that their software can detect. Nevertheless, they list the design patterns that can be detected by their software.

The detection methods can be divided into two groups: *exact* and *inexact*. A detection method is *inexact* if, after detecting a part of the design pattern, the software can report that the design pattern has been recognized.

Inexact detection methods will produce more false positives than exact detection methods.

In an educational environment, we want only to detect completely implemented design patterns. So, we focus on

DOI reference number: 10.18293/SEKE2023-095

an exact detection method, using an explicit specification of design patterns.

We only found a few works after 2019 complementing the review from Yarahmadi et al. We will describe them next.

For several years, machine learning has been used to detect design patterns. An example is a recent study by [8], in which fifteen features of code are defined, such as class names, interfaces, methods, parameters, and a number of variables in a method. These features are used to construct word vectors [7] of n-grams, which are groups of n consecutive words.

Nazar's approach is based on an ensemble [10] of randomized decision trees, which classifies a design pattern. These ensembles are combined with a supervised learning algorithm.

To train a supervised neural network, 1300 Java files from Github were selected. Every Java file either contained one of twelve design patterns or did not contain any design pattern. As a benchmark, 1039 Java files from P-Markt[1] were used. The benchmark was used for calculating the precision and recall.

This resulted in an average precision of 80% and an average recall of 79%.

III. Our approach to design pattern detection

A. Basis of our approach

Our approach builds upon two key foundations. Firstly, it improves upon an earlier detection tool that we developed [2]. Secondly, it is based on our construction of explicit specifications for design patterns [4].

The earlier detection tool utilized exact subgraph matching and relied solely on the names of the participating classes and their relationships. Consequently, it could only detect design patterns that were entirely defined by these elements, providing static decidability [3]. For instance, the Adapter pattern could be detected using this algorithm as it was defined entirely by the names of its participating classes and their relationships. However, the Singleton pattern cannot be detected through an algorithm that offers only static decidability, as the keywords 'private' and 'static' are essential to define this pattern.

Our presented tool addresses this limitation by utilizing an explicit specification that considers multiple features beyond class names and class relationships.

Our explicit specification for design patterns provides a complete definition for all 23 GoF design patterns, with the exception of the Strategy and State patterns, as they cannot be distinguished based solely on their structure and must be identified based on their intended purpose or meaning [4].

Based on our explicit specification for design patterns, our approach does exact subgraph matching. This method provides high precision in detecting patterns. Precision is defined as the number of true positives divided by the sum of true positives and false positives. In an educational setting, it is crucial to minimize the number of false positives, ideally to zero. This is because false positives can erode confidence in the detection software.

[1]http://www.ptidej.net/tools/designpatterns

Using subgraph matching based on the explicit specifications, we can significantly reduce the occurrence of false positives. However, false positives can still occur if the templates match the Java sources but the functionality of the methods does not comply with the intent of the design pattern.

B. Explicit specification of design patterns

The starting point of our specification is, of course, the book of Gamma et al. [5], which describes design patterns using natural language, class diagrams, and examples of C^{++} code.

We summarize our explicit specifications. Details are given in [4]. Important specification elements are classes, attributes, operations, relationships, and modifiers.

The keyword 'interface' does not exist in C^{++} and OMT [12], which is used by Gamma et al. as a modeling technique but does exist in Java and UML. In many cases, an abstract class is equivalent to an interface. For the exceptions, 'interface' is added to the specification language of design patterns.

Dependency is a specification element that is used in Gamma et al. to denote class creating an object of another class [4].

In source code 1-N associations, aggregates and a composites are identically implementated, so they are described as 1-N associations [4].

An example of the explicit specification is Listing 1.

Listing 1. Template description of the Singleton pattern

```
<template name="Singleton">
  <class name="Singleton">
    <attribute name="uniqInstance"
       type="Singleton" modifier="private"
       isStatic="true"/>
    <operation name="Singleton" modifier="private"/>
    <operation name="getInstance" isStatic="true"/>
  </class>
</template>
```

C. Algorithm

The algorithm for detecting design patterns takes Java source code files as input, along with a file named 'templates.xml' that contains explicit specifications for each design pattern.

The Java sources are parsed by a parser generated by cup[2] because of the produced parser's speed [11], and the availability of a grammar of Java[3]. The Java parser generates information about the classes, interfaces, and their relationships. This results in a file named 'inputSystem.xml' with one template that has the same format as the templates in 'templates.xml'. These two XML-files can easily be parsed by a SAX[4] parser, which is part of Java.

For each design pattern in *templates.xml*, we search for a corresponding design pattern *inputSystem.xml* by a recursive "depth-first search" using two phases. Phase 1: In each call of the recursive "depth-first search", an edge in the template we

[2]http://www2.cs.tum.edu/projects/cup/index.php

[3]https://github.com/joewalnes/idea-community/tree/master/tools/lexer/jflex-1.4/examples/java

[4]http://www.saxproject.org/

TABLE I
RESULTS FOR THE TEST SET CODE

Script-number	Number of classes	Lines of code	number of design patterns	Total process time (sec)
1	15	298	5	3.0
2	18	318	6	3.6
3	5	97	2	1.3
4	42	771	11	8.2
5	8	164	3	1.8
6	3	62	1	0.8
7	8	168	1	1.9

are searching for, is matched with an edge in *inputSystem.xml*. The search ends when all edges are matched. Phase 2: Each class in the design pattern we are searching for, is compared with the corresponding class in *inputSystem.xml* by trying to match the attributes and methods of both classes.

To detect relations in Java sources, we adopt the definition of an association given by Guéhéneuc [6], neglecting run-time properties in his definition, because we use static detection. The definition becomes: *An association between class A and B exists when an instance of class A can send a message to an instance of class B. Instances of classes occur as a field, array field, collection, parameter, and local variable.*

IV. ASSESSMENT OF THE APPROACH

To assess the accuracy of our approach, we apply it to both a test set that we have constructed for the purpose of evaluating, and source code that is available "in the wild". The research questions we will answer are: *When given the design pattern specifications from section III-B*:

RQ1 *What is the accuracy of our detection software?*

RQ2 *What is the efficiency of our detection software?*

The accuracy of the detection software can be determined by the ratio of correctly identified and rejected design patterns to the total number of searched design patterns. In cases where a design pattern is not detected, we will investigate the reasons for its failure to be detected. Furthermore, we will also address the occurrence of false positives in the detection results.

The efficiency is expressed as the total process time. During processing, Java files are read, parsed, and searched for design patterns.

The assessments are conducted on a PC with hardware characteristics Core™ i5-6400 CPU @ 2.70GHz x 4.

A. The test set

We have constructed a new test set comprising seven Netbeans projects, each containing a total of 23 distributed design patterns that align with the explicit specifications from section III-B. The test set has been designed to include classes that are associated with multiple design patterns, which increases the complexity of the search process. Furthermore, we ensured that some of the design patterns are not exact replicas of the GoF patterns. For instance, the Abstract Factory design pattern comprises three ConcreteFactories instead of the traditional two.

The results for running our tool on the test set are in Table I. The data shows that the design patterns can be processed within seconds. So the software is fast enough for an educational environment and therefore *efficient*.

Moreover, our proposed tool shows to be *accurate* because all design patterns, except the Facade pattern, can be detected. However, the detection software cannot distinguish between the State and Strategy pattern.

B. Java sources in the wild

We found two repositories for educational purposes with Java sources containing design patterns we will denominate RameshMF[5] and sourcemaking[6]. These repositories offer for each design pattern a map with java sources. We made minor adjustments for the declaration of an attribute, keywords that we do not use, such as enum, module, and related keywords (as described in Section III-C)

The sources of RameshMF contain 18 of the 20 design patterns. This repository has no examples of the design patterns Interpreter, Mediator, Memento, and Visitor. For these design patterns, we used the sources of sourcemaking.

The results are shown in Table II.

TABLE II
RESULTS RAMESHMF & SOURCEMAKING

	Design pattern	Lines of code	Detected	Total process time (sec)
RameshMF				
1	Abstract Factory	567	Y	2.9
2	Adapter	330	Y	2.2
3	Bridge	759	N	4.1
4	Builder	430	N A	1.9
5	Chain of Responsibility	437	N A	1.9
6	Command	596	N	3.4
7	Composite	974	N	4.1
8	Decorator	404	N	3.9
9	Factory-pattern	932	Y	12.7
10	Flyweight	472	N A	2.3
11	Iterator	337	N A	1.7
12	Observer	591	N A	1.7
13	Prototype	620	Y	3.3
14	Proxy	275	N	2.1
15	Singleton	313	Y A	1.3
16	State	118	N	2.4
17	Strategy	61	N	1.9
18	Template Method	369	Y	3.0
sourcemaking				
19	Interpreter	147	N	1.0
20	Mediator	103	N	1.2
21	Memento	57	Y A	0.5
22	Visitor	82	N	1.8

The meaning of the values in column *Detected* is:

- A (Adjusted): A small change was made in the detection code to make detection possible. Examples: declaring an attribute public, and removing an enum definition.

[5]https://github.com/RameshMF/gof-java-design-patterns

[6]https://sourcemaking.com/design_patterns/

- N(o): The implementation of the design pattern matches the intent but not the class diagram as given by Gamma et al. [5]. Therefore, the implementation does not match the explicit specification of design patterns.
- Y(es): Detected without any problem.

The values in column *Lines of code* are the numbers of lines of Java code in the map containing the design pattern.

The results show the accuracy of our software because seven (5 * Y + 2 * Y A) detected design patterns are implemented according to their descriptions, and the implementations of the other 15 design patterns differ from their descriptions. For an educational environment, these results are *accurate* because the implementation of a design pattern has to match fully and not partially. The results are *efficient* because the total process time is at most 12.7 seconds.

V. Conclusions

Our main goal is to show that it is possible to build a design pattern detection tool using an explicit implementation.

The limitation of our tool is that we can make no distinction between State and Strategy patterns because that is impossible using static detection. Also, we chose not to include parts of the grammar that are less relevant to students, such as the keyword module, lambda expressions, and enum definitions.

Another limitation is that the Facade pattern cannot be detected. The specification of the Facade pattern in terms of classes and relations is simply too vague, too broad, to be usable.

Teachers could adjust the specification of the patterns to their own needs. The specification of design patterns is easily adaptable because it is written in XML and documented. Imaginable is, to make it possible to choose which language constructs are in- or excluded.

For verification, we used a test set with 22 design patterns. The positive results of the applied test set contribute to confidence in the specification of the design patterns and the qualities of the software.

However, detection of the couple State and Strategy patterns implies a false negative.

But, if the source code of a method does not contain any statement then the intent of the class and template is not realized.

So, the detection of design patterns without false positives is not guaranteed.

For validation, we used several sources from the Internet. Design patterns that are present could sometimes not be recognized. The implementation of these patterns matched the intent but not the specification of the patterns.

The detection software is useful for an educational environment because the student's elaboration of assignments has precisely to comply with the assignment.

The answers to our research questions are as follows:

1) The *accuracy* of the detection software is that all GoF design patterns, except the Facade pattern, can be detected when they fully match the defining template of the design pattern. No distinction can be made between the State and Strategy pattern. The software is not appropriate for searching for implementations of design patterns that only comply with the intent of a pattern.
2) The *efficiency* of the detection software depends on the number of lines of code and the design pattern. Java sources with less than 1000 lines of code are processed for all GoF design patterns within 12 seconds.
3) The new descriptions of design patterns are useful in an educational environment because a teacher may specify how design patterns should be implemented, and may allow only implementations that adhere to this specification.

In the future, we will try to improve the ease with which teachers may adapt the specification. Also, we may work on the feedback that the tool can give. Of course, we will also find out whether our tool can be really helpful, for teachers and/or students.

VI. Acknowledgement

I thank Prof. Dr. Tanja Vos for her advice, discussions, and support.

References

[1] N. Bozorgvar, A. Rasoolzadegan and A. Harati, Probabilistic detection of gof design patterns, *The Journal of Supercomputing* (Aug 2022).

[2] E. van Doorn, Supporting design process by automatically detecting design patterns and giving some feedback, Master's thesis, Open University (8 2016).

[3] E. van Doorn, S. Stuurman and M. van Eekelen, Static detection of design patterns in class diagrams, in *CSERC '19*, eds. E. Rahimi and D. Stikkolorum (Association for Computing Machinery (ACM), United States, nov 2019), pp. 79–88. 8th Computer Science Education Research Conference (CSERC'19), CSERC ; Conference date: 18-11-2019 Through 20-11-2019.

[4] E. van Doorn and S. Stuurman, Towards more precise descriptions of design patterns, in *Software Engineering Perspectives in Systems*, ed. R. Silhavy (Springer International Publishing, Cham, 2022), pp. 117–140.

[5] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software* (Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995).

[6] Y.-G. Guéhéneuc, A reverse engineering tool for precise class diagrams, in *CASCON*, 2004.

[7] T. Mikolov, I. Sutskever, K. Chen, G. Corrado and J. Dean, Distributed representations of words and phrases and their compositionality (2013).

[8] N. Nazar, A. Aleti and Y. Zheng, Feature-based software design pattern detection, *Journal of Systems and Software* **185** (2022) p. 111179.

[9] R. E. Neapolitan *et al.*, *Learning bayesian networks* (Pearson Prentice Hall Upper Saddle River, 2004).

[10] A. Rahman and S. Tasnim, Ensemble classifiers and their applications: A review, *International Journal of Computer Trends and Technology* **10** (04 2014).

[11] T. J. Parr, S. Harwell and K. Fisher, Adaptive ll(*) parsing: the power of dynamic analysis, in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, eds. A. P. Black and T. D. Millstein (ACM, 2014), pp. 579–598.

[12] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorensen, *Object-Oriented Modeling and Design* (Prentice–Hall, Englewood Cliffs, NJ, 1991).

[13] H. Yarahmadi and S. M. H. Hasheminejad, Design pattern detection approaches: a systematic review of the literature, *Artificial Intelligence Review* **53** (12 2020).

[14] F. Wedyan and S. Abufakher, Impact of design patterns on software quality: A systematic literature review, *IET Software* **14** (02 2020).