A Novel Approach to Automated Design Pattern Detection

Nikolaos Tsantalis, Alexander Chatzigeorgiou, Spyros T. Halkidis and George Stephanides

Department of Applied Informatics, University of Macedonia,
Egnatia 156, GR-54006 Thessaloniki, Greece
nikos@java.uom.gr, {halkidis, achat, steph}@uom.gr

Abstract. The importance of the use of Design Patterns in order to build reusable and well-structured software has been eminent since these patterns have been formalized. Thus, it became desirable to be able to detect which design patterns are present in a software system. Knowing this information it is possible to make an evaluation on different aspects of the system. Though, it is a very difficult task for a software engineer to pinpoint all the Design Patterns present in a system, without any assistance. Addressing this need, techniques for automated design pattern detection have appeared in the literature. Some are based on reverse-engineering of already existing code while others can work already at design level by analyzing UML diagrams. Though, complexity is one of the characteristics of all the methods proposed until now. Furthermore, all of these techniques work only for a limited number of the GoF patterns. Our aim in this paper is to elaborate on a simple approach for automatic detection of design patterns that works by analyzing UML class diagrams. Our method can achieve the automated detection of all GoF patterns that do not require any code specific information to recognize them.
Research Track: Software Engineering

1 Introduction

Since the so-called GoF patterns [7] have been proposed they have been widely used by the software engineering community. This happened due to the fact that it has been practically proven that their use leads to the construction of efficient, well-structured and reusable software systems.

Based on these facts, it is easy to conclude that it is very valuable to be able to identify which of the GoF patterns are present in a software system. The identification of implemented design patterns as part of the reengineering process can assist the software architect in understanding the underlying abstractions, and in the application of pattern-specific rules for the improvement of the design. However, it is easy to estimate that it is a very difficult task for a human engineer to find all the design patterns present in a large software system, whether he is examining the code of an already existing system or examining the UML diagrams [2] of a system to be built.

Noting this problem, various approaches to automatically detect design patterns requiring either software code or UML diagrams as input have appeared in the literature. Nevertheless, most of them seem to be complex and furthermore they are usually applied to only a small subset of the GoF patterns. Specifically, most of the

modern techniques use AI-based approaches to aid in the detection of the patterns and for none of these methods it is clear how they could work for more than half of the GoF patterns. Moreover it is questionable how such an approach would scale to real life projects with hundreds or thousands of classes.

Our aim in this paper is to present a novel approach to automated design pattern detection that is governed by simplicity and covers all the GoF patterns whose detection is not code level dependent. This means that our approach works for 20 of the 24 GoF patterns. Our approach is not to cover code dependent patterns since it recently has been clear to the software engineering community that it is desirable to achieve good software design quality before any line of code is written. So, we would like to analyze a system in terms of the presence of design patterns having as input UML class diagrams.

The outline of our approach is as follows: We first build for each of the 20 GoF patterns we examine, a set of appropriate matrices that summarize the information that is vital to the detection of the patterns. This is the representation of the design patterns. Then, we regard as input to the system the UML class diagram of the system. If we have the code of the software system itself, it is obvious that we can use a reverse engineering tool to get the corresponding class diagram. However, since we are interested at the identification of patterns already at the design level, we assume that any class diagram will be transformed to an intermediate XML representation. By parsing this XML representation we construct the same arrays and vector, we document the system structure using the same set of matrices. This is the system representation. The task of automated design pattern detection is then accomplished by a tree-visualized guided search of the design patterns representation inside the system representation.

The remainder of the paper is organized as follows. Section 2 describes previous work on automated design pattern detection. Section 3 describes our approach in detail. Finally, in Section 4 we draw some conclusions and propose future work.

2 Previous Work

A notion related to design patterns, before these appeared in the literature was the one of *clichés*. In terminology of Rich and Waters, the heads of the Programmer's Apprentice project [11], clichés were "commonly used combinations of elements with familiar names". The Programmer's Apprentice project aimed at developing an intelligent assistant for program developers that worked from knowledge base information about programming stored in the form of clichés. A segment of this project called the Recognizer analyzed source code in various languages and derived a representation of the source programs in a form that could be compared to the clichés stored in the knowledge base. We can consider the Recognizer part of the Programmer's Apprentice as an ascendant of today's automated design pattern detection techniques.

The first attempt to automatically detect design patterns was performed by Brown [3]. In this work Smalltalk code was reverse engineered in order to detect the

Composite, Decorator, Template Method and Chain of Responsibility patterns. The categories of information that the related algorithm was based were class hierarchy representation, aggregation information, association information and information about the messages exchanged between classes of the system.

Prechelt and Krämer [10] used some tools to develop a system that could identify some design patterns present in C++ source code. The design patterns examined were represented as static OMT class diagrams. These class diagrams formed the basis for building some Prolog rules appropriate to aid in recognizing these patterns. Then using structural analysis of the C++ code and combining the result of this analysis with the aforementioned rules the Adapter, Bridge, Composite, Decorator and Proxy patterns could be detected. The Prolog rules that were built were inheritance, aggregation, association and operations between classes related.

As it is noted by Wendehals [14], in order to efficiently detect the design patterns present in a software system a smart combination of static and dynamic analysis is desirable. Most of the recently developed techniques follow this principle. In terms of UML this translates into analyzing the class diagram in order to recover the static information and analyzing the sequence or collaboration diagram for the dynamic information.

Heuzeroth et. al. [8] describe a technique for detecting the Observer, Composite, Mediator, Chain of Responsibility and Visitor patterns by combining static and dynamic analysis. Their method is applied to Java source code and it is not very clear how their approach could be extended in order to be used at design level. Though, the algorithms they present for the static analysis of a software system are governed by simplicity, because no AI techniques are required to implement their approach. Moreover, the dynamic analysis they perform is expressed in the form of simple rules. We note that, although in their paper no complexity analysis of the algorithms is present, it is easy to evaluate that the algorithm for the detection of the Visitor pattern has the highest complexity of $O(n^5)$, where n corresponds to elements examined, while an element can be a class, method or parameter.

The most comprehensive approach presented until now for automated design pattern detection seems to be the technique developed by Bergenti and Poggi [1]. In their approach the input to the automated design pattern detection system is the UML design of the software system to be examined in XMI format. The class and collaboration diagrams are used to detect all pattern realizations. Design pattern candidates are built by examining the class diagrams. These candidates are then evaluated based on information present in the collaboration diagrams. Finally, the patterns detected together with recommendations concerning possible improvements to the design are presented. These recommendations are based on simple design rules that are followed in the correct representation of each design pattern. The main part of the system related to the detection of the patterns is a knowledge base consisting of Prolog rules that describe the main characteristics of the patterns. The patterns detected by this technique include Proxy, Adapter, Bridge, Composite, Decorator, Factory Method, Abstract Factory, Iterator, Observer and Prototype.

A different approach to automated design pattern detection has been presented by Smith and Stotts [13]. Their approach is based on the notion of elemental design patterns. Elemental design patterns [12] are smaller parts than GoF patterns which are present in them. The approach introduced by Smith and Stotts proposes identifying

first these elemental design patterns and then composing these findings to recognize GoF patterns. In order to represent directly relationships and reliances between objects, methods and fields a formal language called rho calculus is used. This language is used to formalize the design patterns. The same language is used to represent the system under consideration. Then, an automated theorem prover is used to detect instances of patterns in the system. Though, it is not clear which heuristic is used to combine the existing predicates in order to achieve this result. Obviously the computationally complexity of examining all the possible combinations, i.e. when no heuristic is applied, is prohibitive. The applicability of this technique is presented with an illustration of the steps required to detect the Decorator pattern. The main power of an approach based on the notion of elemental design patterns is the ability to detect a design pattern after some refactorings [6] have been applied to it.

3 Description of our approach

Our approach can be summarized in two steps. In the first step we model information that is vital to the automated design pattern detection process as a set of eight matrices and one vector. As it will be made clear in the detailed description of this type of modeling, this kind of representation is intuitively appealing for engineers and computer scientists. We first formalize all of the 20 GoF patterns we examine using this set. We then build the corresponding set for the software system under consideration. This can be achieved by parsing the XML representation that can be acquired from the tools used to build the class diagram for the system. In the second step we perform a directed search that can be described by a tree of actions. As we propose though, the actual search will be done in the set of eight matrices and one vector representing the software system.

3.1 Representation of class diagrams as matrices

The representation of the information present in a class diagram, that is vital to the detection of a design pattern, as a set of matrices that show the relations between classes present in the diagram, seems to be very natural. This will be illustrated by showing examples for each kind of the matrices we use. These examples depict the associated representation for a specific GoF pattern. We note that since we do not use, in our approach, any diagrams that provide us with information related to dynamic analysis, we try to use the information present in notes of the class diagrams. These notes usually contain information regarding method invocations. Furthermore, it is important to realize that the Singleton, Flyweight and Template Method patterns require information that is code specific, while the application of the Façade pattern is of a very abstract nature. Thus it was not possible to incorporate these patterns in our approach since we examine pattern detection already at design level.

Associations between classes are shown in the **Association** matrix in which a reference from the row class to the column class is indicated by a "1" in the corresponding cell. As an example, we examine the corresponding matrix for the

Command pattern, which is shown in Table 1. For the sake of clarity we include the class diagram of the Command pattern in Figure 1.

Table 1. The Association matrix for the Command Pattern.

Association	Command	ConcreteCommand	Receiver	Invoker
Command	0	0	0	0
ConcreteCommand	0	0	1	0
Receiver	0	0	0	0
Invoker	0	0	0	0

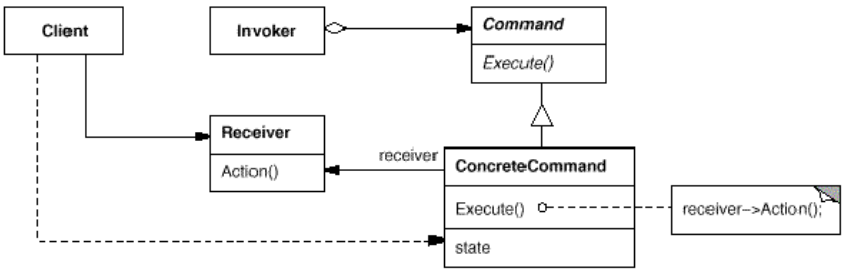

Fig. 1. The class diagram of the Command pattern (Adapted from [7]).

It is easy to see the simplicity by which the association between the ConcreteCommand class and the Receiver class is depicted.

For containment relationships we employ a separate **Aggregation** matrix. The Aggregation matrix for the above pattern is shown in Table 2.

Table 2. The Aggregation matrix for the Command pattern.

Aggregation	Command	ConcreteCommand	Receiver	Invoker
Command	0	0	0	0
ConcreteCommand	0	0	0	0
Receiver	0	0	0	0
Invoker	1	0	0	0

The Aggregation matrix has a "1" in the row for the Invoker class and the column of the Command class since the invoker may cause the invocation of an aggregation of Commands.

Inheritance is captured in a **Generalization** matrix and as an example we use the Abstract Factory pattern. The related matrix is shown in Table 3. We include also the class diagram of the Abstract Factory pattern in Figure 2.

Table 3. The Generalization matrix for the Abstract Factory pattern.

Generalization	AbstractFactory	ConcreteFactory	AbstractProduct	Product
AbstractFactory	0	0	0	0
ConcreteFactory	1	0	0	0
AbstractProduct	0	0	0	0
Product	0	0	1	0

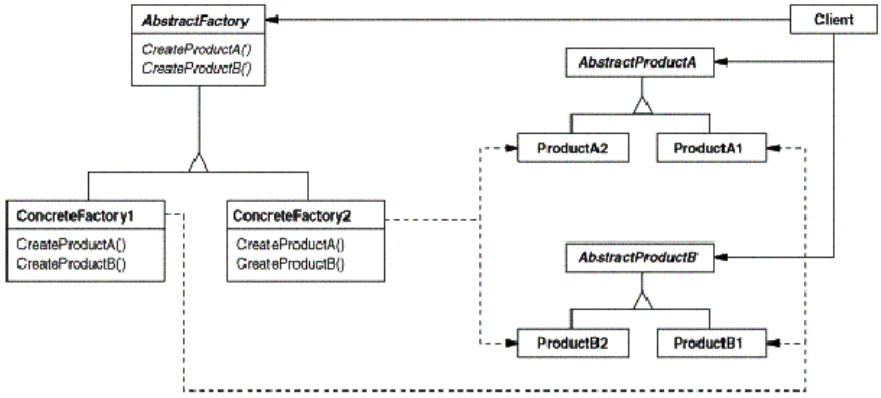

Fig. 2. The class diagram of the Abstract Factory pattern (Adapted from [7]).

By examining the matrix, we can see that there is a "1" in the row for ConcreteFactory and the column from AbstractFactory, since ConcreteFactory is a subclass of AbstractFactory and there is also a "1" in the row for Product and the column for AbstractProduct since Product is a subclass of AbstractProduct.

To capture instantiation of classes by other classes, we build a **Creation** matrix. A cell marked with "1" indicates that the row class creates instances of the column class. The creation matrix for the Abstract Factory pattern is shown in Table 4.

Table 4. The Creation matrix for the Abstract Factory pattern.

Creation	AbstractFactory	ConcreteFactory	AbstractProduct	Product
AbstractFactory	0	0	0	0
ConcreteFactory	0	0	0	1
AbstractProduct	0	0	0	0
Product	0	0	0	0

We can see that there is a "1" in the row for ConcreteFactory and the column for Product since a ConcreteFactory can create a Product object.

To indicate which entities in the design are abstract classes or interfaces we use a simple **Abstract Class/Interface** vector. The corresponding vector for the same pattern is shown in Table 5.

Table 5. The Abstract Class/Interface Vector for the Abstract Factory pattern.

Abstract Class/Interface	AbstractFactory	ConcreteFactory	AbstractProduct	Product
	1	0	1	0

There is a "1" in the columns for AbstractFactory and AbstractProduct, since they are abstract classes.

The Association, Aggregation, Generalization and Creation matrices as well as the Abstract Class/Interface vector are required to detect all of the 20 patterns we examine. The rest of the matrices are needed only for the detection of some of the patterns.

The next matrix stores information to identify classes that inherit a method in which they invoke a method of another class. To examine the **Invoked Method in Inherited Method** matrix we use the Command pattern as example. The corresponding matrix is shown in Table 6. The class diagram of the Command pattern was already presented in Figure 1.

Table 6. The Invoked Method in Inherited Method matrix for the Command pattern. (Explanations, 1: Row class inherits method x from column class, 2: Method x invokes method of column class).

Invoked Method in Inherited Method	Command	ConcreteCommand	Receiver	Invoker
Command	0	0	0	0
ConcreteCommand	1	0	2	0
Receiver	0	0	0	0
Invoker	0	0	0	0

In the Invoked Method in Inherited Method matrix for the Command pattern, there is a "1" in the row for ConcreteCommand and the column for Command since the ConcreteCommand class invokes the execute() method which is inherited from Command. There is also a "2" in the row for ConcreteCommand and the column for Receiver since the execute() method in ConcreteCommand invokes the action() method of the Receiver class.

The **Abstract Method Invocation** matrix shows calls of abstract methods within abstract methods of other classes. We examine the Abstract Method Invocation matrix with the Bridge pattern as an example. The corresponding matrix is shown in Table 7. The class diagram of the Bridge pattern is included in Figure 3.

Table 7. The Abstract Method Invocations matrix for the Bridge pattern

Abstract Method Invocations	Abstraction	RefinedAbstraction	Implementor	ConcreteImplementor
Abstraction	0	0	1	0
RefinedAbstraction	0	0	0	0
Implementor	0	0	0	0
ConcreteImplementor	0	0	0	0

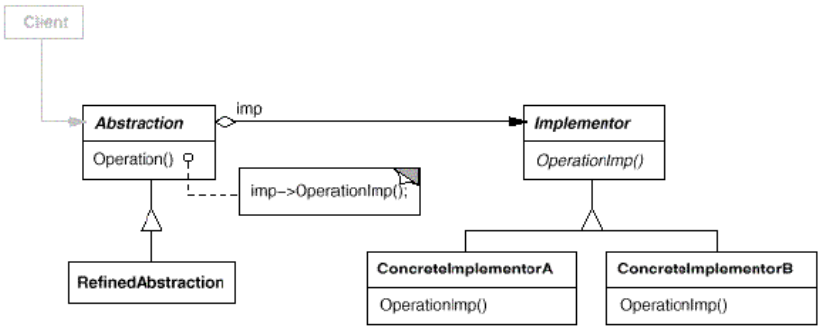

Fig. 3. The class diagram of the Bridge pattern (Adapted from [7]).

In the Abstract Method Invocations matrix for the Bridge pattern there is a "1" in the row for Abstraction and the column for Implementor, since the abstract method operation() of Abstraction calls the abstract method operationImp() of Implementor.

For some patterns it is important to identify whether methods in one class invoke similar methods in another class. Two methods are considered similar if they have equivalent signatures [13]. For this reason a **Similar Method Invocation** matrix is employed which captures calls of similar methods. As an example we show the corresponding matrix for the Decorator pattern in Table 8. The class diagram of the Decorator pattern is presented in Figure 4.

Table 8. The Similar Method Invocation matrix for the Decorator pattern. (Explanations, 1: Row class method calls similar method through reference 2: Row class method calls similar method with super invocation)

Similar Method Invocation	Component	ConcreteComponent	Decorator	ConcreteDecorator
Component	0	0	0	0
ConcreteComponent	0	0	0	0
Decorator	1	0	0	0
ConcreteDecorator	0	0	2	0

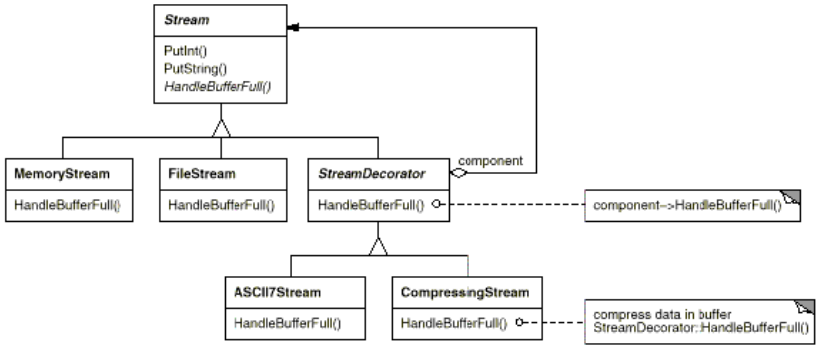

Fig. 4. The class diagram of the Decorator pattern (Adapted from [7]).

There is a "1" in the row for Decorator and the column for Component, since the Decorator calls the operation() method of its superclass Component through a reference. There is a 2 in the row for ConcreteDecorator and the column for Decorator, since the ConcreteDecorator calls the operation() method of its superclass Decorator with super invocation.

Finally, we are also interested in classes that have methods with an object of another class as parameter. For this reason we use a **Method Parameter Reference** matrix in which a "1" in a cell, indicates that one or more methods of the row class have an object of the column class as parameter. The corresponding matrix for the Interpreter pattern is shown in Table 9. The related class diagram is shown in Fig. 5.

Table 9. The Method Parameter Reference matrix for the Interpreter pattern.

Method Parameter Reference	Abstract Expression	Terminal Expression	NonTerminal Expression	Context
AbstractExpression	0	0	0	1
TerminalExpression	0	0	0	1
NonTerminal Expression	0	0	0	1
Context	0	0	0	0

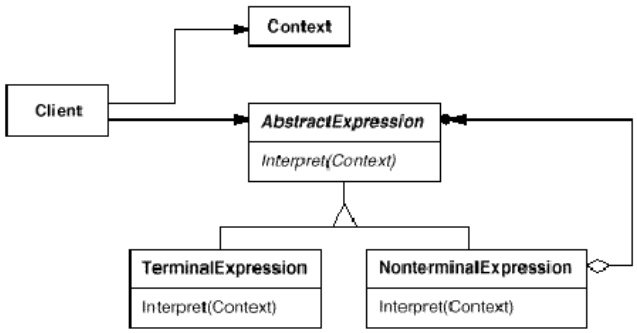

Fig. 5. The class diagram of the Interpreter pattern (Adapted from [7]).

Here, there is a "1" in the rows for AbstractExpression, TerminalExpression and NonTerminalExpression and the column for Context, since the classes in the rows have methods that have a Context object as parameter.

3.2 The Directed Search for the Design Patterns

Having represented all the patterns as a set of matrices and having followed this procedure also for the representation of the software system under consideration there is a method required in order to perform the actual search. One method that could someone think of to achieve this would be to use 2D pattern matching techniques for the matrices and simple pattern matching techniques for the vector [5]. The main idea of using this method, when examining the system for a specific pattern, would be to search for the small matrix of the pattern in the corresponding matrix of the system. The same would apply for the Abstract Class/Interface vector using though 1D pattern matching algorithms for it. We would report that a design pattern occurrence was found when for a specific position of the system representation all the matrices and the vector match. The disadvantage of this approach is though, that pattern matching algorithms require exact matching. We would thus need to have the same ordering in the classes of the system representation as with the classes in the pattern representation in order to find a match when it really exists. It is difficult though to impose some ordering to the classes present in a system.

Having examined this possible solution and its disadvantages we concluded that a directed search based on characteristics that are summarized in the matrices/vector representation could be employed. The criteria that are used in this search to detect the patterns are summarized in the figures that follow.

The directed search is organized as a tree in which the path from the root node to each pattern is traversed by observing the properties of classes and the associations between them. In particular:

- **nodes** represent the class under study
- **edges** represent the conditions that must be fulfilled in order for a transition to take place
- **semicircular arrows** represent iterations through all elements of some type
- **dashed rectangles** (mainly in leaf nodes) contain the description of class' roles in the pattern

The algorithmic complexity of detecting each pattern can be easily inferred from the diagram. Each semicircular arrow and each 'exists' symbol (∃) in a path should be regarded as a separate loop.

The first two figures we examine are the two parts of the same tree and have as common root an abstract class. The first part of the tree is traversed if the abstract class examined has at least one child, while the second part of the tree is traversed if the abstract class examined has at least two children.

In the first part of the tree all GoF patterns we consider *except* Interpreter, Composite, Proxy, Decorator and Memento can be detected. This part is shown in Figure 6.

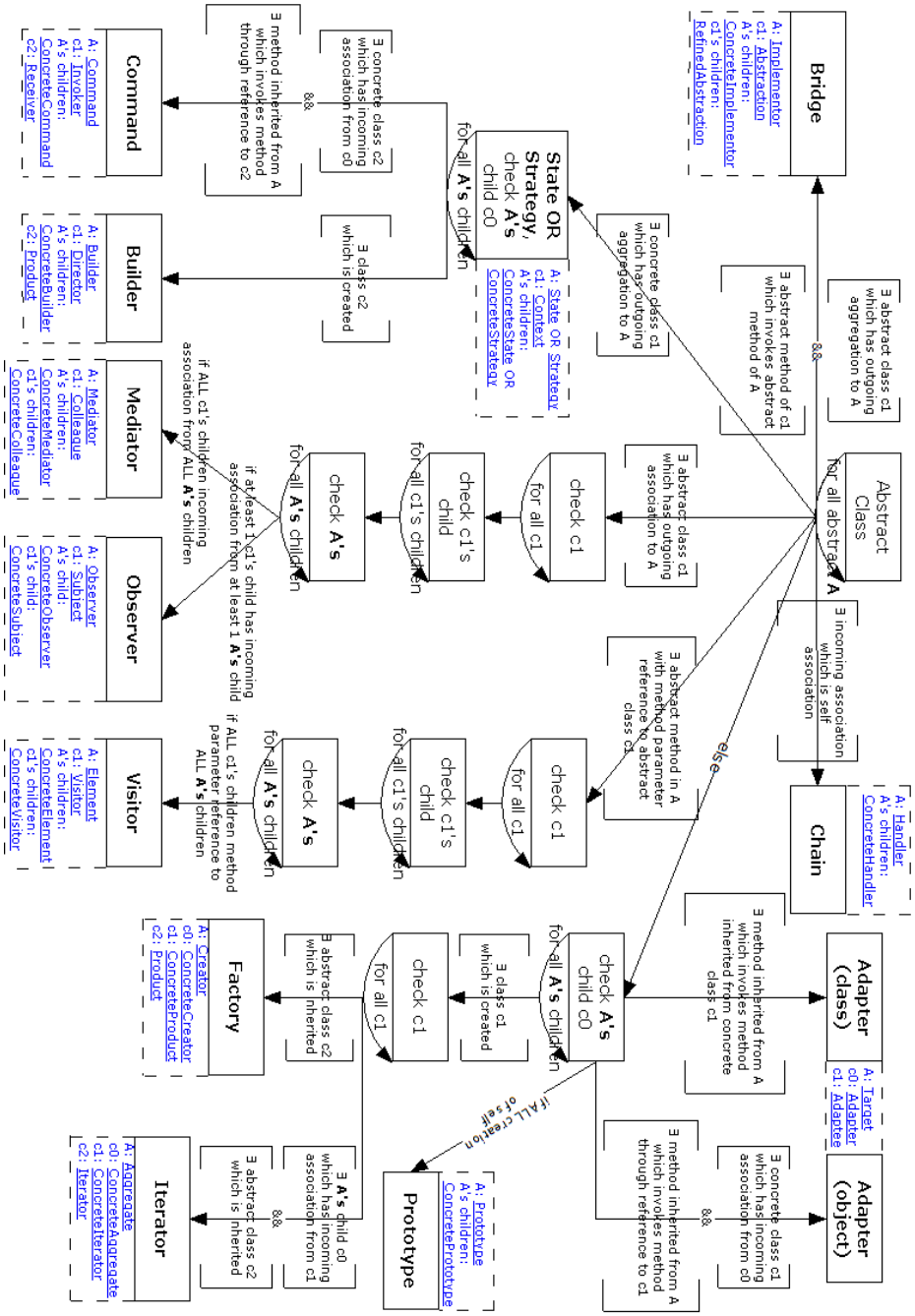

Fig. 6. The part of the search tree that can detect all GoF patterns we consider, except Interpreter, Proxy, Composite, Decorator and Memento.

The Command and Builder patterns are always State or Strategy, so we search for State or Strategy and if this succeeds we report it and continue to search for the Command and Builder patterns. The Abstract Factory pattern is composed of Factory Method patterns. So we can search for Factory Method in order to indirectly detect the Abstract Factory pattern.

In Figure 7 we can see the part of the tree that can detect the Interpreter, Proxy, Composite and Decorator patterns.

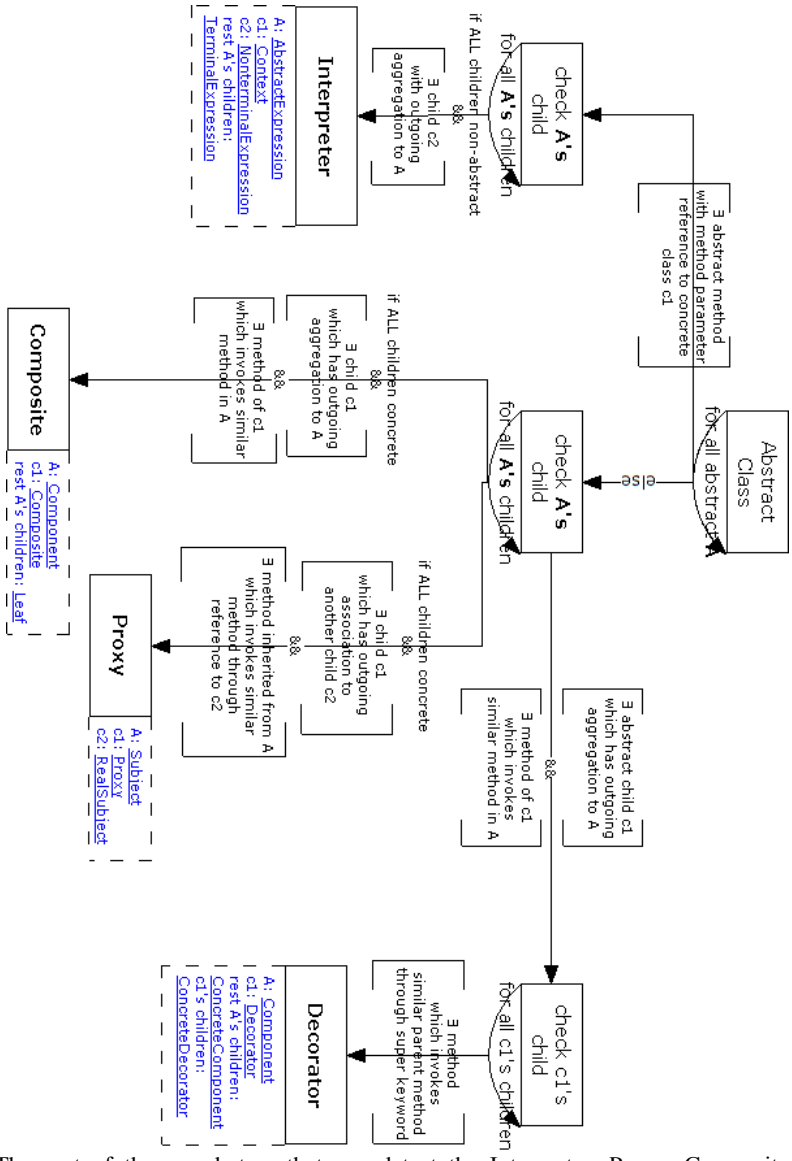

Fig. 7. The part of the search tree that can detect the Interpreter, Proxy, Composite and Decorator patterns.

Finally, we have a separate diagram in Figure 8 for the detection of the Memento pattern, which is a special case.

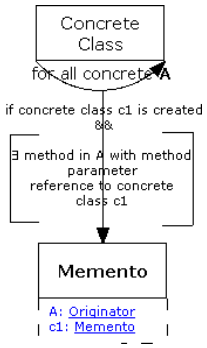

Fig. 8. The diagram related to the detection of the Memento pattern.

To make the parts present in the diagram more clear we include the pseudocode for the Visitor pattern. We have chosen the Visitor pattern, since we concluded that it is one of the patterns that have the largest algorithmic complexity to detect. The pseudocode follows the Cormen, Leiserson, Stein and Rivest [4] conventions.

```
for each abstract class A
 do i ← 1
    repeat
        mpr ← MethodParamReferences(A, i)
        if mpr is reference to abstract class c1
          then isvisitor ← TRUE
                k ← 1
                l ← 1
                repeat
                   c1ch ← Children(c1, k)
                   repeat
                      ach ← Children(A, l)
                      m ← 1
                      methodparamreferenceexists ← FALSE
                      repeat
                        mprch ← MethodParamReferences(c1ch, m)
                        if mprch is method param reference to ach
                          then  methodparamreferenceexists ← TRUE
                      until (mprch=NIL) or (methodparamreferenceexists=true)
                      if (methodparamreferenceexists=FALSE)
                        then isvisitor=FALSE
                      l ← l+1
                   until (c1ch=NIL) or (isvisitor=FALSE)
                   k ← k+1
               until (ach=NIL) or (isvisitor=FALSE)
               i ← i+1
    until (isvisitor=FALSE) or (mpr=NIL)
    if (isvisitor=TRUE)
      then print "Visitor"
```

Fig. 9. The pseudocode for the detection of the Visitor pattern.

As it can be easily inferred from the pseudocode the algorithmic complexity of detecting the Visitor pattern is $O(n^5)$ where n is the number of rows or columns we examine in the matrix representation of the system.

The results of the computational complexity analysis for all 20 GoF patterns are presented in Table 10.

Table 10. Summary of the computational complexity of the algorithms for detecting the patterns examined.

Pattern	Complexity
Bridge	$O(n^2)$
State	$O(n^2)$
Strategy	$O(n^2)$
Command	$O(n^4)$
Builder	$O(n^4)$
Mediator	$O(n^5)$
Observer	$O(n^5)$
Visitor	$O(n^5)$
Chain	$O(n^2)$
Factory Method	$O(n^5)$
Prototype	$O(n^2)$
Iterator	$O(n^5)$
Adapter (class)	$O(n^3)$
Adapter (object)	$O(n^3)$
Interpreter	$O(n^4)$
Composite	$O(n^3)$
Proxy	$O(n^3)$
Decorator	$O(n^5)$
Abstract Factory	$O(n^5)$
Memento	$O(n^2)$

We have mentioned and seen that the highest computational complexity of the algorithms for detecting one of the design patterns examined is $O(n^5)$. However, in practice, the degree of the actual polynomial expression will be lower. For example, considering the tree in Fig. 2, the number of abstract classes in the system will normally be lower than n/2. Moreover, in searches through all descendants of a parent class, the number of children will be substantially lower than the number of classes in the system (corresponding to n). In other words, it is unlikely that a class has as children all other classes of the system.

4 Conclusions and Future Work

In this paper we presented a novel approach to automated design pattern detection. The main characteristics of this approach are its simplicity and the fact that it is intuitively appealing. Specifically we represent design patterns as well as the system under consideration as sets of matrices and the search for the patterns inside the

system is performed based on trees that encode all the required information. Furthermore, by using this specific technique most of the GoF patterns can be detected, namely 20 out of 24. Additionally, the computational complexity of the specific approach is not prohibitive.

As future steps in our approach we are working on a tool that can read the class diagram in XMI format [9] and uses the techniques described in this paper to detect GoF patterns. Furthermore, we would like to extend our approach so that it can detect patterns even when refactorings [6] have been applied to them.

Additionally, in the long term we aim at a methodology that will assist the designers by automatically suggesting specific design patterns in appropriate places, with the aim of improving the quality of a software design.

References

1. Bergenti, F., and Poggi, A. Improving UML Designs using Automatic Design Pattern Detection, In Proceedings of the 12th International Conference on Software Engineering and Knowledge Engineering (SEKE 2000)
2. Booch, G., Rumbaugh, J., and Jabobson, I., The Unified Modeling Language User Guide, Addison Wesley, 1998
3. Brown, K., Design Reverse-Engineering and Automated Design Pattern Detection in Smalltalk. Available at http://www2.ncsu.edu
4. Cormen, T. H., Leiserson, C. E., Stein, C. and Rivest, R. L., Introduction to Algorithms, 2nd edn., MIT Press 2001
5. Crochemore, M., and Lecroq, T., Chapter 6 of Handbook of Computer Science and Engineering, Pattern Matching and Text Compression Algorithms
6. Fowler, M., Refactoring: Improving the Design of Existing Code. Addison Wesley, 1999
7. Gamma, E., Helm, R., Johnson, R., and Vlissides, J., Design Patterns. Addison Wesley, 1995
8. Heuzeroth, D., Holl, T., Högström, G, and Löwe, W., Automatic Design Pattern Detection, In Proceedings of the 11th IEEE International Workshop on Program Comprehension (IWPC 2003)
9. OMG, XMI Catalog of OMG Modeling and Metadata Specifications, http://www.omg.org/technology/documents/modeling_spec_catalog.htm
10. Prechelt, L. and Krämer, C., Functionality versus Practicality: Employing Existing Tolls for Recovering Structural Design Patterns
11. Rich, C. and Waters, R. The Programmer's Apprentice: a Research Overview, IEEE Computer, 21(11), November 1998, pp. 11-24
12. Smith, J. M., An Elemental Design Pattern Catalog. Technical Report TR-02-040, Univ. of Carolina
13. Smith, J. M., and Stotts, D., SPQR: Flexible Design Pattern Extraction from Source Code. Technical Report TR-03-016, Univ. of Carolina
14. Wendehals, L., Improving Design Pattern Instance Recognition by Dynamic Analysis. In Proceedings of the International Conference on Software Engineering 2003 (ICSE 2003)