Static Detection of Design Patterns in Class Diagrams

Static Detection of Design Patterns in Class Diagrams

Ed van Doorn
The Hague University of Applied Sciences
The Hague
E.M.vanDoorn@hhs.nl

Sylvia Stuurman
Open University of the Netherlands
Heerlen
Sylvia.Stuurman@ou.nl

Marko van Eekelen
Open University of the Netherlands
Heerlen
Radboud University
Nijmegen
Marko.VanEekelen@ou.nl

ABSTRACT

Teaching Object-Oriented design on the class diagram level is often a cumbersome effort. Requiring the use of specific design patterns helps the students in structuring their design properly. However, checking whether students used the right design pattern can be a very time-intensive task due to the variety of possibilities of creating structure using design patterns on the high-level class diagrams. For the same reason, it is hard for students to check for themselves whether their solution fulfills the basic requirements that are required by the instructor with respect to the use of design patterns. Efficiency and the quality of design pattern education can be improved by automatic detection of design patterns in UML class diagrams. We introduce a new method to detect design patterns in class diagrams, together with a prototype of a tool that uses this new method. Using this tool, an instructor needs less effort to review solutions of design exercises since the tool can check the basic class requirements automatically. Consequently, an instructor can focus on the more high-level requirements that were set in the exercise and students can easier check for themselves whether their design satisfies the basic required properties on the pattern level. The method offers *static decidability* for those design patterns, that are identified by structural properties e.c. the names of the classes and their associations. It is *non-duplicating*. That is a specific occurrence of a design pattern is not reported multiple times. The method not only detects all 16 *static* Gang of Four design patterns without false positives or false negatives, but also it can detect redundant relations. Our tool contributes to the quality and efficiency of design pattern education, both for students and instructors.

CCS CONCEPTS

• **Software and its engineering → Design patterns**.

KEYWORDS

efficiency of learning and education, design pattern detection

CSERC '19, November 18–20, 2019, Larnaca, Cyprus

ACM ISBN 978-1-4503-7717-1/19/11…$15.00
https://doi.org/10.1145/3375258.3375268

ACM Reference Format:
Ed van Doorn, Sylvia Stuurman, and Marko van Eekelen. 2019. Static Detection of Design Patterns in Class Diagrams. In *Proceedings CSERC 2019 18-20 November 2019 Computer Science Education Research Conference Larnaca, Cyprus (CSERC '19), November 18–20, 2019, Larnaca, Cyprus.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3375258.3375268

1 INTRODUCTION

In many educational situations, automatically detecting design patterns in a class diagram would be of value. For instance, instructors could be supported by their evaluation of designs that students send in [17]. This support will save time for instructors. It also guarantees equal assessment and reduces the amount of monotonous work. It would be also useful as an aid in a class diagram editor for students, to check whether they have represented an intended design pattern. Feedback on simple mistakes would also be welcomed.

In an educational environment, a tool that uses UML class diagram as input and can detect design patterns is wanted. Automatically detecting design patterns based on structural properties can be generalized to automatically detect *any pattern* that consists of classes and their relations. For that purpose, obligatory larger *static* parts may be defined. The contributions of this paper are:

- The new theoretical concepts: static, and non-static design patterns, static decidability, generally complete, non-duplicating.
- A new subdivision of design patterns based on these concepts.
- A new method for detecting design patterns based on these concepts.
- A prototype of a tool that uses this method. This tool can detect all 16 Gang of Four static design patterns and is non-duplicating.
- Detecting design patterns that partially exist.
- Generating limited feedback.

The tool [1] is publicly available. Our paper is structured as follows. We define a new concept in section 2: *static* design patterns, that defines a set of design patterns that are completely defined by their class names and their relationships. We also define the concept of *static decidability* for algorithms that can detect *static* design patterns. The concept of *non-duplicating* is also defined in section 2. For the implementation of an algorithm that can detect all *static* design patterns in a UML class diagram, several requirements are wanted. First of all, it should make no errors and so, only detect true *static* design patterns. Templates for *static* design patterns should be easily definable. It should also detect permutations of design patterns and multiple occurrences of one design pattern only once, i.e. the algorithm should be *non-duplicating*.

[1] http://members.chello.nl/e.doorn1/DesignPatterns/static_decidability

In section 3, we describe earlier approaches to detect design patterns in UML class diagrams. Here, we show that none of these approaches meet our requirements. Some of these approaches can detect all *static* design patterns, but these approaches have shortcomings, e.g. they are not *non-duplicating*.

In section 4, we describe our approach in detail. We explain how the above requirements are fulfilled.

We describe the results of our prototype tool in section 5. We show that the implementation of our method results in a prototype that works in practice. The prototype can detect all 16 *static* Gang of Four [GoF] design patterns [9], because they are fully defined by their class/interface-names and their relations. The number of false positives and negatives, the speed of detection and the improvements of the detection method are discussed. It can read the XMI representation of a class diagram by ArgoUML, the drawing tool for class diagrams and reads templates for design patterns in XML-format. The results are compared to the result of other tools.

In section 6, we refer to articles about four subjects. Experiences about education at university level related to design patterns is clearly an important issue. Several approaches for detecting design patterns in source code are mentioned. Also, an approach to detect anti-patterns is referenced. Research on the relation of occurrences between design patterns and code smells is referenced.

In section 7, we describe our conclusion. The theoretical and practical advances are recalled. The results are summarized as well as the limited feedback.

In section 8, we show the limitations of our approach and give suggestions for future research. In particular, we mention *generally complete* detection algorithm, improved detection for the Abstract factory, improved feedback and measuring the quality of UML class diagrams.

2 STATIC / STATIC DECIDABILITY / GENERALLY COMPLETE / NON-DUPLICATING

In this section, we introduce a new subdivision of the set of all design patterns into two new complementary subsets: *static* and *non-static*. Each subset is related to a type of detection algorithm, that offers: *static decidability* or *generally complete*. The approaches to detect design patterns in section 3 are labeled by these new concepts.

2.1 New categorization for Design patterns and detection methods

Traditionally the set of all design patterns is subdivided by their use. Creational patterns are used to create complex objects. Behavior patterns are used to divide and assign responsibilities to classes and to describe the communication between objects. Structural patterns are used to associate classes to bigger structures [9].

A second subdivision is based on analyzing source code. Detection algorithms can use static analysis, during which the code is not running and during dynamic analysis when the code is running. The first subset of all design patterns is *static structural patterns*, which are detectable based on their classes and relationships. *Dynamic behavior patterns*, which are detectable based on the interaction between objects. This subset of design patterns can be detected by a combination of static and dynamic analysis. *Program-Specific patterns*, which are detectable based on specific keywords and code styles. This last subset of design patterns can also be detected by a combination of static and dynamic analysis [10].

A third subdivision resulted in PINOT, an automated detection tool, which uses the source code as input[15]. The subdivision consists of five subsets. The first subset is based on the programming language. Java provides the Observer and Iterator pattern. The second subset contains structure driven patterns, that are fully defined by the relationships between classes. The third subset contains behavior driven patterns. These patterns model behavior aspects. Examples are Singleton, Strategy, State, Factory method and Decorator. The fourth subset is the domain-specific patterns. The Interpreter and Command combined with the composite and visitor design pattern are used for specific languages. The fifth subset is used for general concepts. The Builder and Memento design pattern constitutes this subset. Their structure is detectable, but their behavior aspects are hard to detect. PINOT can detect structure and behavior driven patterns. PINOT could easily be extended to detect patterns, which are provided by the language. It would only be necessary to detect some keywords.

Another example of the third subdivision is described by Bernardi [5]. He uses a domain-specific language to define patterns and source code. A graph matching search method is used to detect design patterns. The tool can distinguish between the State and Strategy patterns which are structurally identical. The Singleton pattern is detectable and variants of design patterns are also detectable, but the recall and precision of the tool is not 100%.

The first subdivision is not helpful to create detection methods, because it does not give any information about easily identifiable characteristics of design patterns. The second and third subdivision are based on source code analysis but inspired us to give a subdivision for UML diagrams. When we first look at a class diagram of a design pattern, we see several classes and relationships, which concepts are easily identifiable. This leads to the following definitions.

Definition 2.1. A design pattern is *static* if it is completely defined by the names of their participating classes and their relationships.

The Adapter pattern is an example because the pattern can be defined by the names of the classes and their relationships, i.e., an association and one inheritance relation (see Figure 1).

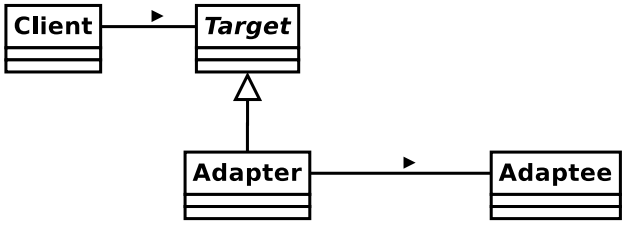

Figure 1: Adapter pattern

A Singleton pattern is not a *static* design pattern, as explained after the next definition.

Definition 2.2. A design pattern is *non-static*, if it needs more characteristics than names of their participating classes and relationships to be defined.

An example is the Singleton pattern. The Singleton class needs a static operation which returns the unique object and a static attribute, which contains the unique object or a null-value.

From an educational point of view, one could state that modeling design patterns which are *non-static* needs more attention and effort than modeling design patterns which are *static*.

Definition 2.3. A detection algorithm offers *static decidability*, if it can detect all *static* design patterns.

Such an algorithm can detect e.g. the Prototype pattern as seen in Figure 7 and the Adapter pattern.

Definition 2.4. A detection algorithm is *generally complete*, if it can detect all design patterns.

This type of algorithm offers more than that offered by *static decidability*. For example, it detects both static patterns (e.g. the Adapter and the Prototype pattern)and *non-static* patterns (e.g. the Singleton pattern).

The relations between these definitions is depicted in Figure 2. The purposes of the classical subdivision and our subdivision differ. The classical subdivision, creational, behavioral, and structural patterns is directed to *using* design patterns. Our subdivision is directed to *detecting* design patterns. The second subdivision is similar to our subdivision, but they also differ. Both subdivisions use static characteristics. However, the second subdivision is based on source code, which has to be executed to detect dynamic behavior. Our subdivision is based on design characteristics.

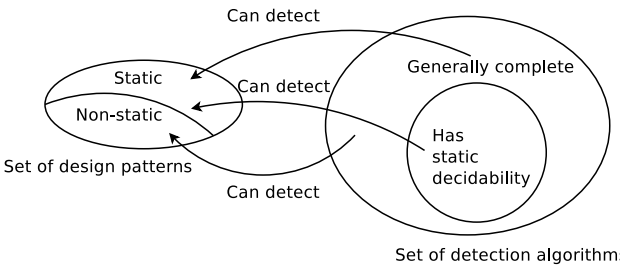

Figure 2: Relation between definitions

The *static* patterns can be shown in a UML structure diagram (e.g. a class diagram). In contrast to a class diagram, names of attributes and signatures of operations are not involved in the definition of *static decidability*. As stated in section 1, an interface can be regarded as a class and class may be abstract. When deciding whether a design Pattern is *static* or not, the differences between class, abstract class, and interface are irrelevant. Relationships are (directed) association, aggregation, composite, inheritance, realization and dependency. Potential multiplicities of relationships are irrelevant because they are not used by Gamma et al. [9].

Table 1 denotes the subdivisions: the *static* and *non-static* Gang of Four design patterns. Design patterns to which an *asterisk* is added, such as Adapter, contain attributes and/or operations in their definition given by Gamma et al. [9]. Their static structure is unique and their given attributes and/or operations have minor influence on the intention of the design pattern. So they are considered to be *static*. However, false positive detection of a design pattern remains possible. For instance, the Factory Method pattern has an unique static structure. So, a method that offers *static decidabilty* will detect an occurrence of a Factory Method pattern. But, an occurrence of a Factory Method needs to create an object. Without creating an object, a false negative will be generated.
There are 23 GoF design patterns, of which 16 are *static* and so an algorithm that offers *static decidability* is sufficient to detect them. For the remaining 23 - 16 = 7 design patterns, we give arguments as to why algorithms that offer *static decidability* are not capable to detect them.

- Façade: Any pattern where one class has connections with at least two other classes, would be a façade pattern. So, a false positive detection would be likely. More information is needed to decide whether this pattern is a façade pattern.
- Prototype: The operation `clone` is essential for detection. So, class/interface-names and their relations do not contain sufficient information to detect this pattern.
- Singleton: For detection, a static operation returning the value of a static attribute, is necessary. So, class/interface-names and their relations do not contain sufficient information to detect this pattern.
- State pattern: This pattern is structurally identical to the Strategy pattern. However, the behavior of the software depends on the state of the context and may change often during runtime. Whereas, the Strategy pattern is used for switching between different implementations of an algorithm. Switching happens less frequently.
  So, more than structural information is needed to distinguish these patterns.
- Template Method: The Template Method can only be detected by taking operations into consideration.
- Visitor: The number of operations of the interface visitor should be equal to the number of classes that implements the interface element.

Section 4.3 describes the multiple detections of one occurrence of a design pattern. Every occurrence of a design pattern should be detected only once. Detection algorithms with this property are named by the next definition.

Definition 2.5. A detection algorithm is *non-duplicating*, if it detects every occurrence of a design pattern only once.

In Section 4 we describe our algorithm which offers *static decidability* and is *non-duplicating*. This algorithm is applied in our prototype.

Static	non-static
Creational Patterns	
Abstract Factory	Prototype
Builder∗	Singleton
Factory Method∗	
Structural Patterns	
Adapter*	Façade
Bridge*	
Composite*	
Decorator*	
Flyweight*	
Proxy*	
Behavior Patterns	
Chain of reponsibility*	State/Strategy
Command*	Template Method
Interpreter	Visitor
Iterator*	
Mediator	
Memento*	
Observer*	

Table 1: Subdivisions of Gang of Four design patterns

3 INTRODUCTION TO DETECTION APPROACHES

This subsection gives an overview of detection methods based on the representations of UML class diagrams. It is indicated whether they are generally complete or offer *static decidability* or not. Whether or not they are non-duplicating is also denoted. Approaches are explained by an example consisting of the Prototype design pattern, which is searched for in a UML class diagram of an example system. For example, Figure 3 displays the essential classes and relationships of the Prototype design pattern while Figure 4 displays the sample system, that contains the said design pattern.

3.1 Matrices

One technique is to represent design patterns as a matrix [7, 19]. For every type of relationship in a class diagram (e.g. association, inheritance, and dependency) a matrix denotes the relationship. If a relationship (for instance, an association) exists between two classes, the corresponding matrix element is one; if not, it is zero.

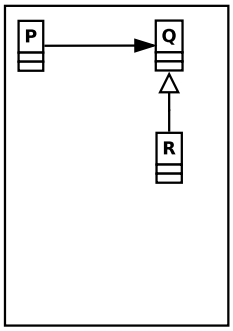

Figure 3: The Prototype pattern

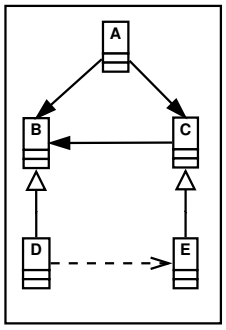

Figure 4: Example of a system

As an example, Table 2 shows for figure 4 the corresponding matrices for the associations, inheritances, and dependencies. Class A is associated with class B, is represented by the number 1 in row A and column B in the association matrix. Likewise, class D inherits from class B, is represented by the number 1 in row D and column B in the inheritance matrix, and class D depends on class E, is represented by the number 1 in row D and column E in the dependency matrix.

Association matrix	Inheritance matrix	Dependency matrix
$\begin{array}{c} \\ A \\ B \\ C \\ D \\ E \end{array} \begin{array}{c} \begin{array}{ccccc} A & B & C & D & E \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$	$\begin{array}{c} \\ A \\ B \\ C \\ D \\ E \end{array} \begin{array}{c} \begin{array}{ccccc} A & B & C & D & E \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix} \end{array}$	$\begin{array}{c} \\ A \\ B \\ C \\ D \\ E \end{array} \begin{array}{c} \begin{array}{ccccc} A & B & C & D & E \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$

Table 2: matrices representing Figure 4

Similarly, abstract classes and interfaces can be represented by matrices. If a class is abstract, its corresponding diagonal element of the abstract matrix is set to one. These matrices can be combined, resulting in one overall matrix [7]. The overall matrix is constructed as follows. The association matrix is as associated with the number 2, the inheritance matrix is associated with 3, and the dependency matrix is associated with 5. The value of the elements of the overall matrix in Table 4 is defined by

$$overallMatrix_{i,j} = 2^{associationmatrix_{i,j}} * 3^{inheritancematrix_{i,j}} * 5^{dependencymatrix_{i,j}}$$

In Table 4 the value 2 in row A and column B is calculated by: $overallMatrix_{1,2} = 2^{associationmatrix_{1,2}} * 3^{inheritancematrix_{1,2}} * 5^{dependencymatrix_{1,2}} = 2^1 * 3^0 * 5^0 = 2$
The association matrix shows in row A and column B the value 1.

An overall matrix is likewise constructed for a design pattern. For Figure 3 the overall matrix is Table 3.

Overall matrix

$\begin{array}{c} \\ P \\ Q \\ R \end{array} \begin{array}{c} \begin{array}{ccc} P & Q & R \end{array} \\ \begin{pmatrix} 1 & 2 & 1 \\ 1 & 1 & 1 \\ 1 & 3 & 1 \end{pmatrix} \end{array}$

Table 3: Overall matrix for Figure 3

Overall matrix

$\begin{array}{c} \\ A \\ B \\ C \\ D \\ E \end{array} \begin{array}{c} \begin{array}{ccccc} A & B & C & D & E \end{array} \\ \begin{pmatrix} 1 & 2 & 2 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 1 & 1 & 1 \\ 1 & 3 & 1 & 1 & 5 \\ 1 & 1 & 3 & 1 & 1 \end{pmatrix} \end{array}$

Table 4: Overall matrix for Table 2 and so Figure 4

To determine whether a design pattern is present in a class diagram, one has to compare the overall matrix of the design pattern to the overall matrix of the class diagram. For our example, compare Table 3 with Table 4. This can be done by cross-validation [7].

Alternatively, this can be done by using Blondel's or Zager's algorithm [6, 19, 21]. This will result in the solution as shown in Table 5. It is complicated to find the solutions by hand. In Table 3 (R, Q) has the value 3. This value has to correspond with one of the values 3 in Table 4. So, (R, Q) corresponds to (D, B) or (E, C). For simplicity, we only elaborate the correspondence (R, Q)⟶ (E, C). Now the correspondence of P has to be found. (P, Q) has the value 2 in Table 3. Q corresponds to C. In Table 4 column C has one value 2 (see row A). So, (P, Q) corresponds to (A, C). Therefore, P corresponds to A. Now, the solution in the third column of Table 5 is found. Likewise, the other two solutions can be found.

	Solution		
	1	2	3
P →	A	C	A
Q →	B	B	C
R →	D	D	E

Table 5: Solutions of comparing Table 3 and 4

By representing design patterns by a matrix, one can detect at least 10 GoF design patterns. When one searches for one of the patterns that can be detected, there are almost no false negatives. These exceptions involve permutations and interpretations of the Factory Method and the State pattern. So, this approach is focused on *static* design patterns and offers *static decidability*. The approach is not non-duplicating because permutations of one occurrence are not interpreted as one.
The authors did not give any details about the time to detect design patterns.

3.2 Decision trees

This subsection describes the use of decision trees as an approach to find design patterns. This approach also uses matrix representations of design patterns. Instead of comparing overall matrices as in section 3.1, a direct search that compares individual matrices one by one is used. This approach results in a decision tree [18].
The approach starts with constructing 7 matrices and one vector. The matrices and vector are used to represent 20 design patterns. The matrices are used for representing associations, aggregations, generalizations, instantiations of objects, method parameter references, similar method invocations, and abstract method invocations. The vector is used to indicate whether a class is abstract or is an interface. The design of the system under consideration is also described by 7 matrices and one vector.
Some patterns contain another pattern. For instance, the Abstract Factory contains the Factory Method. The search for these patterns can therefore be combined. If a Factory Method is detected then the search can be continued to detect an Abstract Factory.
The search for Interpreter, Proxy, Composite and Decorator patterns can be combined [18], because their structures resemble partly. These examples demonstrate that the detection of a design pattern can be based on decisions about continuing a search or differences between patterns. These decisions form a decision tree.
The authors claim that their approach can detect 20 out of 23 GoF design patterns. This approach is thus not *generally complete*, because 3 GoF design patterns can not be detected. It is unknown whether this approach is non-duplicating, because the authors did not pay attention to the possibility of multiple detections of one occurrence of a design pattern.

3.3 Prolog clauses

UML class diagrams may also be represented by Prolog clauses and rules. In that case, classes and relationships are represented by clauses, while design patterns are represented by rules.

As an example, the Prototype pattern (see Figure 3) can be represented by rules, as follows.

```
prototype(P, Q, R):-
class(concrete, P),
class(concrete, Q),
class(concrete, R),
association(P, Q),
inheritance(Q, R).
```

Here, P, Q, and R are the names of classes or interfaces, and the rules specify whether a class should be concrete (or abstract or an interface), and which relationship should exist between which classes.

By this approach, Prechet et al. could detect the patterns Adapter, Bridge, Component,Decorator, and Proxy. The estimation is that there are no false negatives, but there are false positives [14]. This approach has also been used to detect the following patterns: Factory Method, Prototype, Abstract Factory, Composite, Decorator, Adapter, Bridge, Proxy, Observer and Iterator [4].

An advantage of this approach is that it is possible to offer feedback on the detected patterns, involving the names of classes, attributes, operations, the scope of operations, missing operations, operations that may prevent reusability, and suggestions for the implementation of the pattern. The feedback can be shown in ArgoUML [4]. The authors did not give any information about the speed, and the number of false negatives and false positives. It is not clear, whether this approach offers *static decidability*. The description of the representation of a class and relationships suggests a *static decidability* approach, but the authors claim the detection of the Prototype pattern, which indicates an approach that offers more. It is unknown whether this approach is non-duplicating or not, because the authors did not pay attention to the possibility of multiple detections of one occurrence of a design pattern.
A test has shown, that 70 design patterns were detected in 2 seconds on a PC with a Pentium P133 under Windows 95. The precision was ± 14%.

3.4 Four Tuples

Another approach representing class diagrams is to use 4-tuples [1]. A 4-tuple (A, B, T, S) represents the relationship between classes or interfaces A and B. T stands for the type of relationship. T is an integer and may stand for direct association, dependency or generalization. S is a boolean, to indicate the existence of a self-loop (a class which has an association with itself). Our research shows that for none of the 16 static GoF design patterns require the use of the boolean self-loop.

However, our research shows that for none of the 16*static* GoF design patterns is it necessary to use the boolean self-loop element of a 4-tuple to define or detect a *static* design pattern.

Table 6 shows the possibilities for T.

type of relations	
1	direct association
2	dependency
3	generalization

Table 6: Representation of types of relations

This representation has not been implemented, but a straight forward depth-first search algorithm is expected to match the four tuples of a design pattern with a subset of four tuples representing the system under consideration. This search will detect all design patterns in a UML class diagram, that are fully defined by their class names and their relationships. So, this approach offers *static decidability* and it is unknown whether this approach is *non-duplicating*.

Summarizing, we found five ways representing structural elements in a UML class diagram and approaches to detect design patterns: matrix-based combined with cross-validation, matrix-based with a decision tree, Prolog clauses, sum of products, and 4-tuples. Implementations exist for matrix-based combined with cross-validation and Prolog clauses, but is not certain whether they of *static decidability*. For the other three approaches, the four tuple approach has several advantages (see section 1): it will not result in false negatives or positives because an exact search is used, design patterns can easily be defined, a detection algorithm seems to be implementable. It is not clear whether one of the three approaches can detect permutations and multiple occurrences of design patterns.

4 OUR APPROACH

Our approach for detecting *static* design patterns is based on using 3-tuples [8]. To make this approach useful in an educational and professional environment, several features are implemented. We show the representation of templates of design patterns and class diagrams. Templates of design patterns are described by XML. The drawing tool ArgoUML transforms class diagrams to XMI. The XML- and XMI-files are used as input of our prototype tool (see Figure 7). The problem of multiple detections of one occurrence of a design pattern is explained. We also explain the solution to this problem. Detection of illegal relationships within a detected design pattern are described. Finally, we pay attention to detecting design patterns that are partially present in a class diagram

4.1 3-tuples

A *static* design pattern is completely defined by the names of their participating classes and their relationships. Representing a design pattern we need the two names of the classes/interfaces, which are associated and the type of relationship. Possible types of relationship are direct association, inheritance, aggregate, and dependency.

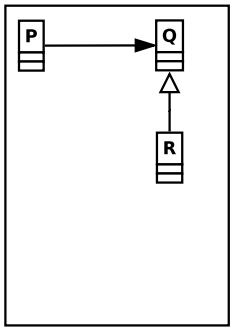

Figure 5: The Prototype pattern

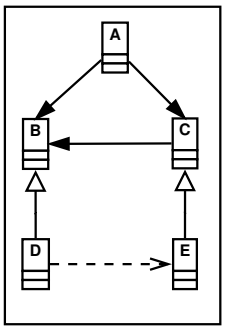

Figure 6: Example of a system

As an example, we repeat Figures 3 and 4 in Figures 5 and 6. We show the 3-tuples for Figure 5 and Figure 6 in Table 7. The table shows two sets of 3-tuples: SYS and DP, representing the system under consideration and the prototype pattern, respectively. The

SYS =	{ (A, B, 1), (C, B, 1), (A, C, 1), (D, B, 3), (E, C, 3), (D, E, 2) }
DP =	{ (P, Q, 1), (R, Q, 3) }

Table 7: 3-tuples for Figures 3 and 4

match between the 3-tuples of SYS and DP is shown in Table 8. The first row shows the two 3-tuples of DP. Each of the subsequent rows shows occurrences of corresponding 3-tuples in SYS. An occurrence means that there is a combination of three classes or interfaces of SYS that can be mapped onto both 3-tuples of DP. We use a recursive depth-first search algorithm to detect the occurrences of DP in SYS. The resulting table is Table 8.

3-tuples DP	(P, **Q**, 1)	(R, **Q**, 3, 0)
SYS 1	(A, **B**, 1)	(D, **B**, 3)
SYS 2	(A, **C**, 1)	(E, **C**, 3)
SYS 3	(C, **B**, 1)	(D, **B**, 3)

Table 8: Corresponding relationships between DP and SYS

The first row shows the two 3-tuples of DP. Each of the subsequent rows show occurrences of corresponding 3-tuples in SYS. An occurrence means that there is a combination of three classes or interfaces of SYS that can be mapped onto both 3-tuples of DP. We show, by making the names of a class bold, which classes should be the same in the two 3-tuples in a row. For example, SYS 1 is one of the three occurrences of the Prototype pattern. The map of the classes of DP into SYS is: P → A, Q → B, and R → D. See also Figures 3 and 4.

A recursive search algorithm tries to match all the tuples of DP one by one. The recursive search starts with a randomly chosen tuple of DP. If a match of this tuple in SYS can be found then the second recursive call tries to match another tuple and so on.

In general, DP contains several tuples. For performance reasons, every time a tuple of DP is chosen, it should connect to already chosen tuples of DP. Therefore, the chosen tuples of DP always constitute a connected graph. The number of tried matched would be enormous when the chosen tuples would not form a connected graph. For example, in the first row of Table 7 the classes P and Q are matched with the classes A and B. In the next step only one of the tuples (C, B, 1), (A, C, 1), (D, B, 3) can be chosen, because the A or B matches.

Is this problem *decidable*? A problem is *decidable*, if an algorithm exists, which for every input halts in a finite number of steps. This algorithm will for every DP and SYS find a match or report it can not find a match in finite time.

Proof: Let N and K be the numbers of tuples in DP and SYS. If N > K then no match is possible and so the algorithm halts. Therefore we assume N <= K. In every recursive call, a tuple of DP is matched with a tuple in SYS. If a call does not result in a match, the algorithm will backtrack and tries to match another tuple of DP with a tuple in SYS. The number of recursive calls is finite, in example N and the number of tuples which can be matched in a call is one less than in the previous call, so the algorithm will end. Therefore, the problem is decidable.

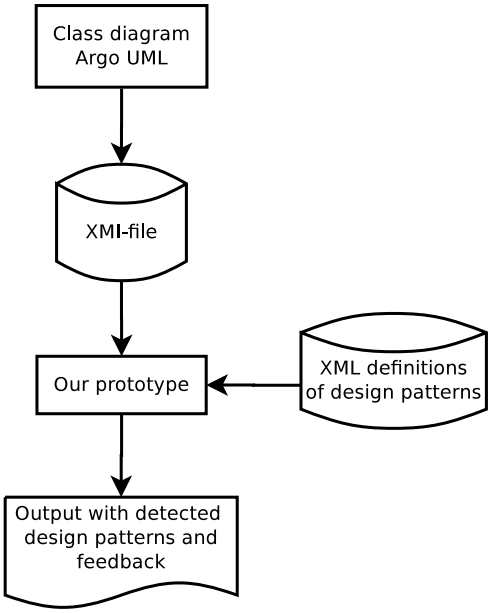

Figure 7: Prototype structure

4.2 The prototype tool with ArgoUML as input

The structure of our prototype tool is depicted in Figure 7.

We used ArgoUML[2] to create UML class diagrams and to generate an XMI-file. A generated XMI-file contains all information of a UML class diagram, which is needed by an algorithm that offers *static decidability*. We used XML templates to represent the definitions of design patterns. Our prototype tool is a command-line tool. The output is shown at the Command Prompt on Windows or in a xterm on Linux.

[2]http://argouml.tigris.org/

4.3 The problem of false multiple occurrences

A naive approach of our algorithm could detect a single occurrence of a design pattern several times. We will exemplify this using two different examples: the Abstract Factory and the Bridge pattern. We will also describe how these problems are solved in general and therefore we developed a non-duplicating algorithm that offers *static decidability*.

In Figure 8, we see one Abstract Factory.

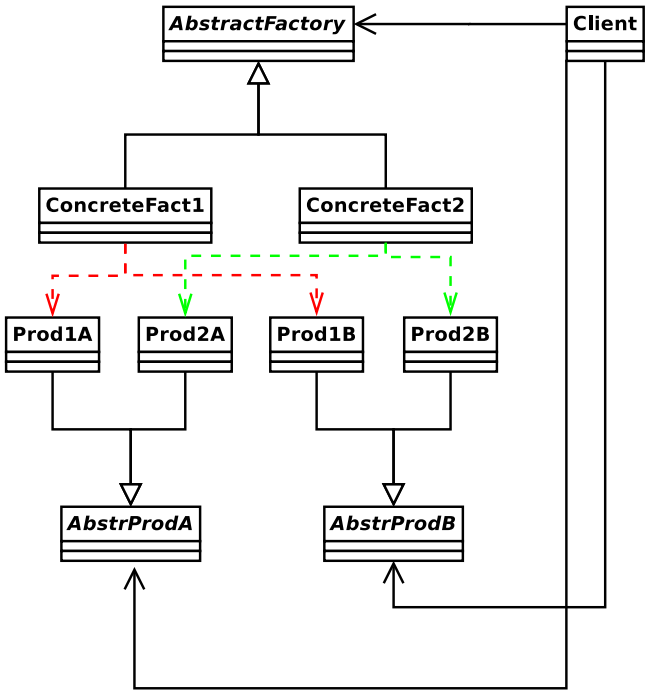

Figure 8: Abstract Factory pattern

Our first approach detected the Abstract Factory pattern four times in this system, because *Prod1A* and *Prod2A* can be interchanged, and *Prod1B* and *Prod2B* can be interchanged. Finding four instances of the pattern is obviously wrong because the four matches are permutations of one match.

Therefore, we improved the algorithm by adding a call to `isUniqueSolution`. These permutations of matches can easily be detected. Permutations have two relevant properties. They consist of the same classes, which is easy to check. Second, the classes are identically connected to the classes of the design pattern. Considering that candidate permutations are found in the same recursive search for a particular design pattern, these classes constitute the same design pattern.

A similar problem arises when one matches Figure 9 with the Bridge pattern (Figure 10).

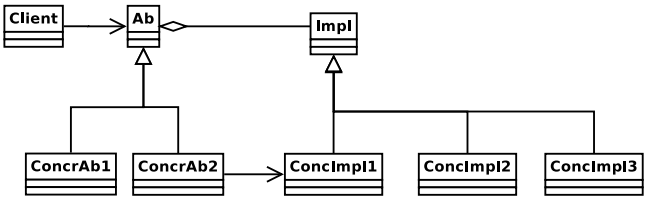

Figure 9: System with Bridge pattern

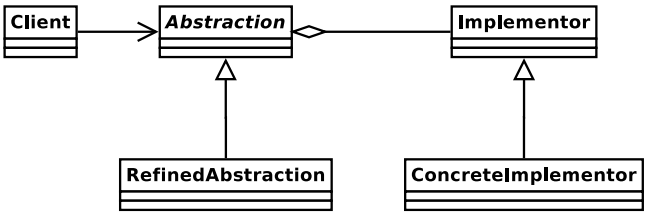

Figure 10: Bridge pattern

Our first approach detected the Bridge pattern 6 times in Figure 9 because the classes *ConcrAb1* and *ConcrAb2* can be interchanged and second, two of the classes *concImpl1*, *concImpl2*, *concImpl3* have to be chosen, which can be done in three ways.

Although the Abstract Factory and the Bridge patterns seem to occur multiple times, the reasons for the multiple occurrences differ. The Abstract Factory may have *two or more* abstract products and *two or more* concrete factories. The Bridge pattern has *one* abstract class *Ab*, which may have multiple realizations and *one* abstract class *Impl* which may have multiple realizations. It is hard to detect the complete Abstract Factory in the system under consideration, if there are more than two products or concrete factories. The problem of detecting a Bridge pattern in the system under consideration is solvable.

This problem is solved by defining a special inheritance association, which may occur in the design pattern. This association indicates that the inheritance association may occur multiple times in the system under consideration and the specializations do not have other associations in the design pattern. The specializations in the system under consideration may, however have associations. The classes in Figure 9 *Ab* and *Impl* have different numbers of specializations and the specializations *ConcrAb2* and *ConcImpl1* are associated. If the design pattern contains a normal inheritance association then numbers of specializations in the system under consideration and the design pattern should be equal, as shown in Figures 3 and 4.

4.4 Feedback on illegal relationships

Figure 9 shows another interesting phenomenon. The association between *ConcrAb2* and *ConcImpl1* is an association between classes, that are part of a match with the Bridge pattern. However, the association does not belong to the Bridge pattern and therefore it maybe a designer's mistake. This means that here, feedback is wanted.

We implemented a method `showSolution` that generates this type of feedback. During the search of a design pattern, all matched classes and associations are marked. `showSolution` checks whether not marked associations between marked classes exist. If so, then this association is redundant.

4.5 Detecting partial matches

The detection of partial existence of a design pattern is useful in an educational and professional environment. It helps to check whether the answer to an exercise is partly realized with the obligatory pattern, or in professional practice to check whether a design pattern is fully modeled. The search algorithm tries to match all 3-tuples. If no match is found, the algorithm tries to match all but one 3-tuples of DP. The number of unmatchable 3-tuples is a parameter of the search algorithm. However, finding a partially matched design pattern is not always useful (see Figure 11).

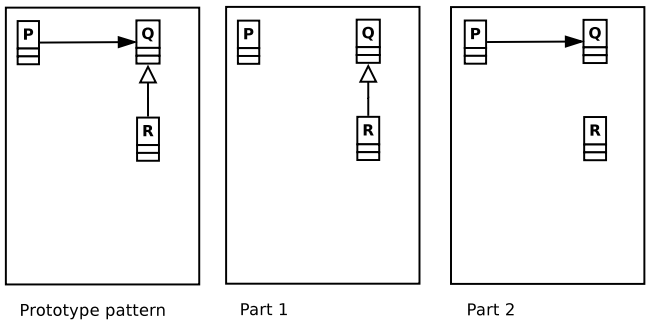

Figure 11: Prototype pattern and two parts

If one association may be missing, the partial design pattern will be detected many times because a single class combined with one inheritance (part 1 in Figure 11) or one association (part 2 in Figure 11) occurs frequently in class diagrams.

Lacking associations may result in false positives. For example, two Factory Methods which do not constitute as an Abstract Factory, may be detected an Abstract Factory, when several associations may be missing.

Detecting design patterns which lack at least one relationship is useful if the remaining parts of the design pattern can be represented by a connected graph. When in Figure 11 the association between P and Q is removed, the remaining parts do not constitute a connected graph and so false positives detections are likely.

5 VALIDATION IN PRACTICE

Here, we show that the implementation of our method works in practice. Our prototype tool can detect 16 of the 23 GoF patterns based on structural properties. We searched for a perfect match with templates of design patterns (based on literature [9]). No false positives and false negatives were reported.

Permutations of design patterns can be detected and reported once. This is shown in the example using the Abstract Factory and the Bridge pattern. There are two key differences between these patterns. The first difference is the number of repeating inheritance structures. The number can be variable, as for the Abstract Factory

or constant, as for the Bridge pattern. The second difference is the number of relationships that specializations may have. The specialization in an Abstract Factory has relationships, whereby the specializations in a Bridge pattern do not have relationships.
Redundant associations in a design pattern are reported as feedback.
The tests also showed that the speed of the prototype tool is good. A class diagram represented by an XMI-file with 33 classes, 49 relationships and 16 partially overlapping design patterns was processed within 0.8 seconds. These tests were executed on an AMD Athlon(tm) 64 X2 Dual Core Processor 5000+ [8].
All the necessary software is publicly available[3].

6 RELATED WORK

In an educational environment, much time is spent on how to apply design patterns since this increases the maintainability of the software. Experiences with teaching design patterns at university level are described by S. Stuurman [16].
The research on detection of design patterns was at first focused on analyzing source code. A modern example is APRT, Another Pattern Recognition Tool. This tool can detect design patterns by parsing Java Sources. The parser is generated by ANTLR (https://www.antlr.org/). The specified design patterns are language independent. Therefore, by changing the specification of the source language (e.g. Java) other object-oriented languages can be used as well [3].
Beside by parse trees source code may also be represented by graphs. Structural and behavioral characteristics of design patterns may be represented by graphs. Bahareh Bafandeh Mayvan et al. showed by analysis of JHotDraw5.1, JRefactory 2.6.24 and JUnit 3.7 that ten design patterns could be detected with 100% precision and 100% recall. In some cases, the precision or recall was not available, because the division was by zero. Their approach can not to distinguish Strategy and State patterns. [2].
De Lucia et al. analyze class diagrams, which are generated from sources. The behavior of candidate design patterns is analyzed based on generated sequence diagrams and runtime analyzes of bytecodes within jar files based on generated testdata. They focused on creational and behavioral design patterns [11].
Prechelt, as described in section 3.3 was able to detect 70 occurrences of 4 design patterns with a precision of ± 14% [14]
Matrix representation of class diagrams for detecting design patterns is described by the approach of Tsantalis [18]. He claims that his approach could detect 20 out of 23 GoF patterns, but there is no empirical evidence.

The approach of Oruc et al. is based on finding subgraphs, representing design patterns, in a graph by a standard algorithm. Their approach succeeded in detecting all 23 Gang of Four design patterns with a precision of 80% and recall 88% on average [12].
Pelzus et al. use a rule-based approach, together with a technique to detect multiple design flaws and detection of code smells to detect anti-patterns such as *The Blob, God class, and Swiss Army Knife* [13].

[3] http://members.chello.nl/e.doorn1/DesignPatterns/static_decidability

The quality of software is positively influenced by using design patterns and detecting and removing code smells. Walter et al. discovered negative relations between some design patterns and some code smells [20].

7 CONCLUSION

We have shown that our prototype tool can automatically detect *static* design patterns and can give simple feedback. This prototype tool can be used as a pilot for a first course about design patterns. We have introduced a new subdivision of the set of Gang of Four design patterns: *static* and *non-static* and a new classification of detection algorithms: *static decidability* and *generally complete*. We have made several contributions to the progress of research to automatically detecting design patterns in UML class diagrams. Addressing the problem of multiple detections of one occurrence of a design pattern is new. We introduced the concept of *non-duplicating* for higher quality detection algorithms. With the presented prototype, 16 out of 23 Gang of Four Design patterns are detectable, within an acceptable time. 16 partially overlapping design patterns were detected within 0.8 seconds in a class diagram containing 33 classes and 49 relationships. Permutations and so multiple occurrences of these design patterns are also detectable. We therefore developed a non-duplicating algorithm that offers *static decidability*. The remaining six design patterns are not detectable by our approach, because they are not fully specified by structural properties.

Feedback can be given but is restrained to the notification of superfluous relationships. Detection of a design pattern, that lacks one or more relationships is useful, if the remaining graph of the design pattern is connected, otherwise false positives may result.
Our prototype tool shows that the detection method of design patterns, and the introduction of a new subdivision of design patterns offers perspectives for educational application.

8 FUTURE WORK

Our next research will focus on the benefits of the use of the prototype tool in an educational environment in which *static* design patterns are introduced. Obviously, instructors and students would be helped if all design patterns would be detectable. So, a *generally complete* algorithm is an important goal for future work. This requires figuring out what information is necessary by such an algorithm and how the information should be structured and supplied to a tool.

If Figure 8 would contain three abstract products, the current prototype tool would detect Abstract Factory three times while it is actually one occurrence. There is currently no simple way disclosed to solve this problem.

Automatically giving feedback would, of course, be a large improvement. This will require a significant extension of the algorithm.

Can the algorithm be extended to analyze Sequence and State diagrams in order to detect non-static patterns?

Finally, the scope of our research could be extended to include other characteristics of class diagrams which may give information of the quality of design, both in education and in development.

REFERENCES

[1] Afnan Salem Ba-Brahem and M. Rizwan Jameel Qureshi. 2014. The proposal of improved inexact isomorphic graph algorithm to detect design patterns. *CoRR* abs/1408.6147 (2014). http://arxiv.org/pdf/1408.6147v1.pdf

[2] Bahareh Bafandeh Mayvan and Abbas Rasoolzadegan. 2017. Design Pattern Detection Based on the Graph Theory. *Know.-Based Syst.* 120, C (mar 2017), 211–225. https://doi.org/10.1016/j.knosys.2017.01.007

[3] Chris Bates and Ashley Robinson. 2017. APRT – Another Pattern Recognition Tool. *GSTF Journal on Computing (JoC)* 5, 2 (2017). http://dl6.globalstf.org/index.php/joc/article/view/1215

[4] F. Bergenti and A. Poggi. 2000. IDEA: A design assistant based on automatic design pattern detection. In *Proceedings of 12th International Conference on Software Engineering and Knowledge Engineering (SEKE 2000)*. 336–343. https://www.sics.se/~nilsf/SEKE.pdf

[5] Mario Luca Bernardi, Marta Cimitile, and Giuseppe Di Lucca. 2014. Design Pattern Detection Using a DSL-driven Graph Matching Approach. *J. Softw. Evol. Process* 26, 12 (dec 2014), 1233–1266. https://doi.org/10.1002/smr.1674

[6] Vincent D. Blondel, Anahí Gajardo, Maureen Heymans, Pierre Senellart, and Paul Van Dooren. 2004. A Measure of Similarity Between Graph Vertices: Applications to Synonym Extraction and Web Searching. *SIAM Rev.* 46, 4 (apr 2004), 647–666. https://doi.org/10.1137/S0036144502415960

[7] Jing Dong, Yongtao Sun, and Yajing Zhao. 2008. Design Pattern Detection by Template Matching. In *Proceedings of the 2008 ACM Symposium on Applied Computing (SAC '08)*. ACM, New York, NY, USA, 765–769. https://doi.org/10.1145/1363686.1363864

[8] Ed van Doorn. 2016. *Supporting design process by automatically detecting design patterns and giving some feedback*. Master's thesis. Open University, Heerlen, The Netherlands. http://dspace.ou.nl/bitstream/1820/7198/1/INF_20160824_Doorn.pdf

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[10] Hakjin Lee, Hyunsang Youn, and Eunseok Lee. 2008. A design pattern detection technique that Aids reverse engineering. *International Journal of Security and its Applications* 2 (01 2008).

[11] Andrea De Lucia, Vincenzo Deufemia, Carmine Gravino, and Michele Risi. 2018. Detecting the Behavior of Design Patterns Through Model Checking and Dynamic Analysis. *ACM Trans. Softw. Eng. Methodol.* 26, 4, Article 13 (feb 2018), 41 pages. https://doi.org/10.1145/3176643

[12] M. Oruc, F. Akal, and H. Sever. 2016. Detecting Design Patterns in Object-Oriented Design Models by Using a Graph Mining Approach. In *2016 4th International Conference in Software Engineering Research and Innovation (CONISOFT)*, Vol. 00. 115–121. https://doi.org/10.1109/CONISOFT.2016.26

[13] Sven Peldszus, Géza Kulcsár, Malte Lochau, and Sandro Schulze. 2016. Continuous Detection of Design Flaws in Evolving Object-oriented Programs Using Incremental Multi-pattern Matching. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. ACM, New York, NY, USA, 578–589. https://doi.org/10.1145/2970276.2970338

[14] Lutz Prechelt and Christian Kramer. 1998. Functionality versus Practicality: Employing Existing Tools for Recovering Structural Design Patterns. *j-jucs* 4, 12 (dec 1998), 866–882. http://www.jucs.org/jucs_4_12/func-tionality_versus_practicality_employing

[15] N. Shi and R. A. Olsson. 2006. Reverse Engineering of Design Patterns from Java Source Code. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*. 123–134. https://doi.org/10.1109/ASE.2006.57

[16] Sylvia Stuurman. 2015. *Design for Change*. Ph.D. Dissertation. Open University, Department of Computer Science. http://cs.ou.nl/phd-theses/sylvia-stuurman-phd-thesis.pdf

[17] Sylvia Stuurman and Gert Florijn. 2004. Experiences with Teaching Design Patterns. *SIGCSE Bull.* 36, 3 (jun 2004), 151–155. https://doi.org/10.1145/1026487.1008037

[18] Nikolaos Tsantalis, Alexander Chatzigeorgiou, Spyros T. Halkidis, and George Stephanides. 2005. A Novel Approach to Automated Design Pattern Detection. In *10th Panhellenic Conference on Informatics, PCI 2005, Volos, Greece - November 11 - 13, 2005 (Lecture Notes in Computer Science)*, Panayiotis Bozanis and Elias N. Houstis (Eds.), Vol. 3746. Springer.

[19] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S.T. Halkidis. 2006. Design Pattern Detection Using Similarity Scoring. *Software Engineering, IEEE Transactions on* 32, 11 (Nov 2006), 896–909. https://doi.org/10.1109/TSE.2006.112

[20] Bartosz Walter and Tarek Alkhaeir. 2016. The Relationship Between Design Patterns and Code Smells. *Inf. Softw. Technol.* 74, C (jun 2016), 127–142. https://doi.org/10.1016/j.infsof.2016.02.003

[21] Laura A. Zager and George C. Verghese. 2008. Graph similarity scoring and matching. *Applied Mathematics Letters* 21, 1 (2008), 86 – 94. https://doi.org/10.1016/j.aml.2007.01.006