Design Patterns

Supporting design process by automatically detecting design patterns and giving some feedback

E.M. van Doorn

Student: 850883525
Date: 24/08/2016

Open Universiteit
www.ou.nl

DESIGN PATTERNS

SUPPORTING DESIGN PROCESS BY AUTOMATICALLY DETECTING DESIGN PATTERNS AND GIVING SOME FEEDBACK

by

E.M. van Doorn

in partial fulfillment of the requirements for the degree of

Master of Science
in Software Engineering

at the Open University, faculty of Management, Science and Technology
Master Software Engineering
to be defended publicly on Wednesday August 24, 2016 at 14:00 PM.

Student number:	850883525	
Course code:	T9232B	
Thesis committee:	Prof. dr. M. C. J. D. van Eekelen (chairman),	Open University
	Dr. ir. S. Stuurman (supervisor),	Open University

An electronic version of this thesis is available at http://dspace.ou.nl/.

Open Universiteit
www.ou.nl
de beste!

DEDICATION

This thesis is dedicated to my parents and my wife.
For my late parents, whose support I still experience.
For my wife, who listened endlessly to my struggles.

Acknowledgement

First of all, I would like to thank Marko van Eekelen and Sylvia Stuurman. They have spent so much time to my guidance. I am also very grateful for offering this research subject and opportunity to me. This research is in line with my interests and talents. By their guidance, I have worked with pleasure to this research.
Rene de Winter, my former teamleader, I am grateful for offering the opportunity to start this study. I would like to thank Matthieu Karel, former coordinator of OU activities of De Haagse Hogeschool. He pointed out to me that, I could also choose a software engineering subject. My colleagues Rene Vogels and Okan Zor, I am grateful for their advices and interest.
For offering the LaTeXtemplate, I thank former OU student Patrick Molijn.
For moral support and patience, I am grateful to my wife. She missed me.

Contents

List of Figures **vi**

List of Tables **vii**

1 Introduction **1**
- 1.1 Research question . . . 1
- 1.2 Scope . . . 2
- 1.3 Research method . . . 2
- 1.4 Thesis overview . . . 2

2 State of the art **3**
- 2.1 What is a design pattern . . . 3
- 2.2 Design pattern characteristics . . . 4
- 2.3 Design pattern detection . . . 4
  - 2.3.1 Template matching . . . 4
  - 2.3.2 Sum Of Products Expressions . . . 10
  - 2.3.3 Rule based / Prolog . . . 11
  - 2.3.4 4-tuples . . . 12

3 Selection detection method **14**

4 Matching algorithm: Implementing and experimenting **18**
- 4.1 Plan . . . 18
- 4.2 Implementing a matching algorithm . . . 19
- 4.3 Experiments . . . 21
- 4.4 Improvements for practical usage . . . 30

5 Comparing to related work **33**
- 5.1 Aggregate Comparison . . . 33

6 Conclusions and Future work **37**
- 6.1 Conclusions . . . 37
- 6.2 New results . . . 38
- 6.3 Future work. . . . 39

A Appendix Approaches to find submatrices **40**

B Appendix Tutorial for detection software **44**

C Appendix Deliverables **47**
- C.1 quickstart.zip . . . 47
- C.2 netbeansproject.zip . . . 48
- C.3 xmi.zip and zargo.zip . . . 48

D Appendix Legend UML class diagram **50**

Bibliography **52**

Books 52

In Books 52

Academic Articles 52

In Proceedings 53

Technical Documentation 54

LIST OF FIGURES

2.1 Example class diagram –> graph 5
2.2 Blondel’s algorithm 7
2.3 Similarity between graphs; copied from [DSZ08] 8
2.4 Example Ba-Brahem 12

4.1 Example Ba-Brahem 22
4.2 Bridge pattern 23
4.3 Bridge pattern 24
4.4 Abstract Factory template and an implementation 25
4.5 13 design patterns 29
4.6 Class diagram used for pattern detection and performance 32

D.1 Legend of UML symbols in class diagrams 50
D.2 Legend of UML symbols in class diagrams 51

List of Tables

2.1 Generalization matrix 5
2.2 Realization vector for interface Interf1 5
2.3 Similarities according Blondel 10
2.4 Node similarities according Zager 10
2.5 Edge similarities according Zager 10
2.6 SOP-expressions of the system under consideration of figure 2.4 11
2.7 SOP-expressions of the Prototype pattern of figure 2.4 11
2.8 Corresponding relationships 13

3.1 Evaluation detection methods 17
3.2 Explanation of table 3.1 17

4.1 Permutation of prototype in the example of Ba-Brahem 22
4.2 Match between the Bridge pattern with one missing relationship and the example of Ba-Brahem 23
4.3 Frequencies of design patterns in example of Ba-Brahem 24
4.4 Gang of Four Patterns 27
4.5 Summarized results of the detection algorithm 27
4.6 Frequencies of detected design patterns in figure 4.6 31

5.1 Software comparison 36

B.1 Possible values of typeRelationship 45

C.1 Content of quickstart.zip 47

SUMMARY

Since the nineties, design patterns are of interest. Since the beginning of this century, research is done to the possibilities of recognizing design patterns in source code. There is little research to recognizing design patterns in UML diagrams and possibilities to provide feedback on those designs. This has led to the following research question: *Are design patterns in a UML class diagram automatically detectable, and can one automatically supply some feedback?* The research question is answered by a literature review followed by creating and testing a prototype, which recognizes design patterns and provides some feedback.

In literature UML class diagrams and design patterns are modelled using matrices, decision trees, boolean expressions, Prolog clauses and 4-tuples. Dong describes a method, by which the various types of relationship can be denoted in one matrix. In his article, however, there is no description of the algorithm by which a design pattern can be detected.[DSZ08]

Unsuccessfully attempt has been made to write a detection algorithm by applying the steepest descent method and Richardson iteration to the method of Dong.

Next, four methods for detecting design patterns are compared. The corresponding articles, including the article of Ba-Brahem, contain no description of an algorithm by which design patterns can be detected. The method of Ba-Brahem, which is based on the use of 4-tuples, seemed most promising. This method also offers the possibility to detect design patterns, which are only partially present. It can also used to give feedback on missing relationships.

For the method of Ba-Brahem I have designed and implemented an algorithm, which is able to detect design patterns. The prototype, which implements the method of Ba-Brahem is able to detect 17 of the 23 Gang of Four patterns. Within one second 13 different design patterns are detected in a class diagram, which contains 57 classes en 61 relationships. See figure 4.5 at page 29. In a second experiment a class diagram represented by a XMI files, which contains 33 classes and 49 associations, was used. The XMI file orginates from ArgoUML. This tool is used to design class diagrams. Within one second 17 different design patterns were detected. See figure 4.6 at page 32. The prototype is also able to detect design patterns, which are partially present in a class diagram. When a class diagram includes a number of classes and interfaces, which form a design pattern, feedback is specified about the relationships between these classes and interfaces, which are not part of the design pattern.
The prototype has made it clear, that instead of a 4-tuple a 3-tuple can be used. One of the attributes of a 4-tuple is intended to indicate whether a class has a self-reference. However, this attribute provides no contribution for the recognition of the Singleton design pattern. An essential characteristic is a self-reference of an *object* and not a self-reference of a class. The prototype is compared with four programs, which are described in literature. Only one

program provides feedback, which is more comprehensive than the feedback from my prototype. Two programs have a graphical user interface, which displays the detected design patterns. How a design patterns is read, is either not disclosed or less simple compared to my prototype. For other issues, such as, recall, precision and performance, the score my prototype is at least equal. The Open University considers to make my prototype user friendly, such that it can be used for educational purposes.

SAMENVATTING

Vanaf de negentiger jaren staan design patterns in de belangstelling. Sinds het begin van deze eeuw is onderzoek gedaan naar de mogelijkheden om design patterns in de code te herkennen. Er is nog weinig onderzoek verricht naar het herkennen van design patterns in UML diagrammen en de mogelijkheden om op een ontwerp feedback te geven. Dit heeft geleid tot de volgende onderzoeksvraag: *Zijn design patterns in een UML class diagram automatisch detecteerbaar en kan op herkende design patterns automatisch enige feedback worden gegeven?* De vraag is beantwoord door een literatuurstudie gevolgd door het maken en testen van een prototype, die design patterns herkent en feedback geeft.

In de literatuur worden UML class diagrammen en design patterns gemodelleerd met behulp van matrices, beslissingsbomen, boolean expressies, Prolog clauses en 4-tuples. Dong heeft een methode beschreven, waarmee de diverse relatietypes in één matrix kunnen worden weergegeven. In zijn artikel ontbreekt echter een beschrijving van het algoritme, waarmee een design pattern kan worden gedetecteerd.[DSZ08]

Zonder succes is getracht om met behulp van steepest descend methode en Richardson iteration een detectie algoritme te schrijven, die op de methode van Dong toegepast kan worden.

Vervolgens is een viertal methodes om design patterns te detecteren vergeleken. De betreffende artikelen, waaronder die van Ba-Brahem, bevatten geen beschrijving van een algoritme waarmee design patterns gedetecteerd kunnen worden. De methode van Ba-Brahem, die gebaseerd is op het gebruik van 4-tuples, leek het meest kansrijk. Deze methode biedt ook de mogelijkheid om design patterns, die slechts gedeeltelijk aanwezig zijn, te herkennen. Als feedback kunnen de ontbrekende relaties eenvoudig worden weergegeven.

Voor de methode van Ba-Brahem heb ik een algoritme beschreven en geïmplementeerd waarmee design patterns kunnen worden gedetecteerd. Het prototype, die de implementatie van de methode van Ba-Brahem bevat, kan 17 van de 23 Gang of Four patterns herkennen. In een class diagram met 57 classes en 61 relaties zijn 13 verschillende design patterns binnen één seconde herkend. Zie figuur 4.5 op blz. 29. In een tweede experiment is een class diagram met 33 classess en 49 associaties als XMI file ingelezen. Deze file is afkomstig van ArgoUML, een tool die gebruikt wordt om class diagrammen te ontwerpen. Hierin zijn binnen één seconde 17 verschillende design patterns herkend. Zie figuur 4.6 op blz. 32. Tevens kunnen design patterns, die gedeeltelijk aanwezig zijn in een class diagram, worden herkend. Als een class diagram een aantal classes en interfaces bevat die een design pattern vormen, kan als feedback worden aangegeven, welke relaties tussen deze classes en interfaces geen onderdeel zijn van het design pattern.
Het prototype heeft duidelijk gemaakt dat in plaats van een 4-tuple een 3-tuple gebruikt kan worden. Een van de attributen van een 4-tuple is bestemd voor het aangeven of een

class een self-reference heeft. Dit attribuut levert echter aan het herkennen van het Singleton design pattern geen bijdrage. Het is namelijk essentieel voor dit design pattern, dat een *object* een self-reference heeft.

Het prototype is vergeleken met een viertal programma's die in de literatuur zijn beschreven. Slechts één programma kan feedback geven die bovendien uitgebreider is dan de feedback van mijn prototype. Twee programma's hebben een grafische user interface, waarmee de resultaten worden getoond. Het invoeren van design patterns is óf niet beschreven óf minder eenvoudig dan met mijn prototype. Op de overige punten, zoals recall, precision en performance scoort mijn prototype minstens gelijkwaardig. De Open Universiteit overweegt mijn prototype gebruikersvriendelijk te maken en voor het onderwijs in te zetten.

1 Introduction

This chapter introduces the research and gives an overview of the next chapters.
Simula is probably the first object oriented language, which was introduced in the sixties. Object oriented programming became popular after introduction C++ during the late 80's. After introduction of OMT by Rumbaugh, object oriented analysis and design became an important modelling technique. UML was introduced about 1995 and is widely accepted as a general object modelling language.
Caused by the popularity of object orientation, recurrent design problems occurred. Recycling the answers to this problems improved the quality of analysis and design. In the 90's a well known book about recurrent design problems and answers is published [Gam+95]. This book is still the standard book about the design patterns. Professionals frequently use design patterns and therefore design patterns are an educational subject.
This thesis is about detecting design patterns in UML class diagrams and also about giving feedback on UML class diagrams, which comprise design patterns.

1.1. Research question

The **research goal** is to conduct a survey of possibilities to

- automatically detect design patterns in an UML class diagram
- give feedback on possible presence of design patterns in an UML class diagram

If feasible, this will result in a proof of concept written in Java.

The **research question** is:

Are design patterns in a UML class diagram automatically detectable, and can one automatically supply some feedback?

Subquestions are:

- What is a design pattern?
- What are the characteristics of a design pattern in an UML class diagram?
- Which methods to detect design patterns are described in literature?
- Which and how feedback can be supplied, based on integral or partially existence of a design pattern in an UML class diagram?

1.2. SCOPE

By using sequence diagrams or source code, the dynamic behaviour can be used for detecting design patterns. This kind of research is, however already done [Wen03].
If feasible, a prototype will be made which is enable to detect design patterns, give feedback and use class diagrams which originate from ArgoUML [1]. This choice is made, because ArgoUML is used by the OU. The prototype may be used in future by the OU.

1.3. RESEARCH METHOD

The goal of this research is to deliver a prototype or a proposal for detecting design patterns and giving feedback. This research is:

- a constructive research, when a prototype is made and tested [Crn10] –or–
- theoretical, when it ends with a proposal.
  In this case the feasibility has to be justified on theoretical grounds.

The research method is therefore quite simple. After studying literature to answer the first three subquestions, a choice has to be made between implementing a prototype or trying to formulate a proposal, by which design patterns can be detected and feedback can be given.
Depending on the choice, a plan for the next phase will be made. This second plan will be about, implementing and testing, or continuing the search for literature and formulating a feasible proposal.

1.4. THESIS OVERVIEW

This master thesis is organized as follows. Chapter 2 provides an overview of literature about detecting design patterns. The chapter 3 describes how the detections are compared and why the 4-tuple method is chosen to implement. Chapter 4 descibes the matching algorithm and the results of the experiments. In chapter 5 my implementation is compared to implementations of other researches. The last chapter contains the conclusions of this research, some new insights and proposals for future work.

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

2
STATE OF THE ART

This chapter provides an overview of related work and answers to three research subquestions.

2.1. WHAT IS A DESIGN PATTERN

Many skills and disciplines have frequently recurring problems. Based on experience, well known general solutions to each of those problems exist. Software design has also recurring problems. The *description* of the problem and a well known general reusable solutions is called a design pattern.

Design patterns are formally defined as:
'Design patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.' [Gam+95]

The book written by Gamma et al. may be seen as the standard work of design patterns. It contains 23 design patterns, which are classified in three groups: creational, structural and behavioural. In honour of the four authors, the 23 design patterns are called Gang of Four (GoF) patterns. Nowadays, there are many more design patterns, which are used in different levels of software engineering:

1. Architectural
   These patterns are used to define fundamental structures of the software systems.

2. Design
   These patterns are used to design subsystems and components.

3. Idiom
   These patterns are used at programming level.

[Cop97]

This research focusses on the second level, to which all 23 design patterns belong.

2.2. DESIGN PATTERN CHARACTERISTICS

This section provides a number of *measurable* characteristics, which are denoted in literature to identify design patterns.

The characteristics are based on:

> ’The design pattern identifies the participating classes and instances, their roles and collaborations, and distribution of responsibilities.’ [Gam+95]

Characteristics, which are named in research are class, abstract class, interface, association, generalization, aggregation, creation, invoked and inherited method, similar method invocation, method parameter reference [DSZ08] [YWG04] [Tsa+05]. The meaning of most the characteristics is obvious. However, creation relates to the dependency in which an object of the class is created within another class. If two classes have methods with the same signature, then these methods are called similar methods. When a method has a parameter from another class, it is called a method parameter reference, which is also a dependency.

It is unclear, whether these characteristics supply sufficient information to distinguish all GoF patterns. Some researchers claim to detect all GoF patterns. However, the design patterns, Strategy and State pattern are structurally identical, and vary only in their behaviour [Nie+02]. To detect the Singleton, Flyweight, and Template Method patterns code specific information is needed. The detection of the Facade pattern is impossible, because of its abstract nature [Tsa+05].

Based on the above given framed quotation of design pattern characteristics, *collaborations, and distribution of responsibilities* are partly covered by association, generalization, and aggregation. Responsibility is also a matter of implementation, and therefore not applicable to this research.

2.3. DESIGN PATTERN DETECTION

For many years the problem of detecting design patterns is investigated. The first attempt to automatically detect design patterns is performed by Kyle Brown [Bro96] in 1996. [Tsa+06] [RG13] Still, a perfect solution is not found. In this section a number of approaches is given to automatically detect design pattern in UML class diagrams.

2.3.1. TEMPLATE MATCHING

Template matching is a technique to find a part/template in an environment, which resembles the given part/template. In computer vision, such techniques are used to find a part in pictures. A similar problem is the search for a subgraph in a graph, which is equivalent to the search for a submatrix in a matrix. A class diagram and therefore a design pattern may be represented by matrices and vectors.

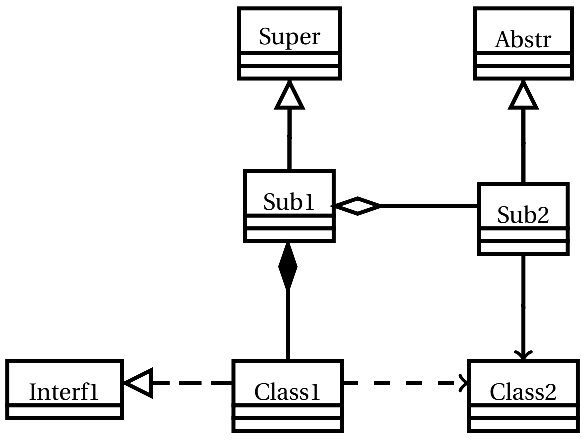


Figure 2.1: Example class diagram –> graph

A legend of the symbols of an UML class diagram is given in appendix D.

The characteristics: associations, aggregations, composites, dependencies, generalizations between classes, may be represented by square matrices. An example is table 2.1 for figure 2.1. For every interface, a vector denotes for every class whether the class implements the interface. An example is table 2.2 for figure 2.1. There is one vector, which describes for every class whether the class is abstract.

The (non)existence of an edge in a graph corresponds to the value zero or one in the matrix.

	Abstr	Class1	Class2	Interf1	Sub1	Sub2	Super
Abstr	0	0	0	0	0	0	0
Class1	0	0	0	0	0	0	0
Class2	0	0	0	0	0	0	0
Interf1	0	0	0	0	0	0	0
Sub1	0	0	0	0	0	0	1
Sub2	1	0	0	0	0	0	0
Super	0	0	0	0	0	0	0

Table 2.1: Generalization matrix

	Interf1
Abstr	0
Class1	1
Class2	0
Interf1	0
Sub1	0
Sub2	0
Super	0

Table 2.2: Realization vector for interface Interf1

It is possible to combine these matrices and vectors into one *overall matrix*. Combining is realized in two steps.

- The vectors are extended to square matrices.
  The new columns only consist of zeroes.
- The overall matrix $ov_{i,j} = 2^{associationMatrix_{i,j}} * 3^{aggregationMatix_{i,j}} * 5^{\cdots} * 7^{\cdots} * 11^{\cdots}$ [DSZ08]

Such overall matrices may be constructed for the system under consideration and the design pattern. So, finding a design pattern in a class diagram, is equivalent to finding a submatrix in a matrix.

A design pattern, (template) may not exactly match a part of a class diagram. Therefore a measure of similarity between the pattern and the part is needed. In computer vision normalized cross validation is often used to determine similarity. Two matrices could be compared by normalized cross validation

$$CCn = \frac{\sum f(x) \cdot g(x)}{|f(x)|\,|g(x)|}$$

where $f(x)$ and $g(x)$ are vectors.
However, this approach is only useful, if the order of rows and columns, and therefore the classes and interfaces correspond. Otherwise, every permutation of the order of rows and columns has to be compared, which is time consuming. There are several ways to circumvent the problem.

Dong et al. preprocessed the overall matrix to limit the space search. For instance, all rows and columns which consist only of one's are removed. This follows from: if an element in the overall matrix $ov_{i,j} = 1$ then all the exponents are zero. Therefore, there are no relationships between the classes and interfaces.
The method is applied to the sources of JUnit, JHotDraw, JRefactory and Log4j. Multiple instances, and variants of four design patterns are recognized.[DSZ08].

Tsantalis et al. build a huge decision tree, containing 20 GoF design patterns, decision points and conclusions (a design pattern). To make a decision, nine characteristics are available.[Tsa+05].

Another way to circumvent the problem of comparing many permutations of rows and columns, is to use Blondel's algorithm. This algorithm is only applicable to adjacency matrices, which contains only one's en zeroes, and not for the overall matrix. Blondel's algorithm calculates the similarity for *every* pair of classes.

BLONDEL'S ALGORITHM

Blondel's algorithm is based on the concepts of authorities and hubs in *one* graph, which represents websites. An authority is the origin of information, and a hub points to authorities. The quality or score of an authority is defined as the sum of the scores of hubs pointing to the authority. And vice versa: the quality or score of a hub is defined as the sum of the scores of the authorities to which it points. This leads to a recurrence relation between the

scores of hubs and authorities.
This idea may be generalized. See figure 2.2

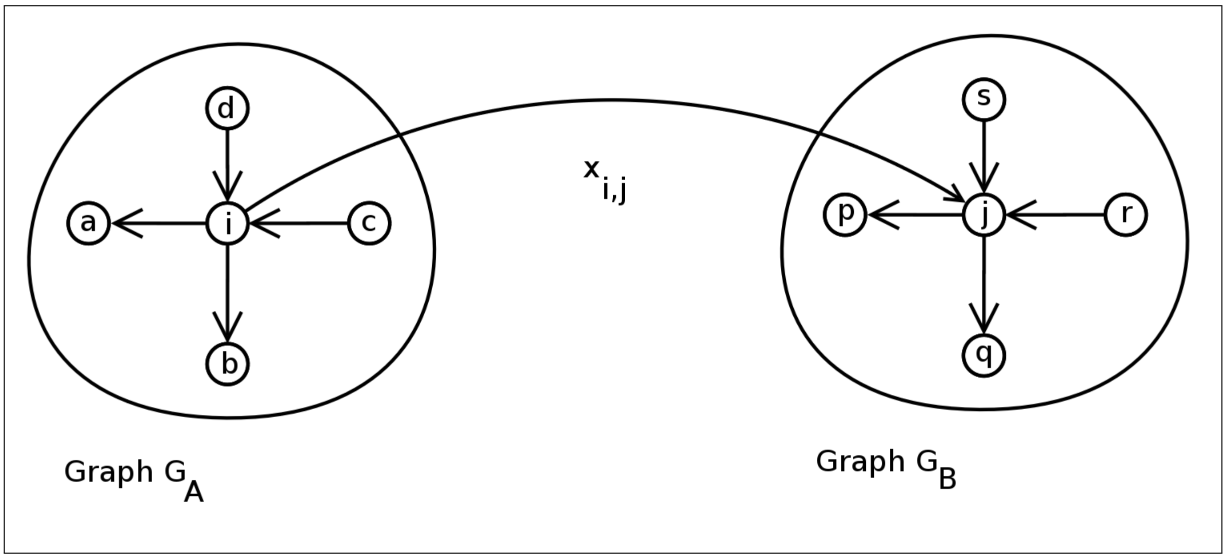


Figure 2.2: Blondel's algorithm

In this figure G_A and G_B corresponds to a design pattern and a system under consideration. The similarity between every pair of vertices have to be determined.

Instead of websites pointing to each other, now the vertices of graph G_A point to G_B : if there is an edge (a_i, a_j) in G_A, and there is an edge (b_i, b_j) in G_B then, there is an edge between (a_i, b_i) and (a_j, b_j) in $\{G_A \times G_B\}$.
For every element of the adjacency matrix $\{G_A \times G_B\}_{ij}$ a score x_{ij} is defined as: sum of the scores of the incoming and outgoing scores of the vertices i and j.

In this example: The similarity between vertices i and j can be calculated by:
$x_{ij} = (x_{cr} + x_{cs} + x_{dr} + x_{ds}) + (x_{ap} + x_{aq} + x_{bp} + x_{bq})$
The formulas for the similarities between the other vertices can be calculated in a similar way.

In general this results in the following recurrent equation:

$$X_{k+1} = BX_kA^T + B^TX_kA,$$

where A and B are the adjacency matrices of G_A and G_B and X = $\{x_{ij}\}$.
The normalized equation is:

$$Z_{k+1} = \frac{BZ_kA^T + B^TZ_kA}{\|BZ_kA^T + B^TZ_kA\|_F}$$

The symbol F indicates the Frobenius norm: $\|A\|_F = \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{m} a_{ij}^2}$

z_{ij} is defined as the similarity of vertex i of A and j of B.

The similarities are approximated by the algorithm:

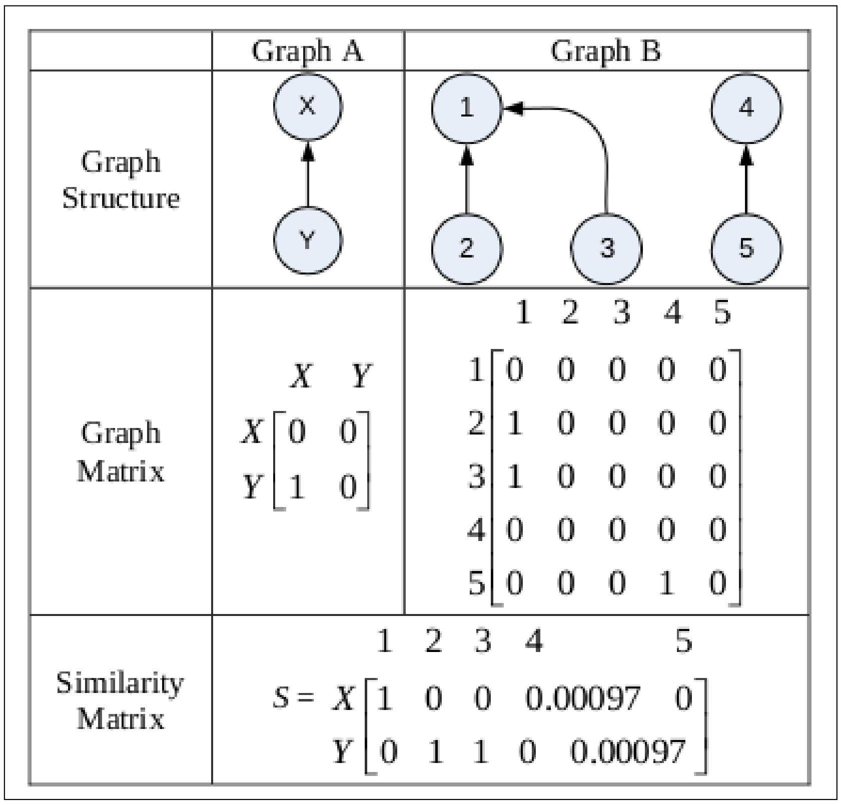
1. $Z_0 = \mathbf{1}$ (a matrix which contains only ones)
2. Calculate an *even* number of times:

$$Z_{k+1} = \frac{BZ_kA^T + B^TZ_kA}{\left\|BZ_kA^T + B^TZ_kA\right\|_F}$$

   until Z has sufficiently converged.

This approach is used by two groups of researchers [GD12] [DSZ08].

Figure 2.3 shows the idea in a very simplified version. The first matrix represents a design pattern: graph A. The second matrix represents the system under consideration: graph B.



$$\begin{array}{c|cc} & X & Y \\ \hline X & 0 & 0 \\ Y & 1 & 0 \end{array} \qquad \begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 0 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 & 0 & 0 \\ 3 & 1 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 1 & 0 \end{array}$$

$$S = \begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline X & 1 & 0 & 0 & 0.00097 & 0 \\ Y & 0 & 1 & 1 & 0 & 0.00097 \end{array}$$

Figure 2.3: Similarity between graphs; copied from [DSZ08]

Instead of the Frobenius norm Dong, uses the 1-norm for every column ($\left\|X\right\|_1 = \sum_{i=1}^{n} |x_i|$).
Therefore, if a column has only one number greater than zero, its value is one. In the next paragraph the results are given for the Frobenius norm.
Two perfect matches between {X, Y} and a subset of {1, 2, 3, 4, 5} can be made: (X, Y) → (1,

2) and (X, Y) → (1, 3). The third match: (X, Y) → (4, 5) however is not discovered. My implementation of the algorithm confirms this fault. Brief experiments give reason to suspect that the fault may only occur, if B is not a connected graph.
Making the match between {X, Y} and a subset of {1, 2, 3, 4, 5} is not described by the two groups of researchers [GD12] [DSZ08]. Making the match is an example of the assignment problem, which belongs to the field of Operation Research, and can be solved by a standard algorithm: the Hungarian method [Tah92].

This approach is able to detect more instances of a design pattern in a class diagram. It is also able to detect variants of design patterns. Like humans, the approach has difficulties to detect design patterns if they intersect [DSZ08].

Tsantalis et al. apply this approach to the Java frameworks JHotDraw, JRefactory and JUnit. The methods efficiency is improved by reducing the size of the graphs, based on the idea that many design patterns involve class hierarchies which communicate. The hierarchies are separated for individual investigation. The approach is able to detect 10 design patterns within the frameworks. It is not clear, whether 10 is the maximum number of detectable patterns [Tsa+06].

ZAGER'S ALGORITHM

An extension of Blondel's algorithm is described by Laura Zager [ZV08]. Blondel's algorithm is based only on the similarity of nodes and the initial **1**−matrix. Besides similarity of nodes, Zager also uses the similarity of edges. An edge in G_A is similar to an edge in G_B, if their source and terminal nodes are similar.
Before the algorithm can be defined, I will first give some necessary definitions.

For the adjacency matrix A, A_S and A_T are defined by:

$$[A_S]_{ij} = \begin{cases} 1 & \text{if } j \text{ is the source node of edge } i \\ 0 & \text{otherwise} \end{cases}$$

$$[A_T]_{ij} = \begin{cases} 1 & \text{if } j \text{ is the target node of edge } i \\ 0 & \text{otherwise} \end{cases}$$

Like Blondel's algorithm, x_{ij} defines the similarity score of node i in G_A and node j in G_B. The similarity score for the *edge* with node i in G_A and node j in G_B is denoted as y_{ij}. The iterative values of x_{ij} and y_{ij} are calculated by:

$$Y_{k+1} = \frac{B_S^T X_k A_S + B_T^T X_k A_T}{\left\| B_S^T X_k A_S + B_T^T X_k A_T \right\|_F}$$

$$X_{k+1} = \frac{B_S Y_k A_S^T + B_T Y_k A_T^T}{\left\| B_S Y_k A_S^T + B_T Y_k A_T^T \right\|_F}$$

$$\text{where } X = \{x_{ij}\} \text{ and } Y = \{y_{ij}\}$$

The similarity scores are approximated by the algorithm:

1. Choose an arbitrary initial matrix X_0, and an arbitrary positive value α.
2. Calculate $Y_0 = \alpha\ X_0$
3. Iterate Y_{k+1} and X_{k+1}, until they has sufficiently converged.

When the algorithms of Blondel (with the Frobenius norm) and Zager are applied to figure 2.3, the similarity matrices are:

	1	2	3	4	5
X	0.57735	0.00000	0.00000	0.00000	0.00000
Y	0.00000	0.57735	0.57735	0.00000	0.00000

Table 2.3: Similarities according Blondel

	1	2	3	4	5
X	0.81650	0.00000	0.00000	0.00000	0.00000
Y	0.00000	0.40825	0.40825	0.00000	0.00000

Table 2.4: Node similarities according Zager

	2 → 1	3 → 1	5 → 4
Y → X	0.70711	0.70711	0.00000

Table 2.5: Edge similarities according Zager

Like the algorithm of Blondel, the matrices according to Zager's algorithm, do not show any similarity at all between the nodes and edges (X, Y) and (4, 5).

2.3.2. SUM OF PRODUCTS EXPRESSIONS

This approach is based on the representation of relationships by means of Sum Of Products. A Sum of products is a number of boolean terms, which are OR-ed. Each term has boolean values, which are AND-ed.
Example of SOP-expression:

$$\begin{aligned} f(A,B,C) &= (A \wedge B \wedge C) \vee (\bar{A} \wedge B \wedge \bar{C}) \\ &= (111 \vee 010) \\ &= \sum (111, 010) \end{aligned}$$

Instead of the boolean operations $\wedge$ and $\vee$ the operations . and + may be used. So $f(A,B,C) = A.B.C + B.C$

Manjari Gupta proposed the following approach [GPT11]. For the types of relation inheritance, association, aggregation, dependency SOP-expression can be made. This types of relation are transitive. Which means in example for inheritance: if A inherits from B, and B

inherits from C, then A inherits from C.
For figure 2.4 the SOP-expressions of the system under consideration are denoted as:

System under consideration	
SOP(directed association)	= A.B + A.C.B
SOP(generalisation_1)	= D.B
SOP(generalisation_2)	= D.E
SOP(dependency)	= E.C

Table 2.6: SOP-expressions of the system under consideration of figure 2.4

Prototype pattern	
SOP(directed association)	= P.Q
SOP(generalisation)	= R.Q

Table 2.7: SOP-expressions of the Prototype pattern of figure 2.4

Detecting a design pattern in a class diagram implicates that all the SOP-expressions of the design pattern should occur in the SOP-expressions of the class diagram.

Algorithms to systematically search for corresponding SOP-expressions and to match the results of the search are not given. This approach is not implemented, through which experimental results are missing. Another limitation is the impossibility to represent a relationship for a class to itself. As a result, the Singleton pattern cannot be discovered.

The approach is able to detect multiple occurrences and variants of design patterns.

Manjari Gupta also wrote an article of Product Of Sums, which is almost identical to her article about Sum Of Products. The conclusion in her second article even contains the phrase 'into sum of product (POS)' instead of 'into product of sums (POS)'
The difference is the interchange of sums and products.

This second article does not describe the search and matching algorithm either. [GR14]

2.3.3. RULE BASED / PROLOG

Searching for a pattern is difficult and elaborate to implement. To circumvent this problem, one can use Prolog, because a general depth first searching algorithm is part of a compiled Prolog program. Prolog is a declarative language. A design pattern is declared by facts and rules. The participating classes are facts and the several types of relationships are represented by rules. This is done by Prechelt for the design patterns Adapter, Bridge, Composite, Decorator and Proxy [PK98]. Four benchmarks with 9 - 343 classes are conducted. Due to soft rules to describe patterns, a recall of 100% is realized, but the precision is at most 50%. The biggest benchmark has a precision of 41%, which took 36 seconds in 1998.

The research of Bergenti may be seen as a continuation of the aforementioned research. Besides, a class diagram Bergenti also uses collaboration diagram, which he uses to refine

the Prolog rules. His software could detect: Factory Method, Prototype, Abstract Factory, Composite, Decorator, Adapter, Bridge, Proxy, Observer, and Iterator.
When a pattern is detected, the software verifies a set of design rules. The rules produces feedback to improve the design. The software is integrated with ArgoUML so that real time feedback is given.[BP00]

2.3.4. 4-TUPLES

In this section a proposal of Ba-Brahem et al. is described to detect (incomplete) occurrences of design patterns [BQ14]. The concepts class and relationship correspond to vertex and edge. The terms *vertex* and *edge* are used in the environment of mathematics and algorithms. The terms *class* and *relationship* are used in the environment of design.

Instead of using adjacency matrices for representing all sorts of relationships, this approach uses 4-tuples. The tuple (A, B, C, D) stands for a relationship between the classes/interfaces A en B. C stands for the type of the relationship and D indicates a self-loop existence. Based on an example of Ba-Brahem, see figure 2.4, an outline of the proposal is given.

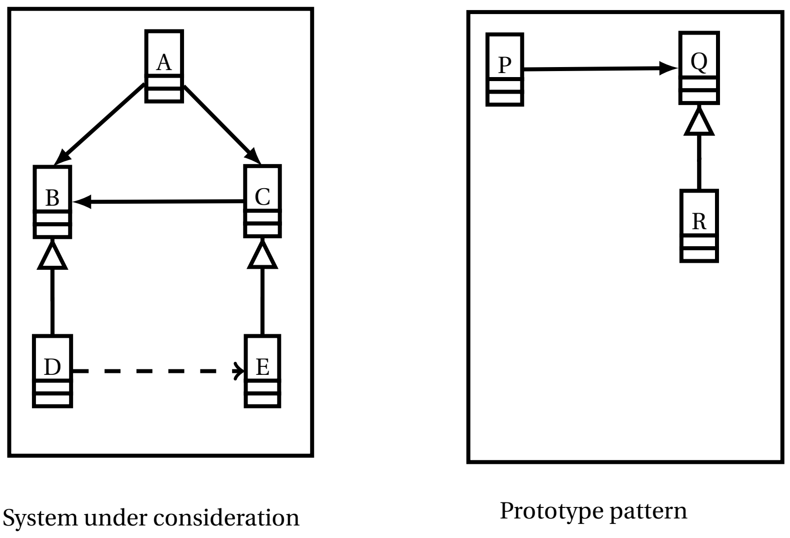


Figure 2.4: Example Ba-Brahem

The first step is building two sets of 4-tuples: SE an DPE, which represent the system under consideration and the design pattern.
SE = { (A, B, 1, 0), (C, B, 1, 0), (A, C, 1, 0), (D, B, 3, 0), (E, C, 3, 0),(D, E, 2, 0) }
DPE = {(P, Q, 1, 0), (R, Q, 3, 0)}
The number of edges of DPE is set to n: $n = |DPE|$.

The second step is iterative:
while $n > 0$
build a table, see table 2.8, which contains n edges of SE corresponding to edges in DPE.
In this example SE contains three occurrences of the design pattern. The algorithms continues with step 3.
If there is no row with n corresponding 4-tuples, n is decremented and step 2 is repeated.

Solution	design pattern: (P, **Q**, 1, 0)	(R, **Q**, 3, 0)
1	system: (A, **B**, 1, 0)	(D, **B**, 3, 0)
2	system: (A, **C**, 1, 0)	(E, **C**, 3, 0)
3	system: (C, **B**, 1, 0)	(D, **B**, 3, 0)

Table 2.8: Corresponding relationships

Step 3 gives the conclusion:

- If all elements of DPE exist in SE ($n = |DPE|$), then the system under consideration contains the design pattern. The number of rows equals the number of occurrences of the design pattern.
- if not all elements of DPE appear in SE ($0 < n < |DPE|$), the design pattern exists partially in the system under consideration.
- If $n = 0$ then the design pattern does not exist in the system under consideration at all.

They claim that 20 GoF patterns and multiple existences of patterns will be detected.

Their proposal has a weakness. If a very small number of elements of DPE appear in SE, which usually will be, a partially occurrence will be given. This correspondence may not be useful.

3 SELECTION DETECTION METHOD

In this chapter a choice is made between the methods for design pattern detection. First, the uncertainty of all data is described. Second, the criteria and the weight of the criteria for comparing detection methods are described. Third, implementable methods are chosen. Fourth, the values of implementable methods are given. At last, a choice is made.

Many articles which describe a method for detecting design pattern, have a lack of information. Some articles are merely a proposal for a method and therefore lack of experimental results. Some authors left behind an essential part of the detecting algorithm, by which the algorithm is hard to implement. In those cases, the evaluated value of the criterion is an educated guess.

The criteria for comparing detection methods, which are used in table 3.1 at page 17 are:

- Is the method implementable?
  A method may be hard to implement for several reasons. The description of the algorithm may be incomplete or very elaborate to implement. For instance, the algorithm described by Gupta is incomplete and Tsantalis uses a huge handmade decision tree. It is obvious, a method should be implementable to be of any use. Therefore, non-implemental methods will be rejected.

- Recall en precision
  These concepts are defined as:

$$Recall = \frac{t_p}{t_p + f_n}$$

$$Precision = \frac{t_p}{t_p + f_p}$$

  where $t_p, f_p\ and\ f_n$ are the numbers of true positives, false positives and false negatives
  Existence of a design pattern should be detected when possible. Incorrect recognition shall prejudice the confidence in the software. Therefore, the weight of recall and precision is high.

- Number of recognizable patterns
  Only a few authors mention the number of design patterns, which are recognized in their experiments. On before hand, some authors confined the number of design patterns in their research.
  Opposed to the number of recognizable design patterns, a high recall and precision is preferable, therefore the weight of this criterion is low.

- Variants of design patterns are detectable.
  A variant of a design pattern is a lookalike of the given template. For instance, a ordinary association may be used in stead of an aggregate association. To be of any use in educational and professional environments, variants have to be detectable and therefore the weight is high.

- Usable for feedback
  A follow-up of this research may be the elaboration of feedback messages. Giving feedback was not an issue in the discussed articles. In some cases, it is not clear whether feedback can easily be given or not. For instance, rule based methods will need for every deviation of a design pattern a separate rule. The number of rules will therefore increase multiple times.
  Taking into account the follow-up of this research, the criterion has a high weight.

To realize a usable system, only methods which are implementable in reasonable time should be considered.

Not implementable

- Template matching as described by Dong
  This method takes into account many characteristics of a design pattern. The occurrence of the characteristics are combined into one matrix. Although, this is a useful approach, the detection algorithm is unclear.
  Instead of using the sketched detection algorithm in one or another way, I briefly investigated three algorithms, which use the combined characteristics. These approaches are described in appendix A. At first, I tried the steepest descend approach. Sadly, this approach did not result in a useful solution. Secondly, the Richardson iterative approach was investigated, without success. At last the brute force search was investigated. Very soon it became clear that runtime would be unacceptable long.

- Template matching as described by Tsantalis
  A huge handmade decision tree have to be made, which is time consuming to construct.

- Sum Of Products Expressions
  After representing the design pattern and the system under consideration, it is unclear which steps have precisely to be made to determine the existence of the design pattern.

Implementable
Some methods are implementable, but need an extension to be useful.

- Blondel and Zager
  Although, calculating the similarity between nodes of a design pattern and the system under consideration is straightforward, an extension is needed to determine which nodes of the system under consideration are to be assigned to a design pattern. This problem is equivalent to the assignment problem in the field of Operation Research. The Hungarian method is the appropriate algorithm to solve the problem. Implementation of the algorithm is available on the internet.

  A simple example (see also figure 2.3 at page 8):

  $$\text{Similarity matrix } S = \begin{array}{c} \\ X \\ Y \end{array} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \\ \end{array}$$

  $$\text{Similarity matrix } \mathrm{S} = \begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline X & 1 & 0 & 0 & 0.00097 & 0 \\ Y & 0 & 1 & 1 & 0 & 0.00097 \end{array}$$

  Making a match by hand is quite simple: X is matched by 1 and Y is matched by 2 or 3. In more complicated cases the Hungarian method is needed to make the matches.

  Regrettably, the methods are only applicable to adjacency matrices. These matrices only contains ones and zeros. Therefore, only one type of relationships between classes can be denoted. An alternative is to make no differences between the several types of relationships. The occurrence of every type of relationship will therefore be denoted by a one.
  After applying the Hungarian method, it has to be checked whether every pair of corresponding relationships are of equal type. When not every pair of edges are of equal type then the Hungarian method has to deliver another solution, which maybe hard to implement.
  In table 3.1 these extensions are denoted as *Extended Blondel* and *Extended Zager*.

- Rule based
  Design patterns only have to be denoted by Prolog clauses. It encompasses the names of the classes and their relationships including their type.

- 4-tuples
  This method has just one problem: matching the relationships of the design pattern and the system under consideration. A backtracking search algorithm should solve this problem.

Some criteria are related. The ability to recognize many variants of design pattern may cause to lower *precision*, because some related classes are incorrectly recognized as a design pattern. The *recall* will be high, because only a very few design patterns will not be recognized.
The methods *Extended Blondel* and *Extended Zager* have the ability to recognize variants of design patterns, because the similarity between nodes does not have to be one. Nodes do not have to match completely. Therefore, these methods score highest for *recall* and just acceptable for *precision*.
The 4-tuple method will only match edges, if they fully match and not like *Extended Blondel* partially match. Variants of design patterns will be detected, as these do not diverge significantly from the template, or a variant is given as a template. Compared to *Extended*

Blondel, *recall* is lower, but *precision* is highest. It is easy to give feedback, because it is easy to detect, which edges of the template do not occur in the system under consideration. Therefore the score of *usable for feedback* is highest
The several low values for *Rule based* are motivated by the describing article [PK98].

In table 3.1 the values of the criteria are shown. The weight are low, high and highest. The

	implementable	*Number of recognizable patterns*	*Variants are detectable*	*Recall*	*Precision*	*Usable for feedback*	*Total*
Weight	3	1	2	3	3	2	
Extended Blondel	0	+	+	++	0	-	9
Extended Zager	0	+	+	++	0	-	9
Rule based	+	+	-	-	-	- -	-8
4-tuples	+	+	+	+	+	+	14

Table 3.1: Evaluation detection methods

meaning and the value of the tokens in table 3.1 are described in table 3.2

Token	Description	Value
- -	Not possible, very hard or very elaborate to realize.	-2
-	Hard or elaborate to realize. Modest	-1
0	Just acceptable.	0
+	Realizable, acceptable.	1
++	Easy realizable.	2

Table 3.2: Explanation of table 3.1

The *total* in table 3.1 is calculated by $\sum weight * value$.

Conclusion
Table 3.1 leads to the conclusion that 4-tuples is a reasonable choice for the remainder of this research.

4

Matching algorithm: Implementing and experimenting

This chapter starts with the basic algorithm of Ba-Brahem, which lacks a matching algorithm. Next, I formulate in pseudo code a matching algorithm. A prototype which contains the matching algorithm is used to experiment. Two series of experiments made clear that the prototype is able to identify 17 out of GoF 23 patterns within acceptable time. It is explained why four patterns cannot be identified. Issues for the remaining two design patterns are discussed. The chapter ends with the description of some practical improvements.

4.1. Plan

Experiments should make clear, whether the proposal of Ba-Brahem contributes to detecting design patterns. The essential part of his proposal comprises the code beneath (see also figure 2.4 and table 2.8 at page 13).

```
n = number of relationships which comprise the design pattern.

while n > 0 do
{
    Fill a table with relationships of the system under consideration,
    which match the relationships of the design pattern.
    /*
     *  How this table is filled, is unexplained.
     */

     if (the table contains rows with n filled columns) then
        print solutions
        exit
     else
        n--
}
```

There are two major issues:

- How to implement an algorithm which fills the table? To fill the table, matches between the relationships of the design pattern and the relationships of the system under consideration have to be made.
- Will the algorithm detect design patterns within reasonable time?

If the answers would be disappointing, another detection algorithm had to be chosen. Other issues, such as how read the templates and the transformation of the system under consideration to 4-tuples are seen as less important. These issues would not lead to failure of the experiments.
These considerations lead to the following steps:

1. Start with the greatest risk: implement a matching algorithm.
2. Test the implementation with the example of Ba-Brahem, figure 2.4, which contains inheritance, associations and dependency.
3. Test the implementation with the Bridge pattern with an extra relationship. This example contains an aggregate, inheritance and associations.
4. Iterative add GoF design patterns to the set of already detectable patterns.
   The goals of this last step are:
   - Determine detectable design patterns.
   - Determine the performance of the algorithm, when the system under consideration grows.
     During this step, all design patterns and the system under consideration are hard coded.
     During the last step, design patterns and the system under consideration will be read from file i.c. from the commandline and the tests will be repeated
   - Improve the algorithm when necessary.
5. Improvements for practical usage
   The templates of the design patterns and the system under consideration will be read from disk. Reading the system under consideration as a XMI-file is preferred. In future, results of educational exercises may be delivered as a XMI-file. This type of file can be produced by the UML modelling tool ArgoUML.

4.2. IMPLEMENTING A MATCHING ALGORITHM

The algorithm which matches the relationships of a design pattern with a part of the system under consideration, was not described by the proposal of Ba-Brahem. A natural approach is to match as many relationships of a design pattern as possible, until all are matched or a relationship of the design pattern cannot be matched. If a relationship of the design pattern cannot be matched, the forgoing match has to be undone and replaced by another match. Because of this backtracking nature, a recursive algorithm is appropriate.

The algorithm was first written in mainly natural language and Java. In several steps the

natural language were replaced by Java code. One these steps is given below.
This mix of Java and structured language clarifies the algorithm, which matches, if possible, the relationships of a design pattern with the relationships of the system under consideration.
The recursive method is part of a class, which represents the design pattern. The relationships of the system under consideration are represented by the parameter *fta*, which is an array of fourtuples [1]

The main line of the recursive method: Try to match a fourtuple of the design pattern with a fourtuple of the system under consideration. Starting by *startIndex* = 0 until *startIndex* equals the number fourtuples of the design pattern.

Ba-Brahem proposal enables detecting a design pattern partially. To realize this feature, the parameter *maxNotMatchableEdges* is used. If in the initial call *maxNotMatchableEdges* is set to zero then the complete design pattern has to be matched.

```
boolean findMatch(FourTupleArray fta, int startIndex,
                  MatchesNames matchedClassNames,
                  int maxNotMatchableEges)
{
     boolean found;
     // if a recursive call has found a complete match, its value is true.

     if (allFourtuplesAreMatched(startIndex))
     {
         if (isUniqSolution())
             showSolution();

         return true;
     }

     found = false;
     // No complete match is found yet.

     dpFt = getFourtuple(startIndex);
     // The current fourtuple of the design pattern,
     // which has to be matched.

     for every fourTuple ft of fta do
     // ft is a fourtuple of the system under consideration,
     // which may be matched already.
     {
         if (areMatchable(ft, dpFt, matchedClassNames))
         {
             updateMatchedClassNames(ft, dpFt, matchedClassNames);

             boolean tmp = findMatch(fta, startindex + 1,
```

[1] 4-tuple refers to the concept used by Ba-Brahem.
fourtuple refers to an object of the class FourTuple.

```
                                    matchedClassNames,
                                    maxNotMatchableEdges);
            // recursive call

            found = found || tmp;
            // These last two statements should not be combined to
            // found = found || findMatch(....);
            // Otherwise, if found == true, findMatch will not be called.

            undoMatch(ft, dpFt, matchedClassNames);
            // In final code this method does not exist.
            // Its intention is realized by using a local variable.
        }
    }

    if (!found)
    // The edge of the design pattern dpFt is not matched with
    // any edge of fta.
    {
        if (--maxNotMatchableEdges >= 0)
            return findMatch(fta, startIndex + 1, matchedClassNames,
                             maxNotMatchableEdges);

        return false;
    }
    else
       return true;
}
```

4.3. EXPERIMENTS

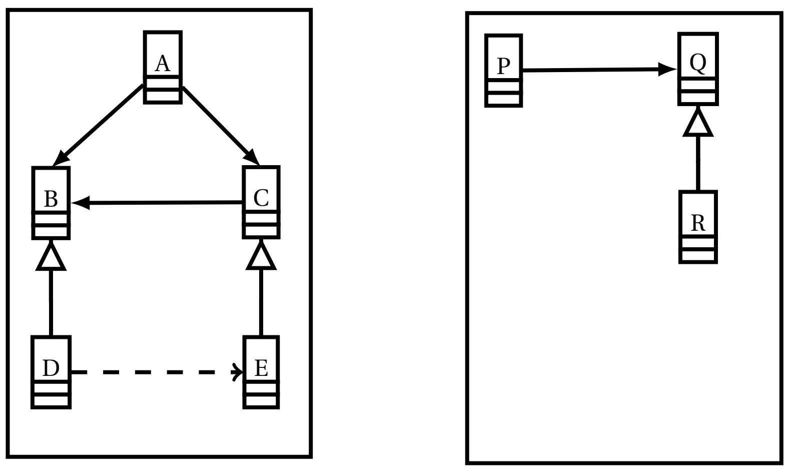
The final versions of the systems under consideration are contained in the separated delivered zip-file in XMI-format. The file containing all the detectable templates *templates.xml* is also located in the zip-file. To conduct an experiment call:

```
java -jar patterndetectionArgouml.jar -x xmifile
     -t template file -n maxNumberOfMissingEges
This call should be written on one line.

Example:
java -jar patterndetectionArgouml.jar -x Ba_Brahem.xmi
     -t Ba_Brahem.xml -n 1

Default values are input.xmi, templates.xml and 0
```

The software is improved during this stage, therefore, some imperfect results cannot be recalled.

Example of Ba-Brahem

To test the algorithm, the example of Ba-Brahem was added to the software. Ba-Brahem used as template the design pattern Prototype. Using this design pattern will be critized later on.



System under consideration

Prototype pattern

Figure 4.1: Example Ba-Brahem

The results are as expected: the template is three times detected.

	Match with the system under consideration		
	1	2	3
P	C	A	A
Q	B	B	C
R	D	D	E

Table 4.1: Permutation of prototype in the example of Ba-Brahem

The prototype pattern, used by the example of Ba-Brahem, contains two types of relationships:

- Directed association
- Inheritance

which were matched. This gives reasons to believe that all five types of relationships (directed association, inheritance, aggregate, composite and dependency) are detectable in general. This is confirmed in all experiments.

To test whether the algorithm can detect a design pattern, which occurs partially in the system under consideration, the Bridge pattern was used. See figure 4.2.

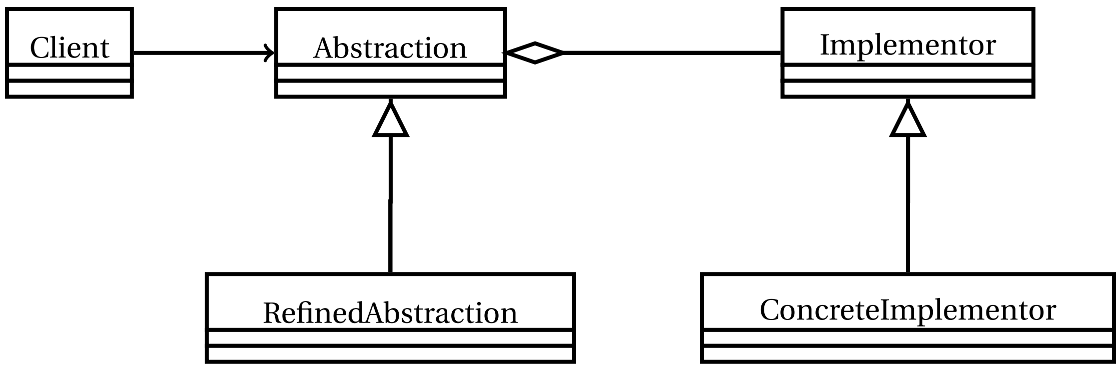


Figure 4.2: Bridge pattern

At first glance, it may be hard to see that the Bridge pattern which one relationship lacks, does occur in the example of Ba-Brahem (see figure 4.1). The mapping in which the aggregate is missing, is however:

Bridge pattern	System under consideration
Client	A
Abstraction	B
Implementor	C
RefinedAbstraction	D
ConcreteImplementor	E

Table 4.2: Match between the Bridge pattern with one missing relationship and the example of Ba-Brahem

The Bridge pattern is an example of a graph, which contains an isthmus, which is an edge whose removal split a connected graph in two connected graphs. Leaving out an isthmus may result in a singleton graph, a graph consisting of one node. Trying to detect a design pattern in a great UML class diagram, may result in many disconnected components of a design pattern. Even a small UML class diagram may contain many design patterns when one relationship is left out. When one relationship is left out, the example of Ba-Brahem contains several designs patterns as shown in table 4.3

Pattern	Frequency
Adapter	3
Bridge	1
Chain of Responsibility	3
Factory method	1
Iterator	1
Memento	1
Observer	2
Strategy	2

Table 4.3: Frequencies of design patterns in example of Ba-Brahem

This exemplifies that leaving out relationships is a feature, which is hardly useful for giving feedback.

Bridge pattern with an extra relationship

This test was conducted to test, whether the software could detect a design pattern, which contains a extra relationship.

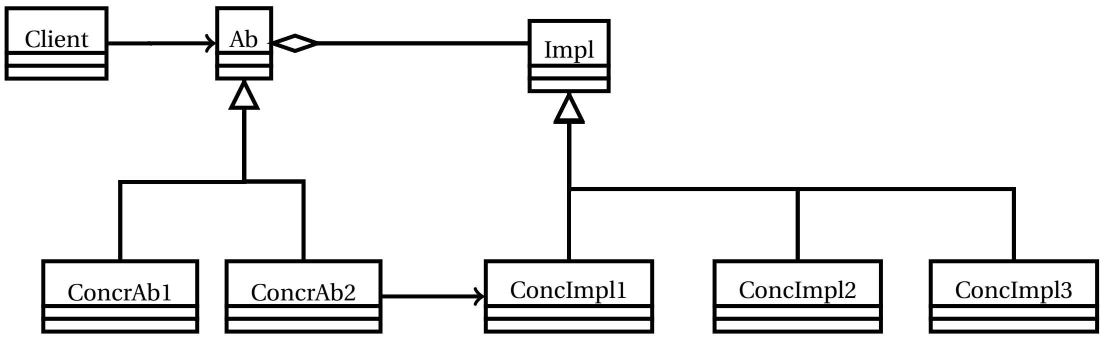


Figure 4.3: Bridge pattern

The Bridge pattern as defined in figure 4.2 was detected in figure 4.3. The extra relationship: *ConcrAb2 –> ConcImpl1* did not prevent detection, but the Bridge pattern was multiple times detected, because some classes like *concrAb1* and *ConcrAb2* can be interchanged. This problem would also occur in e.g. the Strategy pattern.

This problem was solved by adding a new type of relationship and adapting the matching algorithm. Beside *INHERITANCE* the relationship *INHERITANCE_MULTI* is introduced. As defined in the file *templates.xml*: if node1 inherits from node2 then

```
node2 may have more subclasses than the template defines,
and none of them may have relations to other classes/interfaces.
```

In example: *ConcrAb1* inherits from *Ab* and *ConcrAb2* also inherits from *Ab*. *ConcrAb1* and *ConcrAb2* have no relations to other classes/interfaces

The software was also adopted for notifying for extra relationships. This feature may be seen as giving feedback.

Test all Gang of Four patterns

These tests were conducted to determine:

- which design patterns are detectable;
- the performance of the algorithm.

One by one the templates of the design pattern were added to the code and the code of the system under consideration was expanded.

Adding templates to code made clear that some design patterns have essential implemental characteristics. For example, the Singleton pattern has a static attribute and for the Prototype the clone() method is essential. These implemental characteristics cannot be modelled by a 4-tuple.
Some patterns contain repeatable subgraphs e.g. Abstract Factory, Bridge and Mediator. See example 4.4. For the Bridge pattern this problem is solved as stated earlier (see page 24). For the Mediator this problem cannot be solved by only using 4-tuples. Tailor made matching algorithms for those design patterns have to be added to the code.

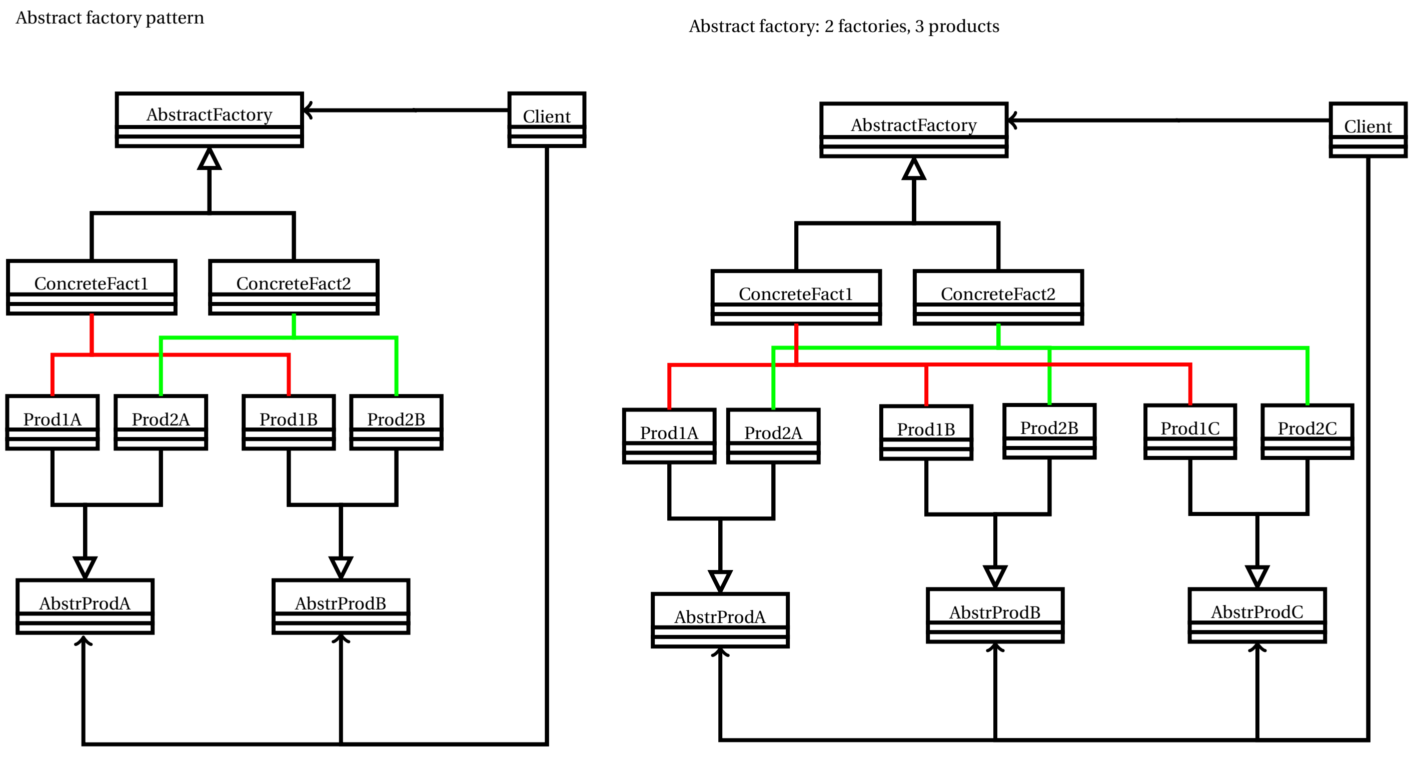


Figure 4.4: Abstract Factory template and an implementation

The result of adding templates to the code is denoted in the next table (table 4.4).

Name	**Detectable**	**Comment**	**Has repeat-able sub-graphs**
Creational Patterns			
Abstract Factory	J	The template has 2 factories and 2 products. Therefore the pattern was 2*2 times detected.	J
Builder	J		N
Factory Method	J	Is detected as part of the Abstract Factory	N
Prototype	N	The method clone() is essential	
Singleton	N	Essential are a static attribute and a static method, which returns the static attribute. Ba-Brahem et al. claim that one of the elements of the 4-tuple, the self reference, could be used for this purpose. A static method can be modelled by an UML class diagram. However, a class diagram cannot specify the statements within the static methods. This is specified by a sequence diagram, or it is an implementation issue. There is another reason: Self reference does not imply a static attribute. A married couple is an example of a self reference class, but I cannot marry myself.	
Adapter	J		N
Bridge	J		N
Composite	J		N
Decorator	J		N
Facade	N	A class structure A - B - C would be a Facade pattern in which B is the facade. In general, every set of classes, which contains one class which is connected to every other class, would be a Facade pattern. This cannot be modelled by 4-tuples.	
Flyweight	J		N
Proxy	J		N

Behavioral Patterns			
Chain of Responsibility	J		N
Command	J		N
Interpreter	J		N
Iterator	J		N
Mediator	J		J
Memento	J		N
Observer	J		N
State	N	The structure is identical to the structure of the Strategy pattern	
Strategy	J		N
Template Method	J	The structure is very simple: just one inheritance of an abstract class. This would result in many false positives. The pattern is therefore not added to the file *templates.xml*	N
Visitor	N	Implemental characteristic: The number of methods in the interface Visitor equals the number of classes which implements the interface element.	

Table 4.4: Gang of Four Patterns

The summarized results are:

Category	Frequency
Detectable, without reservation	14
Detectable, but has repeating parts	2
Identical detectable patterns	2
Unuseful detectable patterns	1
Not detectable	4

Table 4.5: Summarized results of the detection algorithm

One of the attributes of a 4-tuple indicates the presence of a self reference. This attribute is not necessary to detect any of the 17 detectable design patterns. Therefore, this attribute is redundant.

During this experiments two problems are solved. As shown in figure 4.3 some design patterns have repeatable parts. Figure 4.4 shows an implementation of the Abstract Factory. The template of the Abstract Factory may be six times matched, but only three matches are uniq. The resting three are permutations of already detected matches.

The final version of the software only shows uniq matches.

Performance
There are searching problems which are time-consuming, caused by their nature. A well known category of these problems is the set of NP-complete problems, such as the travelling salesman problem and the clique problem (for a given graph, find a subset of vertices, which constitutes a complete subgraph).

It is not easy to derive the time-complexity of the matching algorithm, because of its backtracking nature. The time-complexity is also determined by the number of possible matches. If the number of possible matches is high then recursive calls will be executed many times, which will result in a high order time-complexity. The number of possible matches is hard to guess. However, it is easy to derive the upper bound of the time complexity of a brute force algorithm. For a specific design pattern with k relationships and a system under consideration with n relationships, one has to find a specific permutation. The number of permutations is $P(n,k) = n!/(n-k)!$

While adding templates of design patterns to the code and letting grow the system under consideration, no increase of the runtime was noticeable.

Of this series of experiments, figure 4.5 shows the last class diagram of the system under consideration. This experiment searched for 13 design pattern, among which the Abstract Factory. This figure has 61 relationships and the Abstract Factory template has 13 relationships. In the worst case the brute force algorithm would $P(61,13) > 10^{22}$ times try to match the design pattern with the system under consideration. However, within one second all detected patterns were shown.

The code of this system under consideration is given in the method *void example_complex()* in the class *DetectPattern*, but some modifications are necessary to use the method *void example_complex()*. In the next section (page 30) a similar experiment is described.

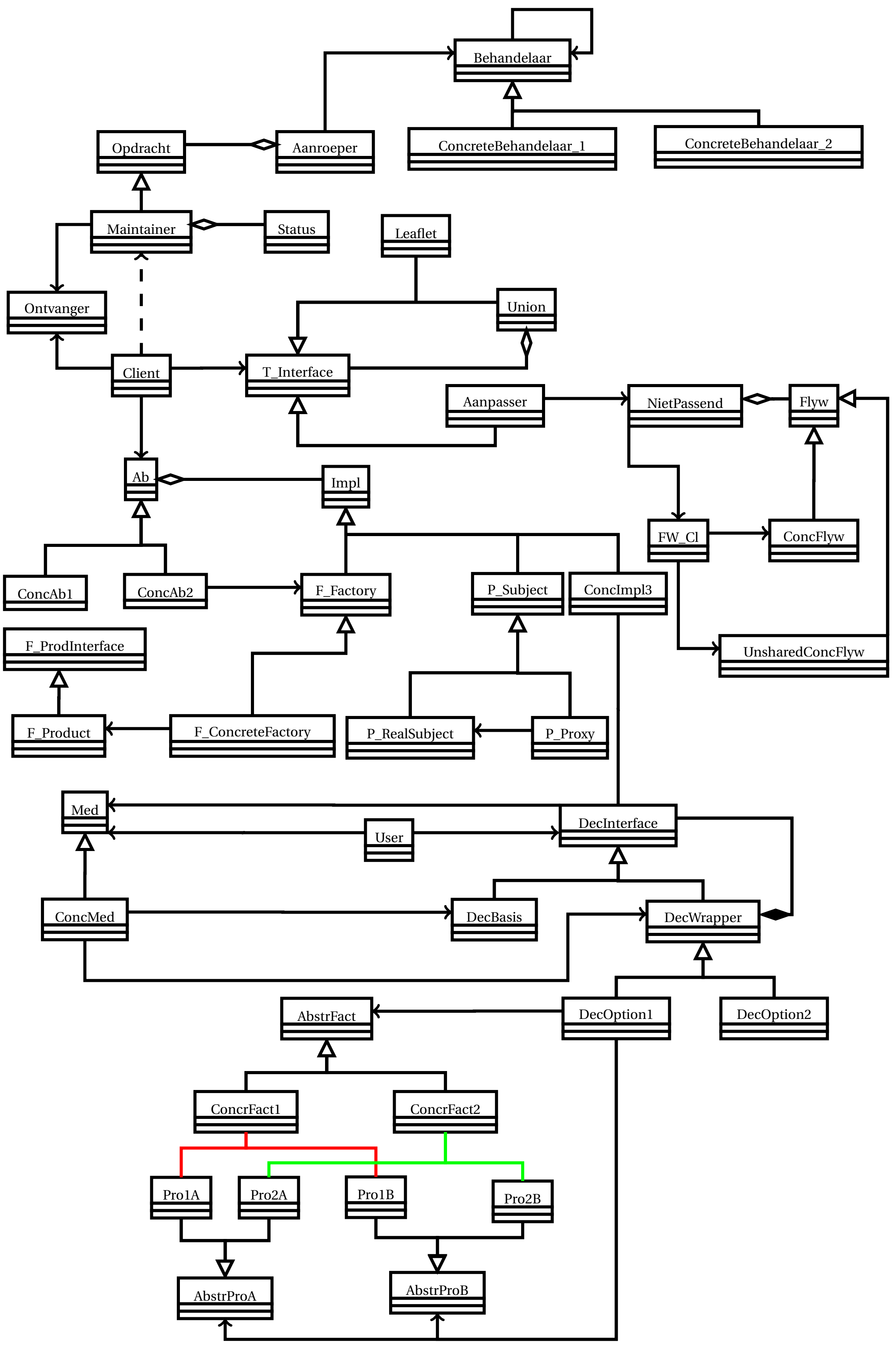


Figure 4.5: 13 design patterns

4.4. IMPROVEMENTS FOR PRACTICAL USAGE

To be of practical use, the software has to be able to read input files from the command line. One input file has to be the output file of a design tool.

- Reading input files
  The templates and the system under consideration were hard coded in the software. Templates are now defined in the default XML-file *templates.xml*, which has a straight forward structure. The structure is defined the template file.
  The template file is processed by a top down parser, which uses the standard available Java SAX-parser. The top down parser implements the design pattern Chain of Responsibility.

  The system under consideration can also be supplied at the command line.

- ArgoUML
  The Open University uses ArgoUML as UML modelling tool. This tool is able to generate XMI-file. XMI stands for XML Metadata Interchange. The format of XMI is maintained by the Object Management Group. XMI describes a XML-format, which can be used to exchange data of UML-diagrams between tools. Although XMI is a standard, a XMI-file generated by ArgoUML cannot be read by the UML modelling tool Visual Paradigm.

  To process XMI-files, the XML-structure has to be clear. The structure which is used by ArgoUML does not resemble the structure defined by the Object Management Group at all, nor is the structure described by documentation of ArgoUML. The Java source of ArgoUML is searched for useful classes, without success.
  I analysed the structure of XMI-files by studying generated XMI-files by ArgoUML of small designs. I focused on only those tags, which are necessary to create fourtuples. Like the template file, the XMI-file is processed by a top down parser, which uses the standard available Java SAX-parser.

- Command line version
  Default the software uses *templates.xml* and *input.xmi* as input files. A tiny tutorial, see appendix B gives detailed information about using the software.

Speed again
After improving the code a second experiment has been conducted to test the speed of the algorithm and its ability to detect design patterns. With ArgoUML a design (see figure 4.4) was made, which contains 33 classes, 49 relationships and 17 partially overlapping design patterns. The generated XMI-file was processed within 0.8 seconds on average. The summarized results are:

Pattern	Frequency
AbstractFactory	1
Adapter	7
Bridge	1
Builder	1
Chain of Responsibility	1
Command	1
Composite	2
Decorator	1
Factory Method	5
Flyweight	1
Interpreter	2
Iterator	1
Mediator	1
Memento	1
Observer	1
Proxy	1
State - Strategy	4

Table 4.6: Frequencies of detected design patterns in figure 4.6

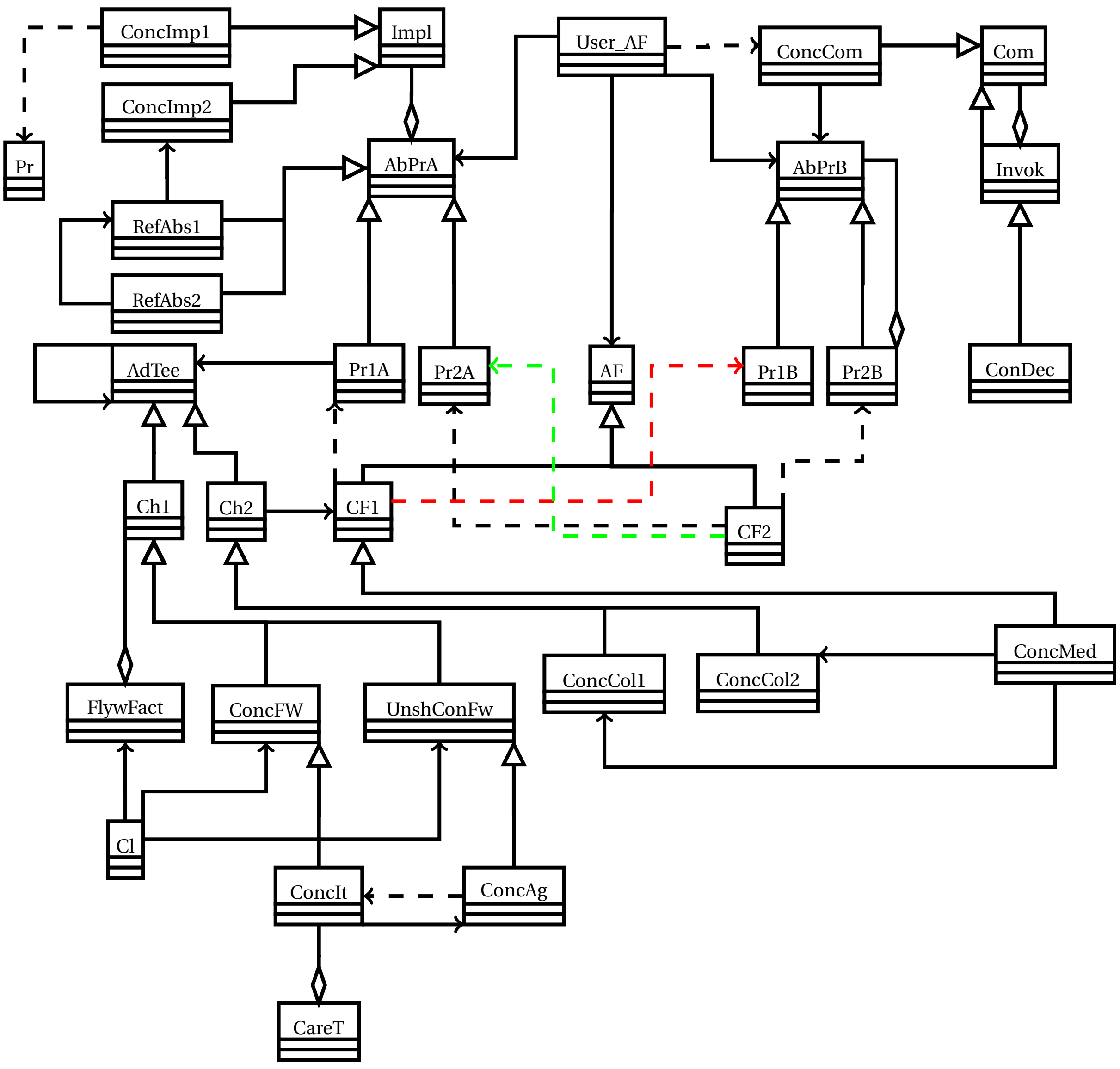


Figure 4.6: Class diagram used for pattern detection and performance

In the next chapter the results of these tests are compared to tests described in literature.

5 COMPARING TO RELATED WORK

In this chapter the test results, shown in chapter 4, are compared to results of experiments conducted by Bergenti, Dong, Tsantalis and Prechelt, which are described in chapter 2.
Other authors write about their approaches and experiments without showing any result i.e. Gautam [GD12] and Tsantalis [Tsa+05]. Of course their articles are not included in comparison.
As stated in chapter 2, the problem of finding a design pattern is equivalent to *subgraph isomorphism problem*. Wikipedia states, based on work of Stephan Cook: *Subgraph isomorphism is a generalization of both the maximum clique problem and the problem of testing whether a graph contains a Hamiltonian cycle, and is therefore NP-complete.* [Coo71] [1] The above cited authors do not mention this relationship, nor do they describe their matching algorithm in detail.

5.1. AGGREGATE COMPARISON

The comparison between the related work of Bergenti, Dong, Tsantalis, Prechelt and my software is summarized in table 5.1 at page 36. Their work is not fully described, so for some criteria an educated guess is made.

Template format
The design patterns, which are searched in the system under consideration, may be given as input in several ways. I use an XML-file, therefore the input file and the program source are separated. The source does not need do be recompiled for every change of the input. Only Bergenti describes his method of input. He uses Prolog clauses, which have to be recompiled.

Number of recognized patterns
My software is able to detect 17 GoF design patterns. As described in chapter 2, a greater number is not possible. All authors tested smaller numbers, but it is very likely that a higher number can be detected, because other design patterns may be defined as well by their characteristics.

[1] https://en.wikipedia.org/wiki/Subgraph_isomorphism_problem#cite_note-1

Recall: does detect variants
The definitions of recall and precision are given at page 14.
Only Bergenti did notice the existence of variants. His software is able to detect permutations of design patterns. Detecting permutations is not straight forward and therefore, for other authors this criterion is denoted as *probably not*.
My software is able to detect design patterns, which contains inheritance with variable number of subclasses, which do not have any other relationships. For instance a Bridge pattern with variable number of abstractions will be detected. However an Abstract factory with three products and two factories will be detected twice as Abstract Factory with two products an two factories.
The number of false negatives, which is necessary to determine the recall, is hard to determine. For instance, in some cases an abstract class may be replaced by an interface. For those cases an expert is needed to determine the number of false negatives. Therefore, I cannot give an estimate of the recall of my software.

Precision
My software only detects a design pattern, if it structure fully corresponds a defined template. So, the number of false positives is zero and therefore the precision is 100%.
Dong uses open source software, which is not documented. He manually checked the results, which depends on interpretation.

Does detect permutations
My software does detect permutations of a design pattern. For instance, a Abstract Factory with two products and two factories will be detected once. Two products and two factories would lead to 4 permutations. Other authors do not discuss this criterion.

Acceptable speed
My experiments are conducted on a PC with the characteristics: AMD Athlon (tm) 64 x 2 Dual Core Processor 5000+, 2.6 GHz and 4 Gb RAM. The 33 classes in figure 4.6 were processed in 0.8 seconds on the average. This is slower than the software of Dong, but my software is able to detect 17 design patterns in one test.
Although some time is spend to reveal the relation between the size of the system under consideration and time to process the data, no clear relation could be established.
The conversion of C++ headers to Prolog clauses by Bergenti took up to two hours.

Uses XMI input
Only the XMI output file of ArgoUML can be parsed.

Easy to define a template
None of the authors give detailed information about the format of their template. A template may be read from a file or be incorporated in code, therefore I denote this criterion as *unknown*.

Gives feedback
Only the Prolog implementation of Bergenti gives detailed feedback. When a design pattern is detected in the system under consideration, the classes may have more associations

than the design pattern. This is shown in figure 4.3 at page 24. My software shows extra relationships. It also is able to detect patterns if some relationships are missing, but as explained in section 4.3, this is not very useful in practice.

Uses a user friendly tool
I made a command line and not a GUI based implementation. In a Windows environment this is not seen as user friendly.

Final remark
The Open University considers to add graphical user interfaces to my software, so that it can be used in an educational environment.

Criteria	Template matching			Rule based	
Author	E.M. van Doorn	J.Dong et al.	N. Tsantalis et al.	L. Prechelt et al.	F. Bergenti et al.
Based on	UML class diagram	Java sources	Java sources	C++ sources	UML class and collaboration diagrams
Template format	XML	XML	unknown	unknown	Prolog clauses
Number of recognized patterns	17 without implemental issues	4	10	5	10
Recall: does detect variants	yes	yes	100% with a few exceptions	100%	unknown
Precision	100%	avoid false positives	100%	14-51%	unknown
Does detect permutations	yes	probably not	probably not	probably not	no, explicitly stated
Acceptable performance	yes	probably	probably	no	probably
Uses XMI input	yes	yes	no	no	implicit by using ArgoUML
Easy to define a template	yes	unknown	unknown	unknown	unknown
Gives feedback	a little	no	no	no	yes
Uses a user friendly tool	no	yes	unknown	unknown	yes, ArgoUML

Table 5.1: Software comparison

6
CONCLUSIONS AND FUTURE WORK

This chapter describes the conclusions of this research, some new insights and proposals for future work.

6.1. CONCLUSIONS

First, the subquestions formulated in section 1.1 and second, the research question are answered.

1. *What is a design pattern?*
   The formal answer is: 'Design patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.' [Gam+95]

2. *What are the characteristics of a design pattern in an UML class diagram?*
   I found the following characteristics: class, abstract class, interface, association, generalization, aggregation, composite, dependency.

3. *Which methods to detect design patterns are described in literature?*
   The following methods are found:

   - Use matrices to represent class diagrams and also design patterns.
     To determine whether a matrix contains a submatrix, one can use cross-validation. However, this approach is only useful if the order of rows and columns of the matrix which represents the design pattern corresponds to the bigger matrix, which represents the system under consideration. Otherwise, every permutation of the order of rows and columns has to be compared, which is time consuming. To prevent a brute force search, preprocessing the matrix of the system under consideration is necessary, as discussed at page 6.
     To determine the similarity between a submatrix and a bigger matrix, one can use two algorithms: the algorithm of Blondel and the algorithm of Zager.
   - Decision trees
     One can represent the decisions that control the search for a design pattern in the form of a decision tree.

    - 4-tuples
      Relationships and classes can be represented by 4-tuples.
    - Prolog is used to represent classes and relationships. The classes and interfaces are represented as facts and the several types of relationships are represented by rules.
    - Sum of products are used to represent classes and relationships.
      Relationships including classes and interfaces are described as strings, which contains boolean expressions.
      For instance: SOP(directed association) = A.B.C + A.D. This means A –> B –> C and A –> D.

4. *Based on the (partial) existence of a design pattern in a UML class diagram, can feedback be supplied, and if so, how?*
   To answer this question, I have limited myself to the method of Ba-Brahem. His method shows how missing 4-tuples can be detected. However, many disconnected parts of the system under consideration may be connected, whereby they constitute an unintentionally and meaningless design pattern.
   My research demonstrates a way to show the 4-tuples not belonging to a detected pattern. They designer may use this information in case he has made a mistake.

The **research question**: *Are design patterns in a UML class diagram automatically detectable, and can one automatically supply some feedback?* can be answered now. In practice it is shown that design patterns can automatically be detected by representing the system under consideration and the design pattern as matrices, decision trees and Prolog clauses when also dedicated algorithms are used. In practice only the Prolog implementation is able to give feedback. I have shown that 4-tuple method can used to automatically detect 17 GoF design patterns. However, my software only gives limited feedback.

6.2. NEW RESULTS

My research provides some new results:

- The concept of 4-tuples to represent design patterns is implementable.
- Ba-Brahem's idea to detect design pattern that are only partially present, is not useful.
- 4-tuples can be replaced by 3-tuple, because the self reference attribute is not useful.
- Only the characteristics: class, (directed) association, aggregate, composite, inheritance and dependency are necessary to detect 17 GoF design patterns. However, the answer to subquestion 2 also contains abstract class and interface.
- Only very simple feedback can be generated.

The limitations of my method are the following:

- Not all GoF design patterns can be detected.
  See table 4.4

- Recall and precision are not measured. For these measurements, one needs examples and opinions of experts.
- Only very simple feedback can be generated.

6.3. FUTURE WORK

This section contains proposals for future work, starting with scientific topics and ending with practical proposals.

- Insight in effective learning
  A thorough investigation is needed to determine, which feedback is appropriate and feasible.
- Insight in modelling
  - Three design patterns: Prototype, Singleton and Visitor have characteristics, which cannot be modelled by a 4-tuple.
    The Prototype pattern contains a class, which calls the clone-method of another class. The existences of methods is not modelled by 4-tuples.
    The Singleton design pattern contains an *object*, which needs to have a self reference. A 4-tuple contains identifiers of classes and/or interfaces, but no identifiers of objects. Therefore a 4-tuple cannot be used to model a Singleton pattern.
    The Visitor pattern contains for every class it visits a corresponding class. This correspondence cannot be modelled by 4-tuples.
    It may be possible to extend the 4-tuple, so that objects and method calls can be modelled. To model the Visitor pattern, the correspondences have to be modelled.
    It may be possible to detect a Facade pattern by marking the Facade class by means of an extra attribute. It will also be necessary to expand the matching algorithm.
  - To detect the Abstract Factory with more than two factories and/or products, the matching algorithm should be improved.
- Insight in the algorithm
  Determine the relation between the runtime and the size of the system under consideration and the design pattern.
- What is needed to make the software applicable?
  For instance:
  - Make the software suitable for various versions of XMI.
  - Make the software GUI-based.

A

APPROACHES TO FIND SUBMATRICES

Problem introduction
Jing Dong describes in subsection 3 an approach for detecting a design pattern in an UML class diagram. The design pattern and the class diagram are represented by two matrices A and B. Whether the class diagram contains the design pattern is reduced to the question: *Which rows and columns of B have to be deleted and how to permute the locations of the remaining rows and columns of B such that the final result is matrix A?*

I tried, without success, three approaches:

- Steepest descend
- Richardson Iteration
- Brute force

These approaches are demonstrated with the next example.

Let A = $\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$ and B = $\begin{pmatrix} 5 & 6 & 7 \\ 8 & 4 & 3 \\ 9 & 2 & 1 \end{pmatrix}$

To answer the question, matrices R and C are used to permute rows and columns of B and to project B such that $\|RBC - A\|_2^2$ is minimal, preferably zero, in which case $RBC = A$. Because the matrices R and C permute rows or columns and project, they may only consist of zeroes and ones and every row and column may contain atmost a one.
If a class is not relevant for a detecting a design pattern then a row and the corresponding column of matrix B have to be deleted. Even so, if two rows are interchanged then the corresponding columns of matrix B have to be interchanged. Therefore C = R^T, the transposed matrix of R.

The answer is:

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 5 & 6 & 7 \\ 8 & 4 & 3 \\ 9 & 2 & 1 \end{pmatrix} \begin{pmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \end{pmatrix}$$

Steepest descend
Problem formulated in general terms:

If A is a (n x n) and B a (m x m) matrix, find matrix R which minimizes

$$Q(R) = \left\| RBR^T - A \right\|_2^2 = \sum_{i=1}^{n} \sum_{j=1}^{n} ((RBR^T)_{ij} - A_{ij})^2 \tag{A.1}$$

The idea behind the steepest descend method is quite simple. If Q is not equal to zero then R has to be changed. In what direction has R to be changed? It can be proven that the gradient of Q: ∇Q is perpendicular to the error surface, in this case Q(R). Therefore, if the elements of R are changed in the direction of ∇Q then this change is most effective.

To implement the steepest descend method, a vector **x** is needed, which contains all r_{ij}: $\mathbf{x} = (r_{11}, r_{12}, \cdots, r_{mn})$. To find the minimum the next formula is used.

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \gamma \nabla Q \tag{A.2}$$

$\mathbf{x_0}$ may be arbitrary chosen, i.e. **1** and $0 < \gamma \ll 1$.

Adding the vector **m**, will prevent the iteration process to end in a local minimum. This will also result in a higher convergence speed.

$$\begin{aligned} \mathbf{x}_{n+1} &= \mathbf{x}_n - \gamma \nabla Q - \alpha \mathbf{m}_n \\ m_{n+1} &= \gamma \nabla Q + \alpha \mathbf{m}_n \\ \mathbf{m}_0 &= \mathbf{0} \end{aligned} \tag{A.3}$$

Step by step the gradient will be derived:

$$\nabla Q = (\partial Q / \partial r_{11}, \ldots \partial Q / \partial r_{nm}) \tag{A.4}$$

Define $G = RBR^T$ hence

$$G_{ij} = \sum_{k=1}^{m} \sum_{l=1}^{m} r_{ik} b_{kl} r_{jl} \quad (\text{note } r^t_{lj} = r_{jl}) \tag{A.5}$$

Using A.1 and A.5 results

$$\partial Q / \partial r_{vw} = \sum_{i=1}^{n} \sum_{j=1}^{n} \partial Q_{ij} / \partial r_{vw} = 2 \left\| RBR^T - A \right\|_2 \sum_{i=1}^{n} \sum_{j=1}^{n} \partial G_{ij} / \partial r_{vw} \tag{A.6}$$

For calculating $\partial G_{ij}/\partial r_{vw}$ there are four distinct cases:

$$\partial G_{ij}/\partial r_{vw} = 0 \qquad (v \neq i \wedge v \neq j) \tag{A.7a}$$

$$\partial G_{ij}/\partial r_{iw} = \sum_{k=1}^{m} b_{wk} r_{jk} = (BR^T)_{wj} \qquad (i \neq j) \tag{A.7b}$$

$$\partial G_{ij}/\partial r_{jw} = \sum_{k=1}^{m} r_{ik} b_{kw} = (RB)_{iw} \qquad (i \neq j) \tag{A.7c}$$

$$\partial G_{ii}/\partial r_{iw} = (BR^T)_{wi} + (RB)_{iw} \tag{A.7d}$$

Substitute formulas A.7 into A.6 and A.4 to calculate A.3.

Richardson Iteration [Kel95]
This approach is based on the following idea:
Let D be a square matrix an **b** a vector. Find a vector **x** such that $D\mathbf{x} = \mathbf{b}$.
It is easy to see: $\mathbf{x} = (I - D)\mathbf{x} + \mathbf{b}$.
The *Richardson iteration* is defined as $\mathbf{x_{k+1}} = (I - D)\mathbf{x_k} + \mathbf{b}$.
It can be proven that, if $\|I - D\| < 1$ then $\{\mathbf{x_k}\}$converges.

Applied to the problem: find R such that $RBR^T = A$.
Is easy to see that $RB(I - R^T) + A = RB$.
But, rows and columns have to be added to $R, R^T and A$ to make this equation valid.
If B is nonsingular then: $R_{k+1} = (R_k B(I - R_k^T) + A)B^{-1}$

The results of this method are not useful.

Brute force
The question is how to permute the rows and columns of B such that $\|RBR^T - A\|_2^2$ is minimal, preferably zero, in which case $RBR^T = A$.

In practice this will result in long runtimes.
If A is a (n x n) and B a (m x m) matrix and n <= m, then the number of permutations is $m * (m-1) * (m-2) * (m-3) * \cdots (m-n+1)$.

if m = 20 and n = 4 the number of permutations is greater than 10^5.

B

TUTORIAL FOR DETECTION SOFTWARE

This a tiny tutorial for using the detection software.

The application needs two input files:

1. An XMI-file generated by ArgoUML, which contains the system under consideration.

2. An XML-file which contains a design patterns.

The overall structure of the XML is informally defined by:

```
<?xml version="1.0" encoding="UTF-8"?>
<templates>
   <template name="name">
       <edge node1="fromA"  node2="toB"  type="typeRelationship"/>
       <edge                                                    />
       .......
   </template>

   <template ....... >
       ......
   </template>
   .......
   .......
</templates>
```

The design pattern is defined by 4-tuples. A 4-tuple is used to define a relationship, an edge, between two nodes. A node is class, an abstract class, or an interface. To define a 4-tuple, only 3 attributes are necessary! The fourth attribute get its value automatically by the application.

typeRelationship	meaning
ASSOCIATION ASSOCIATION_DIRECTED	bidirectional association. unidirectional association from node1 to node 2.
AGGREGATE	node2 is the aggregate.
COMPOSITE	node2 is the composite.
INHERITANCE	node2 is the superclass or (super)interface. node1 is a subclass or subinterface which may have relations to other classes/interfaces.
INHERITANCE_MULTI	node2 is the superclass or (super)interface. node1 is the subclass. node2 may have more subclasses than the template defines, but none of them may have relations to other classes/interfaces.
DEPENDENCY	node1 depends on node2.

Table B.1: Possible values of typeRelationship

example

```
<templates>
   <template name ="Memento">
      <edge node1="Memento"     node2="Caretaker"   type="AGGREGATE"/>
      <edge node1="Originator"  node2="Memento"     type="DEPENDENCY"/>
   </template>

   <template name ="Proxy">
      <edge node1="Client"      node2="Subject"
                                           type="ASSOCIATION_DIRECTED"/>
      <edge node1="Proxy"       node2="Subject"     type="INHERITANCE"/>
      <edge node1="RealSubject" node2="Subject"     type="INHERITANCE"/>
      <edge node1="Proxy"       node2="RealSubject"
                                           type="ASSOCIATION_DIRECTED"/>

    </template>
</templates>
```

command to start the application

```
java -jar patterndetectionArgouml.jar -x xmifile
     -t templatefile -n maxNumberOfMissingEges
This call should be written on one line.

Example:
java -jar patterndetectionArgouml.jar -x Ba_Brahem.xmi
     -t Ba_Brahem.xml -n 1

Default values are input.xmi, templates.xml and 0
```

C
DELIVERABLES

All files are made under Linux. To use under Windows, conversion of the character \n to the characters \r\n, also known als CR/LF conversion, may be necessary to make text files easy readable. The command *zip -l zipfile* will do this job. However, do not apply *zip -l* to binary files.

Binary files:	*.zargo and jar-file
Text files:	*.java, *.xmi and *.xml files

For convenience one zip-file is delivered: *deliverables.zip*. This files contains: *patterndetectionargouml.jar*, *quickstart.zip*, *netbeansproject.zip*, *xmi.zip* and *zargo.zip*

C.1. QUICKSTART.ZIP

These files can be used for a first and quick start.
Start by: *java -jar patterndetectionargouml.jar -t Ba_Brahem.xml* and the result for figure 4.1 at page 22 will be shown.

Figure 4.1 contains a Factory method with one extra edge.
This will be shown by: *java -jar patterndetectionargouml.jar*

Figure 4.1 contains 13 design patterns with at most one association missing.
This will be shown by: *java -jar patterndetectionargouml.jar n 1*

Ba_Brahem.xml
input.xmi
template.xml

Table C.1: Content of quickstart.zip

The file template.xml contains all 17 detectable GoF design patterns.

C.2. NETBEANSPROJECT.ZIP

netbeansproject.zip contains a map/directory to be used with Netbeans 8.0.2. No binairy files are included. The zip-files contains several packages:

Package: patterndetectionargouml
Purpose: main program
PatterndetectionArgouml.java

Package: argoxmi
Purpose: parsing XMI-files, which are generated by ArgoUML
AbstractionElement.java
ArgoXMI.java
AssociationElement.java
ClassElement.java
Constants.java
GeneralizationElement.java
SAXHandler.java
VerwerkSAXTags.java

Package: fourTuples
Purpose: matching between design pattern(s) and the system under consideration.
DetectPatterns.java
FourTupleArray.java
FourTuple.java
FT_constants.java
MatchedNames.java
Solution.java
Solutions.java
TagValue.java

Package: fourTuple.template
Purpose: parsing XML template file
EdgeElement.java
SAXHandler.java
TemplateElement.java
Templates.java
VerwerkSAXTags.java

C.3. XMI.ZIP AND ZARGO.ZIP

xmi.zip and zargo.zip contain corresponding files. A zargo-file can be used to produce the corresponding XMI-file. For convenience are the XMI-files added.

zargo.zip	**xmi.zip**
AbstractFactory.zargo	AbstractFactory.xmi
Ba_Brahem.zargo	Ba_Brahem.xmi
Bridge.zargo	Bridge.xmi
Builder.zargo	Builder.xmi
ChainOfResponsibility.zargo	ChainOfResponsibility.xmi
Command.zargo	Command.xmi
Composite.zargo	Composite.xmi
Decorator.zargo	Decorator.xmi
Flyweight.zargo	Flyweight.xmi
Interpreter.zargo	Interpreter.xmi
Iterator.zargo	Iterator.xmi
Mediator.zargo	Mediator.xmi
Memento.zargo	Memento.xmi
Observer.zargo	Observer.xmi
Proxy.zargo	Proxy.xmi
runtimeComplexity.zargo	runtimeComplexity.xmi
Strategy.zargo	Strategy.xmi

runtimeComplexity is used to measure the runtime of a complex example. See figure 4.5 at page 29.

D

Legend UML class diagram

This appendix gives the legend of UML symbols used in the class diagrams in this thesis.

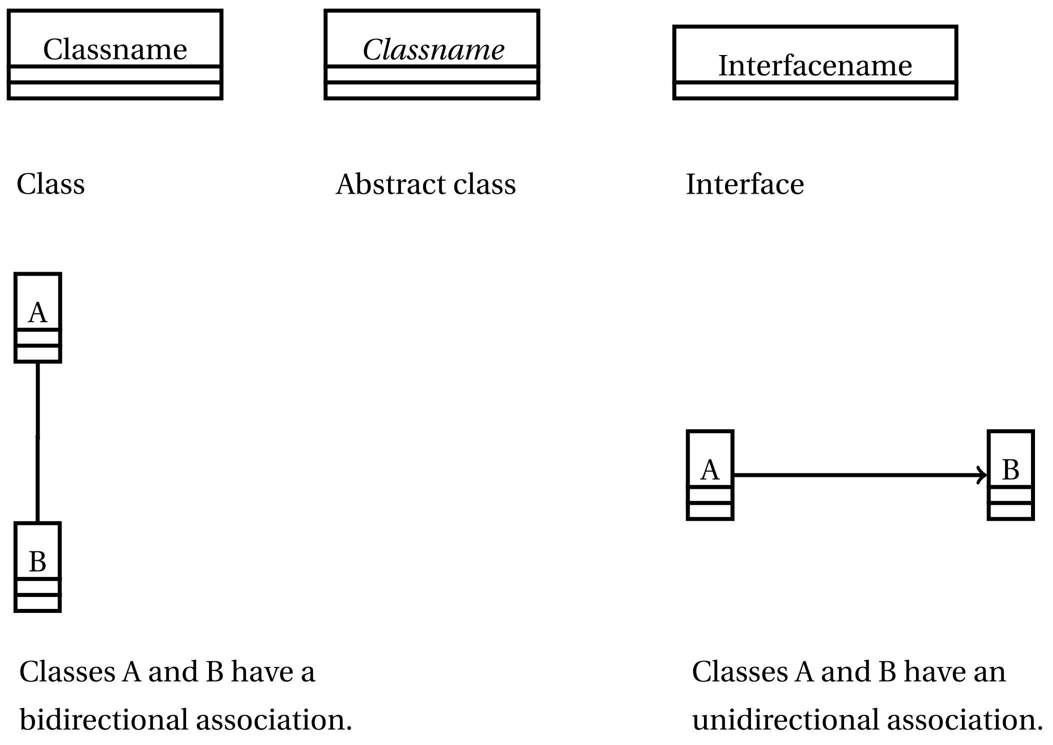
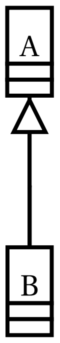
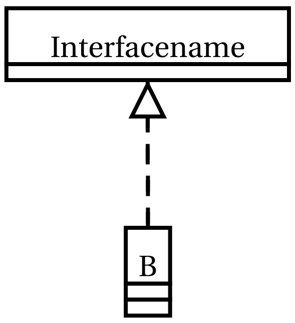

Classname
Classname
Interfacename
Class
Abstract class
Interface
A
B
A
B
Classes A and B have a bidirectional association.
Classes A and B have an unidirectional association.

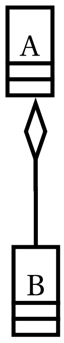
Figure D.1: Legend of UML symbols in class diagrams



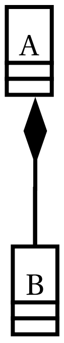
Class B inherits from class A.



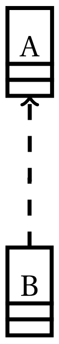
Class B realizes Interfacename.



Classes A and B have a whole/part relation i.c. a aggregation.



Classes A and B have a whole/part relation i.c. a composition.



Class B depends on class A.

Figure D.2: Legend of UML symbols in class diagrams

Bibliography

Books

[Gam+95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2.

[Kel95] C. T. Kelley. *Iterative methods for linear and nonlinear equations*. Frontiers in applied mathematics. Philadelphia: Society for Industrial and Applied Mathematics, 1995. ISBN: 0-89871-352-8.

[Tah92] H.A. Taha. *Operations research: an introduction*. v. 1. Macmillan, 1992. ISBN: 9780024189752.

In Books

[Crn10] Gordana Dodig Crnkovic. "Model-Based Reasoning in Science and Technology: Abduction, Logic, and Computational Discovery". In: ed. by Lorenzo Magnani, Walter Carnielli, and Claudio Pizzi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. Chap. Constructive Research and Info-computational Knowledge Generation, pp. 359–380. ISBN: 978-3-642-15223-8. DOI: 10.1007/978-3-642-15223-8_20. URL: http://dx.doi.org/10.1007/978-3-642-15223-8_20.

Academic Articles

[BQ14] Afnan Salem Ba-Brahem and M. Rizwan Jameel Qureshi. "The proposal of improved inexact isomorphic graph algorithm to detect design patterns". In: *CoRR* abs/1408.6147 (2014). URL: http://arxiv.org/pdf/1408.6147v1.pdf.

[Cop97] J. Coplien. "Idioms and patterns as architectural literature". In: *Software, IEEE* 14.1 (Jan. 1997), pp. 36–42. ISSN: 0740-7459. DOI: 10.1109/52.566426.

[GD12] Amit Kumar Gautam and Saurabh Diwaker. "Automatic Detection of Software Design Patterns from Reverse Engineering". In: *IJCA Special Issue on Issues and Challenges in Networking, Intelligence and Computing Technologies* ICNICT.1 (Nov. 2012), pp. 17–12. URL: http://research.ijcaonline.org/icnict/number1/icnict1013.pdf.

[GPT11] Manjari Gupta, Akshara Pande, and A. K. Tripathi. "Design Patterns Detection Using SOP Expressions for Graphs". In: *SIGSOFT Softw. Eng. Notes* 36.1 (Jan. 2011), pp. 1–5. ISSN: 0163-5948. DOI: 10.1145/1921532.1921541. URL: http://doi.acm.org/10.1145/1921532.1921541.

[GR14] Manjari Gupta and Rajwant Singh Rao. "Article: Design Pattern Mining by Product of Sum (POS) Expression for Graphs". In: *International Journal of Computer Applications* 85.7 (Jan. 2014), pp. 38–42.

[PK98] Lutz Prechelt and Christian Kramer. “Functionality versus Practicality: Employing Existing Tools for Recovering Structural Design Patterns”. In: *j-jucs* 4.12 (Dec. 1998), pp. 866–882. URL: http://www.jucs.org/jucs_4_12/functionality_versus_practicality_employing.

[Tsa+06] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S.T. Halkidis. “Design Pattern Detection Using Similarity Scoring”. In: *Software Engineering, IEEE Transactions on* 32.11 (Nov. 2006), pp. 896–909. ISSN: 0098-5589. DOI: 10.1109/TSE.2006.112.

[YWG04] Tong Yi, Fangjun Wu, and Chengzhi Gan. “A Comparison of Metrics for UML Class Diagrams”. In: *SIGSOFT Softw. Eng. Notes* 29.5 (Sept. 2004), pp. 1–6. ISSN: 0163-5948. DOI: 10.1145/1022494.1022523. URL: http://doi.acm.org/10.1145/1022494.1022523.

[ZV08] Laura A. Zager and George C. Verghese. “Graph similarity scoring and matching”. In: *Applied Mathematics Letters* 21.1 (2008), pp. 86 –94. ISSN: 0893-9659. DOI: http://dx.doi.org/10.1016/j.aml.2007.01.006. URL: http://www.sciencedirect.com/science/article/pii/S0893965907001012.

IN PROCEEDINGS

[BP00] F. Bergenti and A. Poggi. “IDEA: A design assistant based on automatic design pattern detection”. In: *Proceedings of 12th International Conference on Software Engineering and Knowledge Engineering (SEKE 2000)*. PHD in 2001. 2000, pp. 336–343. URL: https://www.sics.se/~nilsf/SEKE.pdf.

[Coo71] Stephen A. Cook. “The complexity of theorem-proving procedures”. In: *In STOC*. ACM, 1971, pp. 151–158.

[DSZ08] Jing Dong, Yongtao Sun, and Yajing Zhao. “Design Pattern Detection by Template Matching”. In: *Proceedings of the 2008 ACM Symposium on Applied Computing*. SAC ’08. New York, NY, USA: ACM, 2008, pp. 765–769. ISBN: 978-1-59593-753-7. DOI: 10.1145/1363686.1363864. URL: http://doi.acm.org/10.1145/1363686.1363864.

[Nie+02] Jörg Niere, Wilhelm Schäfer, Jörg P. Wadsack, Lothar Wendehals, and Jim Welsh. “Towards Pattern-based Design Recovery”. In: *Proceedings of the 24th International Conference on Software Engineering*. ICSE ’02. New York, NY, USA: ACM, 2002, pp. 338–348. ISBN: 1-58113-472-X. DOI: 10.1145/581339.581382. URL: http://doi.acm.org/10.1145/581339.581382.

[RG13] Rajwant Singh Rao and Manjari Gupta. “Design Pattern Detection by Greedy Algorithm Using Inexact Graph Matching”. In: *International Journal of Engineering Research and Technology*. Vol. 2. 10 (October-2013). ESRSA Publications. 2013. URL: http://www.ijert.org/view-pdf/6029/design-pattern-detection-by-greedy-algorithm-using-inexact-graph-matching.

[Wen03] Lothar Wendehals. “Improving Design Pattern Instance Recognition by Dynamic Analysis”. In: *Proc. of the ICSE 2003 Workshop on Dynamic Analysis (WODA), Portland, USA*. May 2003.

TECHNICAL DOCUMENTATION

[Bro96] Kyle Brown. *Design Reverse-engineering and Automated Design-pattern Detection in Smalltalk*. Tech. rep. Raleigh, NC, USA, 1996.

[Tsa+05] Nikolaos Tsantalis, Alexander Chatzigeorgiou, Spyros T. Halkidis, and George Stephanides. *A Novel Approach to Automated Design Pattern Detection*. 2005. URL: https://users.uom.gr/~achat/papers/PCI2005.pdf.