Integrating User Knowledge into Design Pattern Detection

Thesis submitted for the degree of
Doctor of Philosophy
at the University of Leicester

by

MOHAMMAD H. ALSHIRA’H
DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF LEICESTER

DECEMBER 2015
Integrating User Knowledge into Design Pattern Detection

Mohammad H. Alshira’H

Abstract

Design pattern detection is useful for a range of software comprehension and maintenance tasks. Tools that rely on static or dynamic analysis alone can produce inaccurate results, especially for patterns that rely on the run-time information. Some tools provide facilities for the developer to refine the results by adding their own knowledge. Currently, however, the ability of tools to accommodate this knowledge is very limited; it can only pertain to the detected patterns and cannot provide additional knowledge about the source code, or about its behaviour. In this thesis, we propose an approach to combine existing pattern detection techniques with a structured feedback mechanism. This enables the developer to refine the detection results by feeding-in additional knowledge about pattern implementations and software behaviour. The motivation is that a limited amount of user input can complement the automated detection process, to produce results that are more accurate. To evaluate the approach we applied it to a selection of openly available software systems. The evaluation was carried in two parts. First, an evaluation case study was carried out to detect pattern instances in the selected systems with the help of the user knowledge. Second, a user study of a broader range of expert users of design patterns was conducted in order to investigate the impact of their knowledge on the detection process, and to see whether it is realistic that the user can identify useful knowledge for the detection process. The evaluation results indicate that the proposed approach can yield a significant improvement in the accuracy whilst requiring a relatively small degree of user input from the developer. Moreover, the results show that expert users can supplement the design pattern detection process with a useful feedback that can enhance the detection of design pattern instances in the source code.
First and foremost, this work would not have been completed except by guidance of Almighty Allah that allowed my dreams to come true. I would like to thank Allah for giving me the power to believe in myself and pursue my dreams.

I would like to take this opportunity to extend my deepest gratitude to my academic supervisor, Dr Neil Walkinshaw, for constantly offering adequate supervision, and keeping me on the right path during the work on this thesis. He dedicated numerous hours for giving me countless guidance, valuable suggestions, and continuous encouragement. I greatly appreciate his immeasurable efforts for bringing my dreams into reality.

I would like to thank my PhD co-supervisor, Professor Reiko Heckel, PhD tutor, Dr Fer-Jan de Vries, and Professor Thomas Erlebach for their guidance and support during my PhD journey. I would also like to thank all the staff members of the Department of Computer Science at the University of Leicester, especially the department manager, Mrs Karen Smith, for her kindness and encouragement. I would also like to thank the members of my thesis examining committee, Professor Rick Thomas, Dr Ruzanna Chitchyan, and Dr Steve Counsell, for their valuable comments and suggestions.

Many friends and colleagues have shared time with me, and they helped making my PhD enjoyable and make it memorable. I would like to thank Dr Mohammad Kharabsheh, Dr Abdullah AlSokkar, Amin Ajlouni, and Haitham AlZabin. I would also like to especially thank Mohammed Alabdullatif, Alexey Bakhirkin, and Thomas Gransden. I would like to express my deep and sincere gratitude to all those who have participated in our user study, shared knowledge and offered constructive feedback. I would also like to take this opportunity to gratefully and sincerely thank my sponsor, Al al-Bayt University, for granting me full scholarship to pursue my studies abroad.

Last but not least, I would like to thank my mother for being there for the happy times and tough times, for accepting me as I am and for guiding through my life. It would have been harder without her warm prayers. I would also like to thank all my family members especially my brother, Malek, for his support and help. I dedicate this thesis to my daughters, Mira and Sarah, who suffered the most as for not being there for them. I hope I could compensate the days I have spent away from them doing this research. Finally, to the memory of my father who passed away when I was three years old. Father, I have not been with you enough to know everything about you but I love you and miss you dearly.
Contents

Abstract i
Acknowledgements ii
List of Figures ix
List of Tables xi
Abbreviations xiii

1 INTRODUCTION 1
1.1 Research Context ........................................ 1
1.2 Reverse Engineering and Design Patterns .................. 3
1.3 Research Problem ........................................ 4
1.4 Research Objectives ...................................... 5
1.5 Thesis Outline ........................................... 7

2 DESIGN PATTERNS 9
2.1 History of Design Patterns ................................. 9
2.2 GoF Design Patterns ..................................... 11
   2.2.1 Example - Adapter Pattern .......................... 12
## 2.3 The Relevance of Design Patterns ................................................. 13
## 2.4 The Impact of Design Patterns .................................................. 14
## 2.5 Design Pattern Implementation .................................................. 15
## 2.6 Identification of Design Patterns .............................................. 16

### 3 BACKGROUND & LITERATURE REVIEW 18

#### 3.1 Design Pattern Detection (DPD) ............................................... 18

- **3.1.1 Classification of the literature in DPD** ................................... 21
  - **3.1.1.1 Recovery** .................................................................. 21
  - **3.1.1.2 Formalisation** ......................................................... 21
- **3.1.2 Measurement & Evaluation** ................................................. 22
  - **3.1.2.1 Precision & Recall** .................................................... 23
  - **3.1.2.2 Weighted F-Score** ...................................................... 24
  - **3.1.2.3 Role-Based Pattern Evaluation** ..................................... 24
  - **3.1.2.4 P-MARt Pattern Repository** ......................................... 25

#### 3.2 Design Pattern Detection Approaches ....................................... 26

- **3.2.1 Static Analysis** ............................................................... 30
- **3.2.2 Static/Dynamic Analysis** .................................................. 31
- **3.2.3 Constraint Satisfaction Problem & Pattern Detection** ............ 33
- **3.2.4 User Integration Approaches** .......................................... 35
- **3.2.5 Micro-structures Approaches** .......................................... 37

#### 3.3 Design Pattern Detection Tools ................................................. 39

- **3.3.1 Comparative Studies** ....................................................... 39
- **3.3.2 Tools with User Capabilities** ........................................... 40

#### 3.4 Weaknesses of Existing Techniques .......................................... 42

#### 3.5 Summary ................................................................................. 44

### 4 GUIDING DESIGN PATTERN DETECTION WITH HINTS 46
Contents

4.1 Motivating Example .............................................. 47
4.2 Overview of the proposed approach ............................ 50
4.3 Source Code Analysis ............................................ 52
4.4 Mapping to Constraint Satisfaction Problem .................. 53
  4.4.1 Variables ................................................... 53
  4.4.2 Constraints ............................................... 54
  4.4.3 Domain .................................................. 56
  4.4.4 Solution ................................................ 57
4.5 Refinement Process .............................................. 58
  4.5.1 Providing Feedback in the Form of Hints ................. 58
  4.5.2 Example: The Iterative Feedback Process ............... 61
  4.5.3 Using Templates to Guide Hint Selection ............... 63
4.6 Hint Generation Process ....................................... 64
  4.6.1 Methodology ............................................. 65
4.7 Implementation .................................................. 67
4.8 Related Work ................................................... 71
4.9 Summary ........................................................ 72

5 EXPERIMENTS ....................................................... 73
5.1 Case Study Research Question .................................. 73
5.2 Methodology ..................................................... 74
  5.2.1 Subject Systems ......................................... 77
5.3 Results & Discussion .......................................... 78
5.4 Evaluation Against P-MARt Repository ....................... 81
5.5 Reasons for False Detection ................................... 88
  5.5.1 Design Pattern Variants ................................. 88
  5.5.2 Dynamic Data Flow ..................................... 88
5.5.3 Parser Capabilities ............................................. 89
5.6 Threats to validity ................................................. 90
  5.6.1 Internal Validity .............................................. 90
  5.6.2 External Validity ............................................. 91
  5.6.3 Conclusion validity ......................................... 91
5.7 Summary .......................................................... 92

6 PATTERN DETECTION AND USER KNOWLEDGE .......... 93
  6.1 Aims and Objectives ............................................ 94
  6.2 Research Question & Hypotheses .............................. 95
  6.3 Methodology .................................................... 96
    6.3.1 Survey Design ............................................ 97
      6.3.1.1 Population and Sampling .............................. 97
      6.3.1.2 Questionnaire Form and Protocol .................. 98
      6.3.1.3 Pilot Test ........................................... 102
    6.3.2 Survey Operation ........................................ 106
      6.3.2.1 Preparation ......................................... 106
      6.3.2.2 Collection .......................................... 106
    6.3.3 Data Analysis ............................................ 107
      6.3.3.1 Data Coding ......................................... 108
      6.3.3.2 Descriptive Statistics .............................. 109
      6.3.3.3 Hypothesis Testing ................................. 110
      6.3.3.4 Test of Normality .................................. 110
  6.4 Results & Discussion .......................................... 112
    6.4.1 The Frequency of Hints .................................. 113
    6.4.2 The Relation between the Experience and Useful Hints .. 116
  6.5 Threats to validity ............................................ 118
6.5.1 Internal Validity .............................................. 118
6.5.2 External Validity .............................................. 119
6.5.3 Conclusion validity ........................................... 119
6.6 Summary .......................................................... 120

7 CONCLUSIONS AND FUTURE WORK 121
7.1 Overall Summary ................................................ 121
7.2 Conclusions ...................................................... 123
    7.2.1 Findings Significance in Design Pattern Detection .......... 124
    7.2.2 Findings Significance in Software Maintenance .......... 125
7.3 Future Research Directions ..................................... 126

A Implementation 129
    A.1 InFamix (.mse) file ......................................... 129
    A.2 Basic-parser ................................................. 131
    A.3 CSP Input .................................................... 132

B Detection Constraints 133
    B.1 Observer Design Pattern ..................................... 133
    B.2 State/Strategy Design Pattern ................................ 135
    B.3 Template Method Design Pattern ............................ 136
    B.4 Command Design Pattern .................................... 137

C Hint Templates 138
    C.1 Structural Hints ............................................. 138
    C.2 Sequential Hints ............................................. 140

D Survey Request and Questionnaire Form 141
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>D.1</td>
<td>Survey Request Letter</td>
<td>141</td>
</tr>
<tr>
<td>D.2</td>
<td>Questionnaire Form</td>
<td>142</td>
</tr>
<tr>
<td>D.2.1</td>
<td>Introduction</td>
<td>142</td>
</tr>
<tr>
<td>D.2.2</td>
<td>Demographic</td>
<td>143</td>
</tr>
<tr>
<td>D.2.3</td>
<td>Selected Scenario</td>
<td>144</td>
</tr>
<tr>
<td>D.2.3.1</td>
<td>Scenario 1</td>
<td>144</td>
</tr>
<tr>
<td>D.2.3.2</td>
<td>Scenario 2</td>
<td>146</td>
</tr>
<tr>
<td>D.2.3.3</td>
<td>Scenario 3</td>
<td>148</td>
</tr>
</tbody>
</table>

Bibliography

150
## List of Figures

1.1 Thesis Structure ................................................. 8

2.1 Adapter pattern structure [54] ................................. 12

3.1 Key design pattern detection steps ........................... 20

4.1 Example system, along with the corresponding method call graph 49
4.2 Command / Adapter patterns structures ....................... 49
4.3 An iterative pattern-detection approach ....................... 50
4.4 CSP Example - Variables ....................................... 54
4.5 Simple encoding notations types .............................. 55
4.6 CSP Example - Constraints ..................................... 55
4.7 CSP Example - Domain ......................................... 56
4.8 CSP Example - Solution ......................................... 57
4.9 Hints level of detail ............................................ 59
4.10 Adapter pattern constraints .................................... 61
4.11 Candidate Patterns .............................................. 61
4.12 Command pattern candidate .................................... 63
4.13 Example - Hint .................................................. 64
4.14 Famix Core elements [42] ..................................... 67
4.15 Source code parser .............................................. 68
List of Figures

6.1 Questionnaire Form ........................................ 101
6.2 Profile of respondents: Job Role .......................... 112
6.3 Profile of respondents: Level of experience ............... 113
6.4 Positive Hints Frequencies .................................. 115
6.5 Negative Hints Frequencies ................................... 115

A.1 Example of .MSE file produced by InFamix ................. 130
A.2 Example of Basic Parser output ............................. 131
A.3 Example of CSP input ........................................ 132

B.1 Observer Design Pattern ...................................... 133
B.2 Observer Pattern - Detection Constraints ................... 134
B.3 State/Strategy Design Pattern ............................... 135
B.4 State/Strategy Pattern - Detection Constraints ............ 135
B.5 Template Method Design Pattern ........................... 136
B.6 Template Method - Detection Constraints .................. 136
B.7 Command Design Pattern ..................................... 137
B.8 Command Pattern - Detection Constraints .................. 137

D.1 Questionnaire - Scenario 1 ................................. 145
D.2 Questionnaire - Scenario 2 ................................. 147
D.3 Questionnaire - Scenario 3 ................................. 148
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Summary of the literature review</td>
<td>28</td>
</tr>
<tr>
<td>3.2</td>
<td>Related work strengths and weaknesses</td>
<td>29</td>
</tr>
<tr>
<td>5.1</td>
<td>Subject systems</td>
<td>77</td>
</tr>
<tr>
<td>5.2</td>
<td>No hints/structural hints phase detection results</td>
<td>79</td>
</tr>
<tr>
<td>5.3</td>
<td>Sequential hints/Ad-hoc hints phase detection results</td>
<td>79</td>
</tr>
<tr>
<td>5.4</td>
<td>Results comparison based on P-MARt</td>
<td>82</td>
</tr>
<tr>
<td>5.5</td>
<td>Results accuracy comparing to P-MARt</td>
<td>82</td>
</tr>
<tr>
<td>5.6</td>
<td>Detected/shared pattern instances with P-MARt in JHotDraw</td>
<td>86</td>
</tr>
<tr>
<td>5.7</td>
<td>Detected/shared pattern instances with P-MARt in JUnit</td>
<td>87</td>
</tr>
<tr>
<td>5.8</td>
<td>Detected/shared pattern instances with P-MARt in QuickUML</td>
<td>87</td>
</tr>
<tr>
<td>6.1</td>
<td>Pilot Testing - Scenario 1</td>
<td>103</td>
</tr>
<tr>
<td>6.2</td>
<td>Pilot Testing - Scenario 2</td>
<td>104</td>
</tr>
<tr>
<td>6.3</td>
<td>Pilot Testing - Scenario 3</td>
<td>105</td>
</tr>
<tr>
<td>6.4</td>
<td>Measurement scale</td>
<td>108</td>
</tr>
<tr>
<td>6.5</td>
<td>The assessment of the usefulness of facts</td>
<td>109</td>
</tr>
<tr>
<td>6.6</td>
<td>Normality Testing - Scenario 1</td>
<td>111</td>
</tr>
<tr>
<td>6.7</td>
<td>Normality Testing - Scenario 2</td>
<td>111</td>
</tr>
<tr>
<td>6.8</td>
<td>Normality Testing - Scenario 3</td>
<td>111</td>
</tr>
<tr>
<td>6.9</td>
<td>Selection of Hints (in percentage of number of respondents)</td>
<td>114</td>
</tr>
</tbody>
</table>
### List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.10</td>
<td>The relation between the Experience and the Hints in Scenario 1</td>
<td>116</td>
</tr>
<tr>
<td>6.11</td>
<td>The relation between the Experience and the Hints in Scenario 2</td>
<td>116</td>
</tr>
<tr>
<td>6.12</td>
<td>The relation between the Experience and the Hints in Scenario 3</td>
<td>116</td>
</tr>
<tr>
<td>6.13</td>
<td>kruskall-Wallis Test</td>
<td>118</td>
</tr>
<tr>
<td>1</td>
<td>Structural Hints</td>
<td>139</td>
</tr>
<tr>
<td>2</td>
<td>Sequential Hints</td>
<td>140</td>
</tr>
</tbody>
</table>
### Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASG</td>
<td>Abstract Semantic Graph</td>
</tr>
<tr>
<td>AST</td>
<td>Abstract Syntax Tree</td>
</tr>
<tr>
<td>CFG</td>
<td>Control Flow Graph</td>
</tr>
<tr>
<td>CSP</td>
<td>Constraint Satisfaction Problem</td>
</tr>
<tr>
<td>DFA</td>
<td>Deterministic Finite Automata</td>
</tr>
<tr>
<td>DPD</td>
<td>Design Pattern Detection</td>
</tr>
<tr>
<td>DPML</td>
<td>Design Pattern Mark-up Language</td>
</tr>
<tr>
<td>EDP</td>
<td>Elemental Design Pattern</td>
</tr>
<tr>
<td>FOL</td>
<td>First-order predicate Logic</td>
</tr>
<tr>
<td>FUJABA</td>
<td>From Uml To Java And Back Again</td>
</tr>
<tr>
<td>GoF</td>
<td>Gang of Four</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>ICAM</td>
<td>Integrated Computer Aided Manufacturing</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>IDEF0</td>
<td>ICAM DEfinition for Function Modeling</td>
</tr>
<tr>
<td>JavaXL</td>
<td>Java eXtended Language</td>
</tr>
<tr>
<td>LOC</td>
<td>Line Of Code</td>
</tr>
<tr>
<td>LTL</td>
<td>Linear Temporal Logic</td>
</tr>
<tr>
<td>MOF</td>
<td>Meta Object Facility</td>
</tr>
<tr>
<td>OCL</td>
<td>Object Constraint Language</td>
</tr>
<tr>
<td>OOAD</td>
<td>Object Oriented Analysis and Design</td>
</tr>
<tr>
<td>OWL</td>
<td>Web Ontology Language</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td>P-MARt</td>
<td>Pattern-like Micro Architecture Repository</td>
</tr>
<tr>
<td>PADL</td>
<td>Pattern and Abstract-level Description Language</td>
</tr>
<tr>
<td>PDL</td>
<td>Pattern Description Language</td>
</tr>
<tr>
<td>PROLOG</td>
<td>PROgramming in LOGic</td>
</tr>
<tr>
<td>RDF</td>
<td>Resource Description Framework</td>
</tr>
<tr>
<td>RSF</td>
<td>Rigi Standard Format</td>
</tr>
<tr>
<td>SPQR</td>
<td>System for Pattern Query and Recognition</td>
</tr>
<tr>
<td>SWRL</td>
<td>Semantic Web Rule Language</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modelling Language</td>
</tr>
<tr>
<td>VPML</td>
<td>Visual Pattern Modeling Language</td>
</tr>
<tr>
<td>XML</td>
<td>EXtensible Markup Language</td>
</tr>
</tbody>
</table>
This thesis is lovingly dedicated to my mother who dedicated her life for me
Chapter 1

INTRODUCTION

This chapter introduces the research topic of this thesis. Section 1.1 discusses the context within which the research problems arise. Section 1.2 introduces the relation between reverse engineering and design patterns. Section 1.3 discusses the statement of the problem. Section 1.4 presents the overall aim and objectives of this thesis. It also summarises the research outcomes and contributions. Finally, Section 1.5 presents the thesis outline.

1.1 Research Context

Legacy software poses many important problems in software engineering. It needs to evolve in order to meet the continuously changing requirements [84]. Such software does not need to be substituted by new software but it needs to be maintained to meet the new requirements and business needs. However, legacy software is rarely accompanied by valid documentations and their source codes tend to be difficult to comprehend. As reported by Sommerville [114], 50 % - 75 % of the efforts spent on software development are dedicated to software maintenance.
Chapter 1. **INTRODUCTION**

Software maintenance tasks require a firm grasp of how the system was designed. Obtaining such understanding is a particularly time-consuming activity [31]. An object-oriented legacy code may contain thousands of classes and millions of lines of codes. Furthermore, the design documents are often missing, or may not accurately match the source code [90]. The main activity in the software maintenance process is program comprehension [31]. It involves inspecting the source code artefacts in order to gain a sufficient level of knowledge on the analysed system design and architecture. This activity provides insights into the source code, which can help the maintainer to understand the analysed system. Hence, the recovery of the source code design and architecture makes the maintenance task easier.

Detecting design patterns improves the program comprehension process by understanding the source code at the design level [128]. Design patterns are descriptions of solutions to frequently occurring software design problems [54]. The use of design patterns is now widespread (both in academia and in the industry), and is routinely taught in software design courses. Gang-of-Four (GoF) catalogue [54] is the most widely used classification for design patterns. This catalogue presents 23 different design patterns categorised according to their purpose into *Creational*, *Structural*, and *Behavioural* patterns. Each design pattern incorporates structural and behavioural components, and has a particular abstract model specification. Furthermore, each design has a unique intent that consists of a set of participants (classes) and described with roles and responsibilities [66].

Design patterns are rarely explicitly documented in the source code [98]. This is usually due to familiar factors such as the development time-pressure and the continuous code changes. The fact that a class plays a particular role in a pattern is often implied by its name or by its relations to the other classes. Several pattern-mining techniques have been developed to date to automatically highlight the presence of design patterns in the source code. Most of such techniques operate by matching a set of existing design pattern specifications to the source code elements (occasionally with the help of run-time information collected from the program executions).
Chapter 1. INTRODUCTION

1.2 Reverse Engineering and Design Patterns

Reverse engineering is the idea of reconstructing the design of existing software systems. Chikofsky and Cross [29] defined Reverse Engineering as:

“The process of analysing a subject system to identify the system’s components and their relationships and to create representations of the system in another form or at a higher level of abstraction”.

The aim of reverse engineering is to support software understanding, maintenance, re-engineering, and evolution activities by improving program comprehension tasks [127]. The use of design patterns in forward engineering is well-known. The idea behind reverse engineering of design patterns is to detect the presence of pattern instances in the source code of the analysed systems. This can provide better understanding of the software system.

Reverse engineering of the design patterns process utilises the source code of the analysed system as the main source of information. It starts by parsing the source code into an intermediate representation to extract the design facts (e.g. inter-class relationships). Then, it performs pattern recognition based on the extracted information. This is done by matching the extracted facts to the abstract specifications of the design patterns. In particular, it detects occurrence of micro-architecture that is similar to the pattern specifications [38].

The difficulty of reverse engineering patterns is not the same for all pattern categories. In structural patterns, most information can be recognised from the inter-class relationships, which can be trivially extracted from the source code. Behavioural patterns, on the other hand, are considered the most difficult pattern category to be detected [13]. This is because they rely on the availability of representative sets of program executions, and because they cannot be detected from the source code alone.
Chapter 1. *INTRODUCTION*

1.3 Research Problem

A number of approaches have been made during the last decade to reverse engineer design patterns. A representative list of such approaches can be found in [60, 93, 117]. These approaches utilise the source code as the main source of information for the analysed system. They developed meta-model catalogues, which contain possible formalisations for design patterns. Such catalogues are used to obtain the representation of design patterns and to allow for their automatic detection.

Current design pattern detection tools tend to rely mostly on a combination of a static and a dynamic code analysis. Such tools are inaccurate, especially when a design pattern has a substantial dynamic element. In fact, the set of pattern candidates retrieved by most of the currently available automated processes are not accurate. They, usually, fail to detect all the correct patterns and may also detect patterns that are not correct. Thus, the fully automated detection process is challenging. The principal reasons for this inaccuracy are as follows:

- Static analysis can be too conservative [46], which makes it difficult to differentiate between design patterns with similar structures. Design pattern detection tools often tend to rely on source code analysis where run-time behaviour can be difficult to predict.

- Dynamic analysis depends mainly on inputs from the developers which may be incomplete. Even if the tools incorporate run-time information, there remains the (undecidable) problem of identifying a suitable set of inputs to trigger the necessary information to elicit the behaviour that will expose the design pattern [46].

- Design pattern implementations can slightly deviate from the formal design pattern specifications used by the detection tools which make them harder to detect.
Some approaches have attempted to address these problems by involving user inputs. These approaches combine the automatic detection process with the inputs from the user, i.e. validate or discard the detection results. They consider the user interventions at some levels of their detection process. The user collaborates with the detection process unlike in the traditional detection (fully-automated). However, they permit a limited degree of user-interaction by allowing the user to annotate certain classes and to delete inappropriate role assignments. It was noticed with the approaches that support the user capabilities, the user has the ability to scrutinise the detection results at some level of the detection process. The user can inspect the final detection results by analysing the design pattern candidates and rolling out the false positives. However, the user of the tool has to spend much more time to discard the false positive than to trace the correct design patterns.

1.4 Research Objectives

The goal of this research is to develop a new structured design pattern detection process supported by inputs from an expert user in order to enhance the accuracy of the detection results. It considers the design pattern detection process as an iterative and a user-driven process. The motivation is that limited amount of user inputs can complement the automated analyses to produce detection results that are much more accurate. This thesis proposes an approach that seeks to enable the user to feed small hints into the detection process.

The work proposed in this thesis enables the developer to be much more expressive, whilst remaining easy to use. For example, hints can provide information about the dynamic behaviour of the program that might be impossible to verify from a finite number of program executions. Hints can also provide a partial knowledge about the implementations of a design pattern. The proposed approach is intended to be compatible with existing (constraint-based) pattern detection approaches in order
to act as a complementary layer to enable additional feedback from the developer. The following are the set of objectives:

1. Develop a new structured design pattern detection process to actively involve the user in the detection process, and place user knowledge at the heart of the design pattern detection process.
2. Enable expert users to supply their knowledge in an the form of gradually acquired constraints.
3. Study the impact of user knowledge on the design pattern detection process.

This thesis argues that there is a real need for a hybrid detection approach combining the automatic detection tools with the user inputs that support the design pattern detection process when automatic means reach their limits. This research proposes a framework enabling the user to supply domain knowledge in the form of constraints and providing means to integrate this knowledge into the detection process. In order to achieve the aforementioned goal, a new pattern detection framework was built, which incorporates the user knowledge, and implements a prototype tool based on involving the user inputs in the detection process. The evaluation was carried out with respect to real software systems. The general outcomes and main contributions achieved in this research are:

1. A user-driven design pattern reverse-engineering approach.
2. A set of ‘hint-templates’ to enable the easy supply of hints.
3. A proof-of-concept implementation.
4. An evaluation of the approach on three open-source systems.
5. A controlled user study of a broader range of expert users.

The key finding of this research is that it is necessary to actively involve the user expert knowledge in the design pattern detection process. The user must be provided with as much opportunity as possible to contribute their expert knowledge about the system structure, its behaviour, and possible implementation deviations from
traditional pattern definitions. The evaluation results indicate that user intervention has the potential to substantially increase the accuracy of the detected patterns. Moreover, the results of the user study show that the expert user can supplement the design pattern detection process with a useful feedback that can enhance the detection of pattern instances in the source code.

1.5 Thesis Outline

The structure of the thesis is shown in Figure 1.1. The thesis is organised into seven chapters, the remaining chapters are described as follows:

**Chapter 2** introduces the history of design patterns and explains how design patterns have been interpreted in the context of object-oriented design. This chapter also explains the general concept of design patterns, their classifications, and discusses the relevance of design patterns. Finally, this chapter discusses the impact of using patterns, explains the implementation and the identification of design patterns.

**Chapter 3** presents background on the research topic and covers the main concepts related to the design pattern detection process. This chapter also provides a review of a wide range of detection approaches and tools that have been proposed in the literature in order to assess the research contribution and develop a clear direction for the proposed approach. Finally, this chapter presents the main challenges in the design pattern detection area and highlights the problems and limitations of current approaches.

**Chapter 4** presents the proposed design pattern detection approach that places the user knowledge at the heart of the design pattern detection process. The approach is founded on the premise that the source code alone cannot provide
Chapter 1. *INTRODUCTION*

sufficient information to accurately identify the design patterns. This approach posits that the remaining information is supplied by the developer.

**Chapter 5** presents experiments of the proposed design pattern detection approach. The proposed approach was applied to three open-source systems and a comparison of the acquired results is presented in this chapter.

**Chapter 6** introduces user study conducted to investigate wide range of expert users. This study also aims to provide an evidence of the important role that can be played by the user expert in the design pattern detection process.

**Chapter 7** provides conclusions of the conducted research and describes further research issues to be considered as future work.

---

**Figure 1.1:** Thesis Structure
Chapter 2

DESIGN PATTERNS

This chapter aims to introduce the history of design patterns, and explains how design patterns have been interpreted in the context of object-oriented software design. Section 2.1 explains the history of design patterns. Section 2.2 outlines the general concept of patterns within object-oriented design. Section 2.3 discusses the relevance of design patterns. Section 2.4 discusses the impact of using design patterns. Section 2.5 explains their implementation. Finally, Section 2.6 discusses the identification of design patterns.

2.1 History of Design Patterns

Patterns were invented as an architectural concept by Christopher Alexander in the 1960s and the 1970s [9]. Alexander defines a pattern as “A recurring solution to a common problem in a given context and system of forces”. Alexander published several books on architectural design that were concerned with creating and using patterns in the architecture domain. Alexander described a variety of patterns in space, human existence, and events, with the aim of improving people’s living quality. In the following decades, the books of Alexander inspired the domain of Computer
Science. In 1987, Kent Beck and Ward Cunningham started experimenting with the idea of applying patterns to software programming (specifically pattern languages). In 1988, they [20] developed user interfaces in Smalltalk by using some ideas from Alexander’s architecture patterns. Starting from the 1990s, the work on patterns increased, as described below:

- In 1991, Jim Coplien developed a set of patterns called idioms in C++ [30]. These were a type of low level pattern specific to a programming language [27]. After that, Erich Gamma started to concentrate on recurring structures and patterns in his PhD thesis [52].
- In 1992, numerous professionals in software design including Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides congregated together to discuss patterns at the annual conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA) [11]. Later, this group of professionals came to represent the members of the ‘Gang of Four’.
- In 1993, the first version of a catalogue of design patterns was published [53]. This catalogue became the basis for the milestone textbook, which was published two years later.
- In 1995, the ‘Gang of Four’ revealed their milestone textbook Design Patterns: Elements of Reusable Object-Oriented Software [54].

The ‘Gang of Four’ book [54] has been seen as one of the most influential books in Software Engineering, and a milestone for the development of design patterns in the following years. Patterns can be applied to different object-oriented programming languages. Furthermore, patterns can be supported by functional programming just as they have confirmed to be in object-oriented programming [81]. Currently, design patterns can be applied in the areas of software design, configuration management, user interface design, web services, and interaction scenarios [114]. The research presented in this thesis concentrates on the Gof design patterns catalogued in Gamma et. al. [54] within an object-oriented context.
Chapter 2. DESIGN PATTERNS

2.2 GoF Design Patterns

With the publication of the ‘Gang of Four’ book [54], several of its patterns have become well-known in object-oriented software development. Design patterns are descriptions or templates for how to solve frequent software problems. They can be used in a variety of situations in object-oriented design. Design patterns are proposed as solutions for software problems in terms of interfaces, classes, and objects.

According to Gamma et al. [54], each design pattern can be expressed according to a template that provides information about its intent, structure, participant roles, and behaviour. Design patterns were categorised in the seminal work of Gamma et al. [54]. In this catalogue, they were categorised according to their purpose into creational, structural, and behavioural patterns. There follows, a brief overview of each category, followed by an example:

• **Creational Design Patterns** are concerned with the object creation process. The design patterns under this category abstract the process of instantiation. Creational design patterns encapsulate the knowledge about which classes the system uses, and hide the details of how the instances of these classes are created and put together. This category includes the following patterns: Abstract factory, Builder, Factory method, Prototype, and Singleton [54].

• **Structural Design Patterns** are concerned with the composition process. They provide techniques by which classes or objects are composed into a larger structure by utilising inheritance to compose interfaces or implementations. This category is useful to make independent classes work together. There are seven patterns under this category: Adapter, Bridge, Composite, Decorator, Facade, Flyweight, and Proxy [54].

• **Behavioural Design Patterns** are concerned with the relation between objects and classes. They are concerned with the communication and the distribution of responsibilities between classes or objects. They focus on how the
objects are cooperating for a given task, and how this task is distributed between different objects. There are eleven patterns under this category: Chain of responsibility, Command, Interpreter, Iterator, Memento, Observer, State, Strategy, Template method, and Visitor [54].

2.2.1 Example - Adapter Pattern

To explain the concept of a design pattern and its participating roles, we will consider the following example. The Adapter pattern [54] is categorised as a structural patterns. This pattern can be used to solve the design problem where two classes cannot communicate with each other for reasons such as an incompatible interface. This pattern converts the interface of one class into an interface that is expected by clients. The structure of the Adapter design pattern is shown in Figure 2.1. The participating roles are the Client, the Target, the Adapter, and the Adaptee.

![Figure 2.1: Adapter pattern structure [54]](image)

- Client: Collaborate with objects conforming to the target interface.
- Target: Defines the domain-specific interface that client uses.
- Adapter: Adapt the interface of Adaptee to Target interface.
- Adaptee: Defines an existing interface that needs adapting.
Chapter 2. *DESIGN PATTERNS*

Figure 2.1 illustrates the Adapter patterns structure and the participating roles, where the adapter class uses multiple inheritance to adapt one class to another. The intent of Adapter pattern is to convert the interface of a class into another interface. The collaboration between its roles indicates the Client calls operation on the Adapter instance. In succession, Adapter calls the Adaptee operation that carry out the Client request. The consequences of this collaboration have different trade-offs. The Adapter class overrides the Adaptee behaviour. The Adaptee class adapts to the Target by committing to a concrete Adapter [54].

2.3 The Relevance of Design Patterns

The use of design patterns is useful for program comprehension and quality improvements ([54], [109], [15]). Within the context of Reverse Engineering, information related to the presence of a design pattern is useful to understand not only the code, but to realise also the concepts behind its design. The use of design patterns enhances the maintainability of the source code. They help the developer to write a more understandable and a maintainable code [54]. The understandability and maintainability come with familiar solutions to common problems, instead of every developer trying to solve the problem in their own way. Moreover, the use of design patterns enhances the developer productivity and the program quality [99].

Design patterns provide repeatable solutions for software problems and make it easier to reuse successful designs and architectures [54]. Design patterns are not specific to programming and do not translate into a code directly like class libraries and frameworks, which are employed directly in programming [50]. They do not teach developers how to program, but they encapsulate the design concepts, and provide developers with suitable problem solutions from the design aspects. They help developers to select design alternatives as s small part of classes that make a system reusable and avoid alternatives that compromise re-usability.
Design patterns can improve the documentation and maintenance of existing systems [27]. They help developers to extend and modify software architecture and source code. They help the developer to describe the problem and its solutions. Consequently, this helps the maintainers to achieved a significantly better comprehension than the maintenance with source code alone [56]. Furthermore, design patterns enhance communication, both among software developers, among developers and maintainers [99]. They provide a common vocabulary to be used in the context of software problems and solutions. They help for sharing ideas between the developers and for gaining an understanding of design principles [27].

2.4 The Impact of Design Patterns

There is a misunderstanding that design patterns should be applied and followed wherever possible to solve design problems. However, it is very important to judge, when to use a pattern for a particular problem. The bad selection of design patterns can produce negative effects in the software development. This is due to the complexity that patterns introduce in the design and implementation of programs. Several studies in the literature suggested that the use of design patterns may impact negatively the development and maintenance activities.

- Wendorff [124] assessed the impact of patterns in some industrial projects and showed that their impact is comprehensibility negative. Khomh and Guéhéneuc [76] performed an empirical study of the impact of the Gof design patterns on different software quality characteristics. They concluded surprising results as reported in [75]. They found that patterns do not necessarily promote re-usability, expandability, and understandability and do not always improve the quality of programs as confirmed by Gamma et al. [54].
Chapter 2. DESIGN PATTERNS

• Bieman et. al. [23] examined the use of the design patterns presented in Gamma et al. [54] to see how design patterns are applied to real software development projects. This study explored the relationship between design structures in object-oriented software and development and maintenance changes. They analysed four small size systems and one large size system to identify the observable effects of the use of design patterns such as the pattern change proneness. The results showed that some classes that play roles in design patterns are more change prone than others. This indicates that design patterns are targets for bolting things on as they are less flexible.

• Zhang and Budgen [129] analysed a large amount of empirical studies to investigate what evidence is available about how and when their use can provide an effective mechanism for knowledge transfer about design. They found that design patterns do not help novice developers to learn about software design. Many developers used design patterns for their design problem and consider it as a solution of the problem. In addition, many developer focus on the identification and the documentation of patterns instead of their experiences about using the patterns because design patters are generated from the experiences of the software developers.

2.5 Design Pattern Implementation

A key benefit of design patterns is the fact of reusing design instead of code [40]. The structure of design patterns focuses on the relation of classes and objects rather than the specific code with respect to object-oriented relationships. As discussed above in Section 2.3, design patterns are not specific to programming languages. They provide solutions which can be described based in object-oriented design and can be implemented in any object-oriented programming language. The examples
in Gamma et al. [54] are in C++ and Smalltalk. However, design patterns are applicable to many of the object-oriented programming languages.

Design patterns aim to provide an access to successful solutions when software developers face recurring problems. Design patterns have been implemented in mainstream object-oriented programming languages like Smalltalk and C++ rather than in procedural languages (Pascal, C, Ada) [54]. Design patterns do not focus on coding and they are not specific methods as libraries and frameworks. They provide something much more like a guidance for software design and maintenance. However, libraries and frameworks focus on the specific application on programming, and they do not consider such design understandability and maintainability properties.

To explain the implementation of design patterns, we will consider the example in Section 2.2.1. The implementation of the Adapter pattern (shown in Figure 2.1) is straightforward and can be described as follow; The Adapter class offers an interface between classes and objects. The Adapter pattern roles are the Client, the Target, the Adapter, and the Adaptee. The Adapter must be a subclass of the Target and must delegate the Client calls to a method request() of the Target class to a method specificRequest() (with different interface) of the Adaptee class. To be able to do this, the Adapter instance needs to be associated to the Adaptee instance.

### 2.6 Identification of Design Patterns

The use of design patterns in the software development started in 1990s. However, the significance of using patterns in reverse engineering area was started with the publication of Krammer et al. approach [80]. Afterwards, different approaches have been proposed to detect design patterns in the source code. These approaches have utilised the source code as a main source of information for the analysed system, and have developed a meta-model catalogue. This catalogue contains possible formalisations of design patterns to be used to obtain their representations and allows
their automatic detection. The main idea of these approaches involves analysing the classes’ structures in order to find micro structures that are similar to the design pattern structure as described in Gamma et al. [54].

The detection of design patterns in legacy systems is difficult [17], because each design pattern incorporates structural and behavioural components [66]. Moreover, each design pattern has several possible implementations, where the implementation of design pattern may vary in the code style to represent or characterise the pattern [93]. In particular, the detection of behavioural patterns is challenging, since they are highly dynamic in nature, whereas tools often tend to rely on the source code analysis alone. Moreover, they are hard to characterise by their code structure. Behavioural patterns rely on the identification of the responsibilities and the collaboration between the objects at the program runtime, which cannot be simply identified.
Chapter 3

BACKGROUND & LITERATURE REVIEW

This chapter covers the main concepts related to design pattern detection process, and reviews some of past related work. Several approaches have been proposed to date, some of these approaches provide a tool to implement the proposed approach. Section 3.1 discusses the design pattern detection process, and the evaluation and measurement techniques that are currently used to evaluate the accuracy of design pattern detection approaches. Section 3.2 reviews some of the related approaches of design pattern detection proposed in the literature. Section 3.3 presents some of the design pattern detection tools that involve user-level of interventions in the detection process. The limitations and challenges of the current approaches are outlined in Section 3.4. Finally, a summary of the chapter is given in Section 3.5.

3.1 Design Pattern Detection (DPD)

There is a wide range of design pattern detection approaches, supported or implemented in different ways by several tools. Design pattern detection approaches
usually utilise the source code as the starting point for the detection process. The source code represents the main source of information related to the analysed system.

Figure 3.1 illustrates the process performed by most of the currently available design pattern detection tools along with the key steps of their detection techniques. The process starts from the source code of the analysed system as the main source of information about the legacy system, where the legacy system documentation is often lacking or lost [119]. The source code is parsed into an independent intermediate representation such as the Abstract Syntax Tree (AST) or the Abstract Syntax Graph (ASG), and then two types of analyses are required:

- **Static Analysis**: for program structure.
- **Dynamic Analysis**: for program behaviour.

The result of Static/Dynamic analysis of the source code contains:

- **Structural information**: inheritance/association etc.
- **Behavioural information**: this is a call from one class to another class, variable values, and object identities etc.

The detection algorithm receives the extracted design facts as an input, along with the design pattern meta-model (catalogue) that includes the formalisation of each design pattern as described in Gamma et al. [54]. Thus, it performs a matching process between the meta-model and the extracted facts. The results of the detection process can then be viewed in graphical representations (UML) or in a textual form (XMI) [14]. In detail, the design pattern detection process starts by parsing the source code to extract the design facts in order to build a model of the system. This model extracts all the static facts from the source code. The static analysis focuses on the structural aspect of the source code such as the inter-class relationships (i.e. association, aggregation, etc.). It is often useful to transform the extracted information from the source code into an independent intermediate representation (such as the abstract syntax tree (AST)) that helps to simplify the analysis process.
Figure 3.1: Key design pattern detection steps

The detection algorithm receives the extracted design facts as an input, along with a design pattern meta-model (catalogue) that includes a formalisation of each design pattern. The detection algorithm uses the extracted facts and the data gathered from the source code to identify the structural aspects of the design pattern. It works by matching a meta-model catalogue of the patterns to the extracted facts. The detection algorithm recognises the structures in the source code that are capture the structural aspects of the design patterns. This will produce the design pattern candidates that match the pattern structures.

On the other side, dynamic analysis can be exploited to verify the design pattern candidates produced through the static analysis [83]. It focuses on the behavioural characteristics of the design pattern. In particular, it uses the data obtained from the execution of the program to provide the behavioural information such as the call from one class to another, and the variable values, etc [46]. The result of the detection process can be viewed in a graphical representations (i.e. UML), or in a textual form (i.e. XML). The user can then analyse the detection results to validate or discard the design pattern instances with the help of the dynamic information.
3.1.1 Classification of the literature in DPD

Many research efforts were made to date in order to develop and implement tools to understand the legacy software systems through the recovery of the design pattern from the source code. As discussed in Section 3.1, the typical detection process comprises a set of key steps including: the analysis type (static, dynamic, or combination of both), the detection algorithm (i.e: CSP-based, Graph-based, fuzzy logic, etc), and the representation types of the analysed system (XML, UML, etc). For the ease of disposition, existing literature were classified into two interrelated fields: Recovery, and Formalisation.

3.1.1.1 Recovery

The general problem in the area of design pattern detection is building an effective approach that can detect the design patterns from the source code. This problem has been extensively studied over the past twenty years [35, 41, 77, 82, 110]. Many approaches and tools have been developed. Some of the well-known approaches that are relevant to the approach proposed in this thesis are reviewed in Sections 3.2.

3.1.1.2 Formalisation

As discussed above, detection algorithms receive extracted design facts as inputs, along with a design pattern meta-model (catalogue) that includes a formalisation of each design pattern. The GoF [54] design pattern formalisation is an informative definition to represent the design patterns structure. It is used to understand the design pattern principles, and to teach the user how to apply the design patterns in order to solve the common software problems. Thus, many studies in the literature adapted the building meta-model that represent the formal design pattern specifications. Many of the design pattern detection approaches developed to date are concerned with building formal specifications of the design patterns that capture
the structural and the behavioural features. These approaches are concerned with defining a set of meta-entities from which a design pattern description is obtained [8]. A considerable amount of literature has been published on building a design pattern formalisation.

Elaasar et al. [45] defined a formal specification for design patterns based on the Meta-Object Facility (MOF) Specification language. Each design pattern is formulated with the visual pattern modeling language (VPML), which is similar to the UML class diagram, with the ability to specify the design pattern abstraction through the use of object constraint language (OCL) contextual properties.

Mens and Tourwe [88] developed a framework to formulate the design patterns using declarative meta-programming. This approach uses a logic programming language SOUL to formulate the patterns, and to specify their constraints. It further supports the design pattern evolution transformations (re-factoring and pattern merge).

Bayley and Zhu [19] defined a meta-modelling approach for the formal specification of the design patterns based on a first-order predicate logic (FOL). Their design pattern formulation provides a specification for the structural and the behavioural features for all the GoF patterns.

### 3.1.2 Measurement & Evaluation

A large numbers of design pattern detection tools and approaches were proposed to date that are based on different detection techniques. The ultimate goal for all of these approaches is to produce high pattern detection accuracy rates. Therefore, there is a need to evaluate these approaches in terms of detection accuracy for comparison. In the following, a description of the Precision & Recall, weighted F-Score, and Role-based pattern evaluation:
3.1.2.1 Precision & Recall

Design pattern detection techniques are usually evaluated using precision \((P)\) and recall \((R)\) [96]. The precision and recall measurement were developed for the measurements of the natural language, i.e. in terms of Information Retrieval [121]. The relationship between \(P\) and \(R\) determines the accuracy of any detection approach. Ideally, a design pattern detection approach should have good precision and recall rates, where \(P\) values should remain high as \(R\) values increases. In this measurement, there are three possible outcomes for detecting the pattern occurrence including:

- **True Positive (TP)**: The detected pattern instance is implemented in the source code. This indicates the number of real instances of the design pattern implemented and identified by the pattern detection tool.

- **False Positive (FP)**: The detected pattern instance is not implemented in the source code. This indicates the number of instances detected by the tool but which are not correctly implemented in the source code.

- **False Negative (FN)**: The pattern is not detected but it is implemented in the source code. This indicates the number of pattern instances implemented in the source code which cannot be detected by the pattern detection tool.

A good and effective design pattern detection approach is ought to have a high rate of true positives TP and a low rate of false positives FP and false negatives FN. Based on the these outcomes, the precision and recall can be derived as follows:

- The **Precision** measures what proportion of the detected pattern instances are correctly implemented in the source code. It can be calculated using the following formula [95].

\[
P = \frac{TP}{TP + FP}
\]  

\[(3.1)\]
• The Recall measures what proportion of the actually implemented pattern instances are detected by the tool. It can be calculated using the following formula [95].

$$R = \frac{TP}{TP + FN}$$

(3.2)

3.1.2.2 Weighted F-Score

The use of precision and recall measurement is challenging since the increase in the precision values is often related to the decrease in the recall values and vice versa. Subsequently, Pettersson et al. [95] found that the precision and recall must be assessed in combination not individually. They proposed a standard solution to use the weighted harmonic mean of the precision and recall (weighted F-Score) as a complement to \(P \& R\) values, which can be calculated as:

$$F_w = \frac{(1 + w^2)PR}{w^2P + R}, w = 2\sqrt{2} \approx 2.8$$

(3.3)

The suggested value of \(w \approx 2.8\). With this weight, the highest F-Score is obtained, if both the precision and the recall are high. A precision of 100 percent with a recall of 100 percent, will give an \(F_w\) of 100 percent. A precision of 100 percent with a recall of 50 percent result in a value of \(F_w\) of 61 percent. In contrast, a precision of 50 percent with a recall of 100 percent give an \(F_w\) of 72 percent. Thus, Pettersson et al. [95] proposed to use this weighted F-Score as a complement to \(P\) and \(R\) values.

3.1.2.3 Role-Based Pattern Evaluation

As discussed in Section 2.2, each design pattern consists of a set of multiple roles. Although this is true, there are only few key-roles that can uniquely characterise the design pattern (e.g. the Adapter, and the Adaptee roles in the Adapter pattern). The non-key roles are the roles that do not affect the occurrence of their design patterns in the source code (e.g. Adapter pattern non-key roles are the Client and
the Target). The set of the detected roles are known as “the occurrence type” and the number of the detected roles are known as “the occurrence size” [95].

Generally, the detection result is represented as a set of the participating classes and interfaces with respect to the roles described in Gamma et al. [54]. Pettersson et al. [95] found that the precision values of the detected patterns are decreased when more roles of the pattern (occurrence size) are considered. Therefore, they proposed the use of different occurrence types to represent the detected patterns based on the pattern key-roles. Role-based pattern evaluation adapts the classical precision and recall metric. Instead of focussing purely on the exact detection of an entire pattern instance, they assumed that the evaluation should be carried out in terms of the proportion of the correctly assigned roles per pattern.

The detected pattern instances (occurrence type) can contain all the design pattern roles as described in Gamma et al. [54]. However, only the key-roles can uniquely determine a design pattern. Nevertheless, after detecting the key-roles, the non-key roles can be detected easily as they usually closely related to the key-roles. Furthermore, the consideration of the smaller occurrence type makes the results more comparable to other approaches.

3.1.2.4 P-MARt Pattern Repository

P-MARt [59] (Pattern-like Micro-Architecture Repository) is a repository of classes forming micro-architectures similar to the design patterns. This repository works as standard benchmark that helps in quantitatively analysing the results of the current design pattern detection approach results. In fact, the lack of standard benchmarks of design pattern instances makes the evaluation process challenging [95]. Most of the approaches in the literature to date were evaluated manually based on the evaluator judgement. However, some of the design pattern instances can be considered as a true positive by an evaluator and as a false negative by
another evaluator. Consequently, the peer reviewed patterns P-MARt repository was proposed as a standard comparison benchmarks.

P-MARt was built based on the manual investigation of several software systems to find micro-architectures similar to design patterns. In particular, P-MARt repository team [63] created this repository using different sources including: i) some studies in the design pattern literature, such as the study of Bieman et al. [23] who recorded the design pattern playing role classes in several C++, Java programs; ii) the results from Ptidej design pattern detection tool (Ptidej - Pattern Trace Identification, Detection, and Enhancement in Java) [7]; iii) the students manual analyses of Java programs, concluded from the assignment of their undergraduate course and graduate course. Then, they validated all these micro-architectures manually before their inclusion in the final repository. However, they do not claim that this a complete repository for all micro-architectures similar to design patterns in the given systems.

3.2 Design Pattern Detection Approaches

Table 3.1 provides a summery of all the design pattern detection approaches reviewed in this chapter. Many approaches and tools were proposed in the area of design pattern detection during the last decades since the introduction of the GoF design patterns by Gamma et al. [54] in 1995. This section covers some of the key approaches in the literature. In order to highlight the related work that specifically relevant to the approach proposed in this thesis, the relevant design pattern detection approaches can be classified using five categorises as follows: (Table 3.2 provides a summary of the strengths and weaknesses of each of these category)

1. Static: Static detection approaches utilise the structural aspect of the source code to capture the pattern instances. They exploit the structural relationships
between the classes that can be extracted from the source code to recognise the design pattern structures in the source code. Section 3.2.1 presents some of the well-known static approaches.

2. **Static/Dynamic**: Some approaches combined the static analysis with dynamic information to improve the accuracy of the detection results. These approaches utilise the run-time information, and rely on the availability of a representative set of program executions in order to capture the pattern behavioural aspects in the source code. Section 3.2.2 presents some of the approaches that utilised the dynamic information in the detection process.

3. **CSP-Based**: CSP-Based detection approaches consider the detection process as a constraint satisfaction problem (CSP). The approach proposed in this thesis is intended to complement existing constraint-based pattern detection approaches, such as DeMIMA [60] or Ptidej [7, 58, 89]. Some of the related approaches that are built based on constraint satisfaction technique (CSP-Based) are discussed in more detail in Section 3.2.3.

4. **User integration**: Some of the current approaches consider the user interventions at some levels of their detection process. The user collaborates with the detection process unlike in the traditional detection (fully-automated). These approaches combine the automatic detection process with the inputs from the user (i.e: validate or discard the detection results). Current user integration approaches are reviewed in Section 3.2.4.

5. **Micro-structures**: A *micro-structure* is a simple type of program construct or arrangement that can be represented as a property of a program element (classes, methods, and attributes, etc), or as a relation between these elements [14]. Micro-structures provide an indication for the presence of design patterns inside the code. This enables the users to validate or discard the results provided by the design pattern detection tools. Some of well-known micro-structures in the literature are presented in Section 3.2.5.
### Table 3.1: Summary of the literature review

<table>
<thead>
<tr>
<th>Approach</th>
<th>Technique</th>
<th>Tool</th>
<th>Experimentation</th>
<th>Analysis</th>
<th>Languages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kramer et al. [80]</td>
<td>Prolog-Based</td>
<td>PAT</td>
<td>NME/ LEIDA/ ZAPP/ ACD</td>
<td>Static</td>
<td>C++</td>
</tr>
<tr>
<td>Shi and Olsson [110]</td>
<td>Data flow &amp; control flow</td>
<td>PINOT</td>
<td>ANT/ JavaAwt/ JHotDraw/ JavaSwing</td>
<td>Static</td>
<td>Java</td>
</tr>
<tr>
<td>Dong et al. [39]</td>
<td>Matrix &amp; weight</td>
<td>DP-Miner</td>
<td>JavaAwt/ JHotDraw/ JUnit/ JEdit</td>
<td>Static Dynamic</td>
<td>Java</td>
</tr>
<tr>
<td>Kazes et al. [72]</td>
<td>Bit-vector</td>
<td>-</td>
<td>JHotDraw/ Juzzle/ QuickUML</td>
<td>Static</td>
<td>Java</td>
</tr>
<tr>
<td>DeLucia et al. [33]</td>
<td>Visual language</td>
<td>-</td>
<td>JHotDraw</td>
<td>Static Dynamic</td>
<td>Java</td>
</tr>
<tr>
<td>Heuerer et al. [69]</td>
<td>Prolog-Based/ runtime</td>
<td>-</td>
<td>JavaSwing</td>
<td>Static Dynamic</td>
<td>Java</td>
</tr>
<tr>
<td>Hayashi et al. [68]</td>
<td>Prolog-Based/ Meta-patterns</td>
<td>Implemented as Eclipse plug-in</td>
<td>Sample programs</td>
<td>Static Dynamic</td>
<td>Java</td>
</tr>
<tr>
<td>Wang and Tserepos [122]</td>
<td>REQL query/ ordered RSF</td>
<td>DPVK</td>
<td>EffleVisison/ Sample code</td>
<td>Static Dynamic</td>
<td>Eiffel</td>
</tr>
<tr>
<td>Wendehals and Orso [125]</td>
<td>Finite Automata</td>
<td>-</td>
<td>Java SWT</td>
<td>Static Dynamic</td>
<td>Java</td>
</tr>
<tr>
<td>Tsantalis et al. [117]</td>
<td>Graph-Based</td>
<td>DPD-tool</td>
<td>JHotDraw/ JRefactory/ JUnit</td>
<td>Static</td>
<td>Java</td>
</tr>
<tr>
<td>Keller et al. [74]</td>
<td>UML visualisation</td>
<td>SPOOL</td>
<td>Three industrial C++ systems/ RT++</td>
<td>Static</td>
<td>Java C++ Smalltalk</td>
</tr>
<tr>
<td>Rasool et al. [104]</td>
<td>SQL-queries</td>
<td>DRT-tool</td>
<td>Examples/ JHotDraw/ Apache Ant</td>
<td>Static</td>
<td>Java</td>
</tr>
<tr>
<td>Almusair et al. [10]</td>
<td>Ontology-based</td>
<td>Sempatrec</td>
<td>JUnit/ JHotDraw/ JRefactory</td>
<td>Static Semantic</td>
<td>Java</td>
</tr>
<tr>
<td>Niere et al. [93]</td>
<td>Fuzzy reasoning</td>
<td>Fujaba</td>
<td>Java AWT/ JGL libraries/</td>
<td>Static Semantic</td>
<td>Java</td>
</tr>
<tr>
<td>Rasool et al. [103]</td>
<td>Feature-Based</td>
<td>-</td>
<td>JUnit/ JHotDraw/ JRefactory</td>
<td>Static</td>
<td>C++</td>
</tr>
<tr>
<td>Gathéneuc and Antonioli [60]</td>
<td>CSP-Based</td>
<td>DoMIMA</td>
<td>JHotDraw/ JRefactory/ JUnit/ Map- perXML/ QuickUML</td>
<td>Static Dynamic</td>
<td>Java</td>
</tr>
<tr>
<td>El-Boussaidi et al. [44]</td>
<td>Constraint-Based</td>
<td>-</td>
<td>Sample inputs</td>
<td>Static</td>
<td>Java</td>
</tr>
<tr>
<td>Von Detton et al. [120]</td>
<td>ASG/UML sequence</td>
<td>Eclipse</td>
<td>Java AWT/ JGL libraries/ SWT/ JHotDraw/ JUnit</td>
<td>Static Dynamic</td>
<td>Java</td>
</tr>
<tr>
<td>De lucia et al. [34]</td>
<td>Structural properties</td>
<td>-</td>
<td>ePAD</td>
<td>Static Dynamic</td>
<td>Java</td>
</tr>
<tr>
<td>Approach</td>
<td>Description</td>
<td>Strength</td>
<td>Weaknesses</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Static</strong></td>
<td>These approaches exploit the structural relationships between the classes that can be extracted from the source code to recognise the design pattern structures in the source code.</td>
<td>These approaches utilise the structural aspect of the source code to capture design pattern instances, by focusing on the inter-class relationships and their dependencies.</td>
<td>- The detection results acquired by static analysis approaches show high rate of false positives (FP) and false negatives (FN).</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Static approaches are incapable of precisely detecting the design patterns that share the same structural aspects.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Dynamic</strong></td>
<td>These approaches combine the static analysis with dynamic information to improve the accuracy of the detection results. These approaches utilise the run-time information, and rely on the availability of a representative set of program executions in order to capture the pattern behavioural aspects in the source code.</td>
<td>- Design pattern detection approaches exploit the dynamic information to improve their detection results by reducing the false positives (FP) rate. However, The dynamic analysis appropriates with small number of execution traces (number of candidate classes).</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Dynamic analysis is effective in identifying behavioural patterns, but mainly when the design pattern detection tool is supported by a complete and representative set of test cases for the analysed system.</td>
<td>- The main difficulty in the dynamic analysis is the implementation variants of design pattern, where each pattern can have various possible implementations for the same expected behaviour.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Dynamic analysis approaches capture the system behaviour, but it is unable to verify the intent of the design pattern.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Dynamic analysis produces many false positives (FP) when the number of classes increases.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>User integration</strong></td>
<td>The user integration approaches consider the user interventions at some levels of their detection process. The user collaborates with the detection process unlike in the traditional detection (fully-automated).</td>
<td>- These approaches combine the automatic detection process with the inputs from the user to validate or discard the detection results.</td>
<td>- The user of the tool has to spend much more time to discard the false positive than to trace the correct design patterns.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Micro-Structures</strong></td>
<td>A micro-structure is a simple type of program construct or arrangement that can be represented as a property of a program element (classes, methods, and attributes, etc.), or as a relation between these elements.</td>
<td>- Micro-structure provides an indication for the presence of design patterns inside the code. This enables the users to validate or discard the results provided by the design pattern detection tools.</td>
<td>- The use of such kind of micro-structures alone is not enough to enhance the detection results and might increase the complexity of design patterns in the source code.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.2: Related work strengths and weaknesses
3.2.1 Static Analysis

Tsantalis et al. [117] proposed a graph-based approach for detecting patterns. This approach computes similarity scores between the graph vertices. They introduced a similarity measure between the matrices representing either the systems or the desired patterns. The directed graph can be built from the class diagram and mapped into a square matrix representation. This representation is matched using a similarity scoring algorithm against the design pattern descriptions. The similarity scores are reported to the user, with the purpose of giving the user an idea about the amount of matching.

Kramer et al. [80] proposed a knowledge-based approach based on Prolog rules detection. It extracts the design information from a C++ code and uses the Prolog rules to recover the design patterns. In particular, the source code is represented as a set of Prolog facts, and the design patterns are described as the Prolog predicates that are derived from the design patterns definitions in Gamma et al. [54]. Thus, Prolog queries are applied to detect instances of design patterns. This approach is limited to detect the Gof structural design patterns [54].

Shi and Olsson [110] proposed an approach based on a static program analysis which utilises the inter-class relationships in order to detect patterns. Furthermore, they proposed a new classification for all the GoF patterns in the context of reverse engineering. In this approach, they analysed the abstract syntax tree (AST) of the method bodies in order to build the control flow graph (CFG) for program elements. Then, the (CFG) is analysed to verify the presence of a pattern. They detected all the GoF [54] design patterns from the source code but with large number of false positives and false negatives.

Kaczor et al. [72] proposed an approach to detect patterns by formulating the design pattern detection as an approximate string matching using the bit-vectors algorithm [91]. They expressed the pattern detection with operations on finite sets
Chapter 3. BACKGROUND & LITERATURE REVIEW

of bit-vectors. The bit-vector algorithm searches for the exact and the approximate patterns instances by performing specific operations on the finite sets of the bit-vectors. The design patterns and the analysed source code are expressed in terms of string representations, which are represented by classes and their relationships (association, aggregation, composition, instantiation, inheritance). In particular, their algorithm matches the string representations of design patterns and the analysed source code by performing operations on finite sets of bit-vectors. They conducted an experiment to detect the abstract factory and the composite design patterns.

3.2.2 Static/Dynamic Analysis

De Lucia et al. [33] proposed an approach to detect the behavioural patterns, which combines the static and dynamic analysis. Firstly, static analysis is conducted in order to capture the structural aspects of the design pattern. This phase produces a set of candidate patterns. Then, the dynamic analysis is conducted on these candidates by tracing the calls of the methods at a program run-time. This approach is further extended by De Lucia et al. [35], they enhanced the approach of De Lucia et al. [33] by using a model checker to improve the design pattern candidates that are produced in the first phase (static analysis).

Heuzeroth et al. [69] proposed a detection approach that specifies the properties of the design patterns as predicates. The detection process is performed in terms of queries derived from the pattern specification. In particular, static analysis is performed on the abstract syntax tree (AST) in order to produce a set of candidates that match the patterns static rules. Then, the dynamic analysis takes these candidates as an input, to check whether each candidate does match the dynamic design pattern rules, and then accepts or rejects this candidate accordingly.

Hayashi et al. [68] introduced an approach based on static and dynamic analysis. The static analysis extracts the design facts such as the classes, the methods, etc.
These facts are then represented as facts in Prolog. The detection works as a Prolog program by matching the facts with a set of conditions that satisfy a particular design pattern. These conditions are defined as the Prolog rules. In dynamic analysis, the source code is executed to extract the execution records.

Wang and Tzerpos [122] proposed a design pattern detection approach for Effiel systems. Their approach describes each design pattern using its static structure and dynamic behaviour. In the static analysis phase, they use REQL [126] scripts to characterise the static structure of the source code. The detection matching is based on the structural definition of each design pattern. Then, they refine the detected instances against the behaviour definition of the design pattern, which is described in Rigi standard format (RSF). The final stage in this approach is the manual evaluation by the user. The user can manually verify and justify the candidate instances by filtering out the detection results, and determining undetected design pattern instances.

Wendehals and Orso [123] proposed a detection approach that combines the static analysis approach proposed by Niere et al. [93] with dynamic analysis. It utilises an automaton to represent the behavioural properties of the design patterns. The automata are manually acquired from the behavioural patterns. The dynamic analysis then watches the methods invocation during the execution, of the source code, and matches these invocations against the acquired automata. They verified the feasibility of their approach by detecting the State and Strategy patterns.

Dong et al. [39] proposed an approach to detect design patterns based on the use of matrix and weight. The detection process includes several analysis phases including structural, behavioural, and semantic. In particular, they built a matrix of the source code of the analysed system in the structural analysis phase. The source code classes are represented in rows and columns and the relationships between the classes are represented as a value in a matrix. Likewise, the design patterns are described as matrices and weights that derived from the GoF design pattern specifications [54].
Chapter 3. BACKGROUND & LITERATURE REVIEW

The detection algorithm works by matching the metrics and weights in arithmetic computations. The behavioural analysis checks method delegations and dependency links between classes as well as checks if a class delegates the method call. In the semantic analysis some naming conventions are verified. It explores the pattern related knowledge from the naming conventions of each class for indications of the roles that the class may play in the potential patterns. For example, a class with its name including "State" is highly likely to be part of the State pattern.

3.2.3 Constraint Satisfaction Problem & Pattern Detection

A constraint satisfaction problem (CSP) [116] can be represented as a set of variables and set of constraints that bounds the combination of concrete values of the variables in a specified domain. It is denoted by the triple <Variable (V), Domain (D), and Constraint (C)> where:

- **V** is a set of variables $v_1, \ldots, v_n$
- **D** represents the set $d_1, \ldots, d_n$ of the domains of the values such that every $v_i \in V$ is represented by $d_i \in D$.
- **C** represents a set of constraints $c_1, \ldots, c_m$, where each constraint limits the values in $D$ that can be taken by variables in $V$.
- **A solution** of such constraint satisfaction problem is a complete assignment of variables $s : v_j \rightarrow d_j$, which satisfies the constraints in $C$.

When considering the constraint satisfaction problem (CSP) from the perspective of design pattern detection [7], the domain is the source code of the analysed system, and the variables are the elements of the various design patterns that need to be detected. The relationships between the design pattern elements that constitute a particular design pattern are captured in the form of constraints. The solution of such CSP (in terms of design pattern detection) maps the design pattern elements to their corresponding elements in the source code of the analysed system.
Rudolf [106] proposed an approach to translate graph matching problem into equivalent constraint satisfaction problem. In this approach, each node and each arc of the graph is mapped to a distinct variable. For each node variable, the domain is the set of nodes in the target graph. For each edge variable, the domain is the set of edges in the target graph. The restrictions of a graph are expressed using constraints. Later, El-Boussaidi et al. [44] proposed an approach to apply Rudolf’s approach [106] to describe the design pattern detection problem using the constraint satisfaction technique. They utilised the explicit representation of the design patterns and a meta-model of the design patterns specification in Gamma et al. [54].

Albin-Amiot et al. [7] proposed a design pattern detection approach, that is further developed by Guéhéneuc et al. [89]. This approach examines the detection process as a constraint satisfaction problem. This approach solves the detection problem given by matching the actual source code structure to the design pattern structure. In particular, the problem domain is derived from the source code of the analysed system, while the constraints and the variables are derived from the design pattern abstract model. In this work, they introduced a meta-model Pattern Description Language PDL to describe and formalise the design patterns abstract model. Then, they used an explanation-based constraint solver called Palm [71] to solve the CSP problem. It detects the micro-architectures that are identical or similar to the design pattern in the source code. This approach detects the distorted micro-architectures of the design patterns in the source code. Thus, the detected instances of patterns were associated with a similarity rate. Furthermore, they use Java extended language (JavaXL) to modify the source code to comply with a design pattern description, this makes the distorted versions compliant with the design pattern models. Later, the improved approach of Guéhéneuc et al. [65], uses the program metrics (such as the size, the cohesion, and the coupling) and machine-learning algorithm to fingerprint the participating roles (classes) of a design pattern.

Guéhéneuc and Antoniol [60] proposed a multi-layered approach to detect the micro-architectures (motifs) similar to the design patterns in the source code. A motif is
typically a class diagram describing the solution to the recurring problem addressed by a design pattern. The first layer aims to obtain an abstract model of the source code. This model is expressed using a meta model inspired by UML, and includes all the constituents found in the object-oriented Java code: classes, method, inheritance, etc. The second layer aims to describes the system at a higher level of abstraction, it provides a model of the source code using the binary class relationship such as: the association, the aggregation, etc. The third layer aims to identify the design motifs in the abstract model. In particular, it transforms the definition of each design pattern using a set of constraints. These constraints are then used to match the design patterns against the obtained model from the source code. This approach uses the explanation-based constraint in order to find these micro-architectures. This approach exploits a constraint solver, called Palm [71], to perform an exact detection. The constraint solver performs an explanation-based constraint solving. This means that the results are paired with the list of the matched (or not) constraints. The PADL meta-model is also used to represent the detected design motifs. Each design motif is described in terms of the roles, the methods, the associations and the inheritance relationships among the roles. Later, Guéhéneuc et al. [61] proposed an extended approach based on the explanation-based constraint programming. They enhanced the approach of Guéhéneuc and Antoniol [60] using numerical analysis.

3.2.4 User Integration Approaches

Binun and Kniesel [24] proposed a detection approach based on enhancing and combining some variations to the output of different analysis strategies. They defined a methodology to improve the detection accuracy for such detection strategies by relaxing the restrictive constraints and combining the structural and the behavioural constraints. The role of the user in this approach is just to check if the detected design pattern candidates are incomplete. Then, the user can generate the needed re-factoring to make the complete implementation for the candidates.
Keller et al. [74] proposed a design pattern detection approach that splits the detection process into two phases, an automatic and an interactive. Their detection process contains two phases. In the first phase, the design pattern structure is identified, and then presented to the user as a UML diagram. They involved the user at the end of the detection process to refine the results by confirming or deleting the wrong candidates.

Rasool et al. [104] proposed an approach to detect the design patterns based on the annotations of the source code of the analysed system. These annotations are used in forward engineering to document the implemented design patterns. Annotations allow the transfer of knowledge between the users involved in the detection process. The user can select the specification language for theses annotations. Later, Rasool et al. [103] proposed an approach to detect the design patterns based on the features (structural) of the pattern. This includes the classes, the relationships, and the method returns. They built a catalogue of formal definitions of patterns features including the relationships between the classes, and the method return types. Then, they applied these definitions to multiple search techniques with the purpose of detecting design patterns. In their approach, the user can browse and highlight the detected design patterns within the class diagrams. The user can also select the feature type from a list of the available feature types, as well as defining new feature.

Alnusair et al. [10] proposed an ontology based approach to detect design patterns. In this approach, the ontology formalism are used in order to characterise the conceptual knowledge of the source code. The semantic rules are used to capture the structures and the behaviours of the design patterns in the source code. The user can relax/edit the rules using a specialised ontology editor.

Niere et al. [93] represented the source code as an abstract syntax graph (ASG). The detection process then works by matching the graph with a set of detection rules. This approach allows the user to annotate certain classes and to delete the inappropriate role assignments. The user has the capability to look at the detection results.
and mark the design patterns for deletion. In particular, the user can assign fuzzy thresholds for the detected sub-patterns that may influence the detection results.

Balanyi and Ferenc [16] proposed an XML-based detection approach using a specific language called DPML language (Design Pattern Mark-up Language). This approach uses DPML to describe the structural features of each design pattern including the classes, the attributes, the operation, and the relationships. It starts by analysing the source code in order to build the Abstract Semantic Graph (ASG). Then, the detection process works by matching the sub-structures in the ASG with the DPML descriptions to find the design pattern instances in the source code. This approach offers the user the ability to modify/define the design pattern descriptions.

Later, Ferenc et al. [47] enhanced this approach by applying machine learning techniques to improve the detection results.

### 3.2.5 Micro-structures Approaches

Micro-structures can help to identify aspects that are fundamental for the presence of design pattern instances inside the source code. Considering patterns as compositions of simpler elements can significantly reduce the possibility of identifying the design patterns implemented in the source code [14]. The use of micro-structures in the detection process can be important. This is because the results of the design pattern detection are commonly characterised by low precision. Therefore, having means to help the user to verify the output could improve the detection results. Different kinds of micro-structures were proposed to date in the literature. These can have different objectives, including the design pattern detection, the identification of common programming constructs, and the extraction of architectural relationships within the source code. The following are some of the most well-known micro-structures in the literature:
• **Elemental Design Patterns (EDPs):** EDPs represent an in between level of abstraction among the source code and the high-level formalisation of the design patterns [112, 113]. Each design pattern is expressed as a construction of EDPs. These EDPs capture the mechanisms that are specific to the object-oriented programming. This includes object creation, the abstraction of interface, the delegation, etc. EDPs represent an intermediate level between the source code and the design patterns, reducing the design pattern detection to the detection of EDPs. EDPs are associated to the functionalities of the design pattern structure [14].

• **Sub-patterns:** A sub-pattern is an individual description of the common structures of design patterns [92, 93]. The use of sub-patterns reduces the pattern structure into sub-patterns which are far simpler than the same design pattern. However, one problem is that each of these sub-patterns may have several variants. Besides, these sub-patterns are used by other design patterns. Sub-patterns are associated to the syntactic form of the structures of the design patterns.

• **Micro-patterns:** Micro-patterns are similar to the design patterns except that micro-patterns stand at a lower level that is closer to the implementation level. Gil and Maman [55] proposed a catalogue of 27 micro-patterns defined starting from classes and interfaces. They performed an analysis on the occurrence of micro-patterns and showed that a majority of classes follow one or more of the design patterns in the catalogue.

• **Design pattern Clues:** Clues are rules defined in terms of the roles constituting each design pattern, and in terms of the micro-structures that characterise them. Arcelli et al. [49] proposed an approach to refine and validate the results provided by the experimentation of common design pattern detection
tools. They manually evaluated the selected tools based on the system documentation (in case it traces the existence of design patterns within the system), and based on the personal knowledge about design patterns.

3.3 Design Pattern Detection Tools

Several design pattern detection tools have been implemented in the literature that provides different capabilities. However, the comparison between these tools is difficult due to the lack of a standard and a common comparatives benchmark [64]. For this reason, different comparative studies were conducted in order to investigate these approaches. Since the focal point in this thesis is the user intervention, some studies that highlighted the user’s role and level of intervention in the detection process are reviewed. This section starts by presenting some of the comparative studies for the current detection tools in Section 3.3.1, followed by a review of some well-known tools in Section 3.3.2.

3.3.1 Comparative Studies

Bellay and Gall [21] compared four detection tools according to four functional categories including: analysis, representation, editing/browsing and general capabilities. The analysis category illustrates the tools parsing capabilities. The representation category characterises the tools output form. The editing/browsing category compares the capabilities of the tools in terms of providing the users browsing and navigation capabilities. The general capabilities category concerns different aspects including the storing capabilities, the platform, and supporting multi-users.
Arcelli et al. [14] classified the design pattern detection tools based on the information used during the detection process to tools that consider the “entire” representation of the design patterns, tools that consider a ”minimal” set of key structures design patterns, and tools that consider the ”sub-components” design patterns.

Gueheneuc et al. [64] introduced a comprehensive comparative framework for current detection tools in terms of eight categories including the context, the intent, the users, the input, the technique, the output, the implementation, and the tool. The concern with this comparative framework is the user and the input categories. Under the user’s category, they compare the tools in terms of the tools’ assumption regarding the users, such as what is expected from the user. Furthermore, under the input category, their framework compares the tools in terms of the required inputs, such as what input does the detection tool accept.

### 3.3.2 Tools with User Capabilities

The following is a review of the most common design pattern detection tools that involve a user level of interventions in their detection process.

**ePAD** - Eclipse plug-in for Design Pattern Analysis and Detection [34]. ePAD is implemented as an Eclipse plug-in. It encloses two main detection phases; the structural analysis phase takes the source code of the system under study as input in order to extract the UML class diagram. Then, creates a textual representation of this diagram. The detection process works by matching this representation with the design patterns definitions library. The behavioural phase takes these candidates as inputs and validates them against the design pattern behaviour specification. This tool provides visualization feature of the detected pattern instances as a class diagram. This feature supports the user intervention in the analysis of the detection results. The users can access the source code of the classes participated in the detected pattern instances.
to analyse/re-factor it. For example, the user can assign an instance as false positive.

**Reclipse** - Reclipse is a Reverse Engineering tool suite based on FUJABA [120]. Reclipse tool works as a plug-in for the Eclipse IDE [1]. The Reclipse tool identifies the structure of the design patterns graphically, and, if applicable, identifies the design patterns behaviour in order to verify the detected candidate instances. Reclipse allows to graphically specify the patterns’ structure and its behaviour to inspect the obtained candidate instances. It allows the user to annotate certain classes, and to delete inappropriate role assignments.

**PTIDEJ/DeMIMA** - Pattern Trace Identification, Detection, and Enhancement in Java) [4]. Ptidej comprises a set of tools that provide different tasks with the main aim of identifying the design patterns and enhancing the quality of the object-oriented code. It provides the capabilities of detecting the design pattern instances in the source code using a constraint solver. Ptidej uses the explanation based constraint programming to identify the design patterns in the source code. DeMIMA [60] (Design Motif Identification Multi-layered Approach), on the other hand, is a tool implemented on top of Ptidej framework using Java programming language. It adopts the existing libraries of Ptidej framework to use the constraint solver. It utilises the explanation-based constraint programming and the constraint relaxation to identify the complete (or the approximate) micro-architectures, similar to the design patterns. In this tool, the user must be familiar with the analysed system source code and can provide explanations on the detected design pattern candidates. Moreover, this tool allows the user to relax the detection constraints to provide approximate occurrences.

**DPJF** - Detect Patterns by Joining Forces, DPJF is the resulting tool for the approach of Binun and Knives [24]. DPJF is developed as a set of SWI-Prolog modules. The role of the user in this approach is just to check if the
detected pattern candidates are incomplete. The user then can generate the
needed re-factoring to reach the complete implementation of the design pattern
candidates.

**DPD-tool** - Design Pattern Detection Tool (DPD-tool) is detection tool imple-
mented in the approach of Tsantalis *et al.* [117]. It provides the user with a
GUI allowing it to select the software system that needs to be analysed and
to generate a list of the recovered design pattern instances.

### 3.4 Weaknesses of Existing Techniques

Empirical studies on the accuracy of design pattern detection approaches have re-
peatedly shown that, for certain patterns and systems, existing tools can remain
highly inaccurate [101]. There are two principal reasons for this inaccuracy. First,
certain patterns can be highly dynamic in nature, whereas tools often tend to rely
only on the source code analysis. Even if the tools do accept the runtime-information
as the input, the (undecidable) problem of identifying a suitable set of inputs to
elicit the necessary information to detect the design pattern still remains. Second,
design pattern implementations can be an approximate. The developer might have
the pattern intent in mind, but it might deviate from the corresponding pattern
specification used by the tool.

Static analysis alone is not sufficient. Static analysis can be too conservative as
it is difficult to detect the design patterns that have similar structure. Moreover,
design patterns with behavioural aspect rely on the dynamic method invocation.
In fact, dynamic analysis is relying on the availability of runtime information (i.e.
method execution), which is challenging in reality, particularity if the system under
analysis is an old system. Current design pattern detection tools tend to rely entirely
on a combination of static and dynamic code analysis. Such tools are inaccurate,
especially when the design pattern has a substantial dynamic element. The specific
problems of design pattern detection and the key reasons for this inaccuracy are outlined below:

- **Source code analysis is inherently inaccurate:**
  Static source code analysis is necessarily conservative [46]. The inherent inability to distinguish between feasible and infeasible runtime behaviour means that it is impossible to, for example, accurately infer the possible runtime-types of an object. This means that any pattern detections that are based on static analysis alone can include a large proportion of false-positives, especially when the design patterns in question include a substantial dynamic element.

- **Dynamic analysis is inherently unsound:**
  Conversely to static analysis, dynamic analysis is highly precise (any observed execution is by definition feasible), but is also inherently unsound [46]. It is generally impossible to identify a finite sample of program executions that are guaranteed to capture the full range of a program behaviour (especially without any substantial a-priori knowledge of the implementation - as it is usually the case in pattern detection).

- **Patterns are often underspecified:**
  Patterns are underspecified since they were proposed as templates or guidelines to solve the design problem and can be implemented using different object-oriented programming languages [54]. Design pattern formalisation is used to understand the design pattern principles, and to teach the user how to apply the design patterns in order to solve common software problems [54]. However, there can be several other properties that intuitively apply to the pattern, but are not explicitly specified (mainly because they are presumed to be obvious from a developer’s standpoint).

Pattern implementations inevitably deviate from the template due to its intrinsic generality [107]. Most design pattern detection tools use specifications of design patterns that almost directly reflect their original templates (e.g. in
Gamma et al. [54]). These capture the essence of what should occur within a pattern implementation. However, they do not contain other potential variations that would probably be obvious to the developer, but are ignored by the tool-based approaches. If one tries to address this by adding in as many constraints as possible, there is a real risk that, for certain projects that implement the patterns in non-orthodox ways, these overly strict definitions will fail to detect many patterns.

- **The intent of a design pattern can be undecidable:**

  Design patterns can share the same syntactical structure, but can serve completely different purposes. Determining which (if any) design pattern was intended by the developer cannot be decided by an algorithm, but the invariably requires a degree of user judgement.

These factors can often conspire to make accurate pattern detection virtually impossible. If the software under analysis involves complex class hierarchies, the possible runtime object types become increasingly hard to determine. If the system in question is a library or a framework, it can become even harder to collect a ‘representative’ set of execution traces for dynamic analysis. In this situation, behavioural design patterns that are loosely specified but contain a substantial dynamic element can become even harder to detect if their structure is shared by other patterns with different intents. This is the case for the design patterns including: State, Strategy, Bridge, Chains of Responsibility, Decorator, and Proxy [54].

### 3.5 Summary

Many approaches and tools used in the process of design pattern detection were reviewed in this chapter. Based on the review of the literature in the design pattern detection, and given the aforementioned challenges in Section 3.4, the design pattern detection process cannot be completely automated. The set of pattern candidates
retrieved by the automated processes does not, in general, coincide with the set of the correct design patterns implemented in the source code. Indeed, the fully-automated detection process will fail to retrieve all the correct design patterns, and may also retrieve the design patterns that are not correct. Therefore, there is a need for a hybrid approach that combines the automatic tools with the user inputs.

The use of micro-structures is motivating because the detection results are commonly characterised by a low accuracy. Therefore, having a mean to help the user to verify the produced output could increase the accuracy of the detection results. In addition, considering a design pattern as compositions of simpler elements may significantly enhance the possibility of identifying the design pattern in the source code. Micro-structures can provide the basic information that can provide an indication for the presence of design patterns inside the code. However, the use of such kind of micro-structures alone is not enough to enhance the detection results. The motivation is that the integration of the user in an iterative process helps to refine and gradually enhances the accuracy of the detection results.

The main point of this thesis is to highlight the users’ role in the design pattern detection process, and how this knowledge can be used to enhance the detection results. It is important to note that the user plays a significant role in the design detection process. No one can consider the detection process as a fully-automated process. However, it is the semi-automated process that needs the user judgements at the some point when the automatic means is limited. The motivation here is to investigate the users’ knowledge, and to study the impact of this knowledge in the design pattern detection process.
Chapter 4

GUIDING DESIGN PATTERN DETECTION WITH HINTS

A new design pattern detection approach is proposed in this chapter that places developer knowledge at the heart of the design pattern detection process. The aim is that developer-supplied hints can counter-act the core problems discussed in Section 3.4. Essentially, a ‘hint’ enables the developer to provide a small fact about the system in question, or about the implementation of the patterns, which cannot be readily derived from the source code alone.

The proposed approach seeks to enable the user to feed small ‘hints’ into the detection process. Most existing tools permit a limited degree of user-interaction. For example, some tools allow the user to annotate certain classes, or to delete inappropriate role assignments. However, the approach proposed in this chapter enables the developer to be much more expressive, whilst remaining easy to use.

Hints can provide vital information about the dynamic behaviour of the program that might be impossible to verify from a finite number of program executions. Hints can also provide a partial knowledge about the implementations of the design pattern. The proposed approach is intended to be compatible with existing
(constraint-based) pattern detection approaches, to act as a complementary ‘layer’ to enable additional feedback from the developer.

Depending on the developer’s familiarity with the system, they could, for example, assert that a particular method call certainly cannot happen (despite the fact that it might appear in the static call graph), thanks to their knowledge of the run-time object types. The developer could state that a particular class or method certainly does or does not play a particular role within a design pattern. They could also, for example, search for a stricter definition of the design pattern than is commonly used, or only want to search for occurrences where a specific method is only ever invoked after a call to another method.

Section 4.1 presents a motivating scenario to motivate and explain the proposed approach. Section 4.2 introduces an overview of the proposed design pattern detection approach. Section 4.3 presents the source code analysis process used in the proposed approach. Section 4.4 presents the mapping of design pattern detection process to constraint satisfaction problem. Section 4.5 presents the refinement process. Section 4.6 presents the hint generation process. Section 4.7 presents the implementation of a proof-of-concept tool. Section 4.8 presents the related work to the approach proposed in this chapter. Finally, Section 4.9 presents a summary of this chapter.

4.1 Motivating Example

To illustrate the proposed approach, a small toy system example is presented in this section, which is used as a running example throughout this chapter. Suppose that the system shown in Figure 4.1 (a) is under investigation. The extracted call graph is shown in Figure 4.1 (b). The developer is interested in detecting any instances of Command pattern [54]. This pattern is identical in structure to the Adapter pattern [54]. Both design patterns are identical, with a substantial dynamic element, and
are shown in Figure 4.2. Though identical in structure, the patterns are completely different in purpose.

The Command pattern [54] provides an interface that manages the operations, encapsulating all the information as one object to be able to call other methods. The main participating roles include the Command, the Client, the Invoker and the Receiver. Invoker determines when the method will be called and the Receiver performs the operations. In detail, the abstract class Command, with a method execute(), has a number of Concrete classes ConcreteCommand, which override the execute() operation. The Invoker class aggregates with the Command class and keeps references for all the derived ConcreteCommand classes. Each ConcreteCommand class encapsulates Receiver objects, which are used to invoke the Receiver actions().

The Adapter pattern [54] provides an interface between all the classes and the objects. The participating roles are the Client, the Target, the Adapter, and the Adaptee. The Adapter must be a subclass of the Target and must delegate the Client calls to a method request() of the Target class to a method specific-Request() (with different interface) of the Adaptee class. To be able to do this, the Adapter instance needs to be associated to the Adaptee instance.

Typical tools would be forced to suggest that both Adapter and Command patterns fit the system shown in Figure 4.1. In this example, there are several possible instances of the Command pattern. However, each Command pattern instance could represent an Adapter pattern instance as well. The AInvoker class could also correspond to the Client role in the Adapter pattern. C1, C2 and C3 could fulfil the Adapter role, etc. Ultimately, there is no specific mechanism to guide the tools, in a systematic manner, towards making the correct choices of patterns, or to give them the additional information they require to make accurate decisions about patterns.
Figure 4.1: Example system, along with the corresponding method call graph

(a) Example system

(b) Call graph

Figure 4.2: Command / Adapter patterns structures
4.2 Overview of the proposed approach

The approach we will propose belongs to a broader family of design pattern detection techniques that are based upon Constraint Satisfaction Problem (CSP). The goal of this approach is to provide a framework within which the developer is able to gradually add missing information to the detection process in the form of ‘hints’. These hints are fed as small units of knowledge about the design pattern that can guide the constraint solver towards a more accurate set of design patterns.

The approach is founded on the premise that the source code alone cannot provide sufficient information to accurately identify the design patterns. A schematic diagram of the proposed approach is shown in Figure 4.3. The basic flow from the source code to the constraint solver mirrors the flow adopted by the conventional tools such as the tools of Ptidej [58] and DEMIMA [60]. The following is a brief

Figure 4.3: An iterative pattern-detection approach
description of the key elements in this approach: (More detailed description for each elements will be provided in later sections)

1. **Source code analysis:**

The source code is the main source of information regarding the analysed system, and forms the search domain within which the user seeks to identify the design patterns. In the source code analysis, the source code is parsed to extract its main structural elements (classes and their methods), and the relationships between them (inheritance, associations, etc.). This step is akin to the step used by approaches such as Ptidej [58] to transform the source code into the PADL description language.

2. **Mapping to Constraint Satisfaction Problem (CSP):**

The constraint satisfaction problem CSP is constructed from the source code and the abstract model of the design patterns. In particular, as discussed in Section 3.2.3, the main parts of the CSP are the Variables $V$, the Constraints $C$, and the Domain $D$:

- The variables $V$ are derived from the design patterns definition. They represent the source code element (a class, or an interface) that constitute a given role of a design pattern based on the design pattern abstract model in Gamma *et al.* [54].

- The constraints $C$ between the variables specify how the source code elements are related and how the design pattern elements need to be related. The relationship between these elements are represented based on the design pattern abstract model in Gamma *et al.* [54].

- The domain $D$ is derived from the source code. It represents the set of elements in a given source code along with their relationship.

The constraint solver provides a solution for the constraint satisfaction problem CSP by identifying the assignments from the domain $D$ for the variables $V$
that obey the constraints $C$. The solution of such a CSP maps the design pattern elements to their corresponding elements in the system. It provides a set of elements from the given source code whose relationships satisfy the constraints of the CSP problem.

3. **Refinement Process:**

At this point, the conventional design pattern detection process is accomplished. Inevitably (due to the reasons discussed in Section 3.4), some of the detected design patterns will be incorrect or entirely superfluous. As shown in Figure 4.3, the proposed design pattern detection process is iterative. The developer has the opportunity to inspect the current proposed solution and pick out any obvious problems. This is accomplished by, for example, inspecting the source code to see whether any of the code comments or identifiers corroborate (or contradict) a potential pattern. The developer might wish to carry out a cursory dynamic analysis (e.g. with the help of a debugger) to, for example, determine whether a particular method really does override another method, or whether a class can take on a particular role in a design pattern. The refinement information are fed back to the detection process in the form of additional constrains.

### 4.3 Source Code Analysis

Generally, in traditional design pattern detection process, it is common for the source code to be the main source of information for the analysed system. The source code works as an input to the detection process. In this phase, the source code is parsed to extract the design facts (classes, methods, and attributes). In particular, the source code analysis phase produces the static model for the source code architecture and behaviour (statically). The architecture model contains all the classes, the methods, the attributes, and their types. The behaviour model contains
4.4 Mapping to Constraint Satisfaction Problem

A relatively broad family of existing pattern detection techniques treat the design pattern detection as a constraint satisfaction problem (CSP) [7, 13, 44, 62]. A set of constraints are used to represent the design pattern abstract model. These constraints are derived from the abstract models (specifications) that describe their structure. These constraints incorporate the classes, the methods and the relationships between the classes such as the inheritance, the association.

The constraint solver receives the extracted design facts as an input, along with the design pattern constraints. It subsequently performs a matching process between the pattern constraints model and the extracted facts. It reports all the structure that matches the abstract model. As a result of this step, the design pattern candidates are produced. When considering a CSP from the perspective of the design pattern detection [7], the main parts of this CSP problem contain the variables $V$, the domain $D$, the constraints $C$, and the solution. As an example of how to translate a design pattern detection problem into a CSP, consider the running example presented in Figure 4.1 as follows:

4.4.1 Variables

Variables are roles in design patterns for which the goal is to find a source code element. In particular, variables are either (a classes, or an interface) of the various
design patterns need to be detected. The variables are the elements that the developer needs to detect by assigning the design pattern roles to the elements of the source code of the analysed system. Figure 4.4 shows the variables for the Command pattern, which represent the roles that need to be identified in the source code (or its respective abstraction as shown in Figure 4.2).

\[ V = \text{command, concreteCommand, invoker, receiver} \]

Figure 4.4: CSP Example - Variables

### 4.4.2 Constraints

The set of constraints \( C \) capture the relationships between the variables. In particular, this set of constraints characterise the individual design patterns. This set correlates to the relationship among the design pattern elements/roles that constitute a particular abstract model of a given design pattern. The knowledge about existing entities and relationships within the system can be captured using the simple notations, the various types of encoding notations shown in Figure 4.5 can be used to directly encode the relationships as shown in Figure 4.2.

Figure 4.6 represents the constraints \( C \) that capture the relationships between the Command pattern elements (constitute a Command design pattern specification in Gamma et al. [54]). The terms used to express the constraints depend on various factors including the choice of constraint solver and the sort of information the user seeks to reason about. A notation was adopted in this approach that is as intuitive as possible, and captures the key relationships within an object-oriented system.
all-classes(): All the classes.
all-methods(): All the methods.
all-attributes(): All the data types.
override(c₂,m₁): Class c₂ provides different implementation for a method m₁ that is already defined and/or implemented in its parent class.
invoke(m₁,m₂): Method m₁ calls method m₂.
super-class(): A class from which other classes are derived.
sub-class(): a derived class that inherits from the super class.
has-method(c₁,m₂): Class c₁ has a method m₂.
abst-method(m₁): Method m₁ is declared abstract.
protected-method(m₂): Method m₂ is declared protected.
final-method(m₁): Method m₁ is declared final.
inherited(c₂,c₁): Class c₂ inherit from class c₁.
abst-class(c₁): Class c₁ is declared abstract, it may include abstract methods.
Interface(c₁): Class c₁ is an interface, all methods in an interface are abstract.
par-Declared(c₁,par₁): Data type par₁ declared in class c₁.
par-Parent(c₂,par₁): Data type par₁ is type of class c₂.
same-Signature (m₁,m₂): Method m₁ and m₂ have the same name.
aggre(c₁,c₂): Class c₁ aggregates class c₂.
asso(c₁,c₂): Class c₁ associates class c₂.
dominate(m₁,m₂): For any sequence of calls m₂ must be preceded at some point by m₁.

**Figure 4.5:** Simple encoding notations types

<table>
<thead>
<tr>
<th>Command Design Pattern:</th>
</tr>
</thead>
<tbody>
<tr>
<td>super-class(Command)</td>
</tr>
<tr>
<td>sub-class(ConcreteCommand)</td>
</tr>
<tr>
<td>inh(ConcreteCommand,Command)</td>
</tr>
<tr>
<td>aggre(Invoker,Command)</td>
</tr>
<tr>
<td>asso(ConcreteCommand,Receiver)</td>
</tr>
</tbody>
</table>

**Figure 4.6:** CSP Example - Constraints
4.4.3 Domain

The domain of the variables is the set of possible source code elements and their relationships. The domain $D$ is constructed by parsing the source code to extract the relevant facts. The domain $D$ of the facts extracted from the motivating example in Section 4.1 is shown in Figure 4.7.

Figure 4.7: CSP Example - Domain
4.4.4 Solution

A solution of such CSP problem maps the elements in the system (domain) to their respective roles (variables). The solution of the CSP problem produces a set of entities from the source code whose relationships satisfy the pattern constraints. The solution for the Command pattern constraints is shown in Figure 4.8.

![Command Pattern Candidates:](image)

Command Pattern Candidates:

Command: Com
ConcreteCommand: C1, C2, C3
Invoker: AInvoker
Receiver: Receiver1, Receiver2

Figure 4.8: CSP Example - Solution

Detecting the Command pattern using only the typical detection roles (constraints) produces many false positives. It reports all class structures that match the Command pattern structure. This means that it will report the Adapter pattern as a Command pattern because the two patterns share the same structure. Even though the detected structures directly reflects the pattern specification given by Gamma et al. [54], one could argue that it is underspecified.

There are several other properties that intuitively apply to the pattern, but are not explicitly specified (mainly because they are presumed to be obvious from a developer’s standpoint). For example, we would not expect an implementation of the pattern to contain a method call from `Adaptee.specRequest()` to `Adapter.request()`, or `Adaptee.specRequest()` is only ever invoked after a call to `Target.request()`.
4.5 Refinement Process

The final phase of the proposed design pattern detection approach is the integration of the user knowledge. Before integrating the user knowledge to the detection process, three assumptions were made that are related to the expert users. Firstly, it was assumed that the user has a good knowledge of the analysed system. Secondly, the user must be familiar with the design patterns structures. Thirdly, the user has a good programming skills.

This phase enables the user to refine the CSP results by providing small additional constraints which we refer to as “Hints”. These are small facts about the system that might be reflected by the developer knowledge or intuition, but cannot be detected by analysing the source code. The primary aim is to correct any obvious mistakes in the solution generated by the constraint solver. However, the supplied hints need to take the form of additional constraints that can be added to the constraint satisfaction problem in order to be processed by the constraint solver.

4.5.1 Providing Feedback in the Form of Hints

Hints are small facts that can provide a sort of useful information about individual design pattern role or to identify relationships between two roles. The hints can capture arbitrary the properties of the detected patterns or of the behaviour of the underlying system. The ‘direct’ properties that deal with the design patterns are straightforward to envisage. Hints take a form of constraints that are represented as program elements (classes, methods, and attributes) or as relations between these elements. Hints are derived based on the personal knowledge of the design patterns structure and implementations.

Figure 4.9 shows the hints types, hints at the program level can provide information about the pattern behaviour of the program that might be impossible to verify from
a finite number of program executions (sequential). They can also provide partial knowledge about the design pattern implementations themselves (structural). For example, the developer could identify a common relation between the design pattern roles. These hints give the user more information related to single role that can be constituted with various design patterns.

At the pattern level, the developer could state that a particular class or method certainly does or does not play a particular role within a design pattern structure. The developer might detect a class that has been mistakenly attributed to a particular role, and might wish to formulate a constraint that states otherwise (i.e. ‘Class X is not an Adapter’ for example). The developer might also notice that a large number of false positive patterns are produced because the program analysis has failed to identify or rule-out a particular property of the source code.

All of the structural constraint types can be directly extracted by a straightforward source code analysis. However, these fail to capture the sequential constraints on the method calls, which can be so crucial to accurate pattern detection. Although the invoke constraint states that method $m_1$ invokes method $m_2$, this does not imply that $m_1$ must always occur before $m_2$. For example, in behavioural design patterns, the sequencing of the method calls can often provide valuable evidence about the accuracy of a pattern.

Figure 4.9: Hints level of detail
In the Command pattern, the \texttt{Invoker} class has to instantiate the \texttt{ConcreteCommand} class \textit{before} the \texttt{execute} method is called. Only after the \texttt{execute} method is called the \texttt{Receiver.action()} method is called. Such constraints are not commonly factored into the standard definitions of patterns and they are not generally considered when a program is analysed for pattern detection. They do, however, (as illustrated in Section 4.5.2) represent a nice example of how relatively simple hints can potentially lead to significant reductions in false positives.

As was alluded to previously, for behavioural patterns, the sequence in which methods are invoked is an important factor in determining whether a candidate set of classes implements the pattern or not. Since method invocations are especially difficult to accurately extract from source code (especially object-oriented code), this can be a key cause of false positive patterns. The use of hints enables the developer to add basic information to refine the static analysis results (e.g. method \texttt{m}_1 cannot call method \texttt{m}_2). However, such simple constraints are insufficient for relating the methods that are not directly connected by a method call, but might have an arbitrary set of intermediate method calls. For example, referring to the Command pattern in Figure 4.2, one might wish to state that \texttt{Receiver.action()} is invoked only after the invoking method in the \texttt{Invoker} class without caring about what the specific intermediate method calls may be.

To capture this more general sequential constraint, a further constraint \texttt{dominate} (\texttt{m}_1, \texttt{m}_2) is introduced stating that for any sequence of calls \texttt{m}_2 must be preceded at some point by \texttt{m}_1. This relation can be computed without additional inputs in the form of a static or a dynamic analysis. The call graph that relates all the different methods together can be constructed from the various \texttt{invoke} relations. Given such graph, the \texttt{dominate} relation can be determined by applying the standard dominator analysis (as commonly applied to Control Flow Graphs [6]). An example of the effectiveness of this additional constraint is discussed in Section 4.5.2 with respect to the running example in Section 4.1.
4.5.2 Example: The Iterative Feedback Process

As discussed in Section 3.4, there are several patterns with different intents that share the same structure. The flexibility of the proposed approach enables the developer to address these problems by gradually refining the set of constraints. This section illustrates how the proposed design pattern detection approach applies this iterative reduction process. The process of detecting the Command pattern with respect to the system shown in Figure 4.1 is illustrated below. Let us assume that the basic set of constraints representing the parsed source code are already there (as shown in Figure 4.7). Also, assume that the basic Command pattern definition is the one given in Figure 4.6. Let us also assume that the constraint-based definition of the Adapter pattern are as shown in Figure 4.10. Having been processed (along the lines of conventional constraint-based detection techniques), the developer is left with two possible candidate patterns as shown in Figure 4.11.

```
super-class(Target)
sub-class(Adapter)
inh(Adapter,Target)
aggre(Client, Target)
asso(Adapter,Adaptee)
```

**Figure 4.10:** Adapter pattern constraints

```
Command Pattern Candidates:
Command: Com
ConcreteCommand: C1, C2, C3
Invoker: AInvoker
Receiver: Receiver1, Receiver2

Adapter Pattern Candidates:
Client: AInvoker
Target: Com
Adapter: C1, C2, C3
Adaptee: Receiver1, Receiver2
```

**Figure 4.11:** Candidate Patterns
Chapter 4. GUIDING DESIGN PATTERN DETECTION WITH HINTS

Upon inspection, the developer might consider it odd that \( \text{C2} \) is considered as a \texttt{ConcreteCommand} role, even though it is an abstract. Gamma \textit{et al.} [54] book only specifies that the methods have to be concrete (overriding the methods in the superclass), but does not explicitly state that the class itself should not be abstract. This is an example of underspecification, which makes sense; some developers might want to consider it as an instance of \texttt{ConcreteCommand}, but this assumption is not applicable to the present case. The developer in the proposed approach adds a constraint (hint) to the pattern specification to make this explicit: \textit{The \texttt{ConcreteCommand} class must not be abstract.}

\[ \text{Not(abst-class(ConcreteCommand))} \]

As a consequence, \( \text{C2} \) is ruled out as a candidate, leaving only \( \text{C1} \) and \( \text{C3} \) as candidate classes. As a further hint, the developer might wish to add the constraint that, in the Command pattern, it must be the case that the \texttt{action()} method in the \texttt{Receiver} class is only ever invoked after the \texttt{execute()} method in the \texttt{Invoker} class. Again, this is not something that is explicitly specified by Gamma \textit{et al.} [54], and might not be universally desirable for pattern detection, as it might be considered in some cases to be too restrictive. However, for this case, the developer might want to enforce it in this case, which can be achieved by adding the \texttt{dominate} relation to the Command pattern specification:

\[ \text{dominate(invokerMethod, ReceiverMethod)} \]

As can be seen in Figure 4.1 (b), \texttt{ExtInvok.inv()} calls both methods in \( \text{C3} \), which means that \texttt{AInvoker.driver} does not dominate either of the methods in \( \text{Receiver2} \). Therefore, the constraint rules out \( \text{Receiver2} \) as a \texttt{Receiver} class and \( \text{C3} \) as a \texttt{ConcreteCommand} class. Accordingly, this restricts the Command pattern roles to the results shown in Figure 4.12.
Chapter 4. **GUIDING DESIGN PATTERN DETECTION WITH HINTS**

**Figure 4.12:** Command pattern candidate

Finally, the developer might decide, upon closer inspection of the source code and its identifiers, that Com is unlikely to be a Target class in the Adapter pattern. Accordingly, the developer might choose to add the following constraint:

\[ \text{Not(Com == target)} \]

Finally, this leaves the single Command pattern outlined above. In this example, the constraints have exclusively been concerned with tightening up the pattern definitions. However, they could just as well pertain to the system itself. If, for example, the developer knows that one methods always dominates another methods, this information can be readily added in exactly the same manner as the hints given above.

### 4.5.3 Using Templates to Guide Hint Selection

The requirement that the developer can readily phrase the arbitrary hints in the form of constraints may occasionally be unrealistic. Constructing proper constraints can require time, effort, and some knowledge of using constraint solvers, none of which are necessarily available. Addressing this problem is very much part of our ongoing work. However, one key effort we have made is inspired by a solution to a similar problem that arose in the context of software model checking.

In their work on model checking, Dwyer *et al.* observed that, despite their ability to verify properties about software and hardware systems, formalisms such as Linear Temporal Logic [97] were not used as widely as they ought to be. They addressed
this by proposing a small collection of LTL ‘templates’ [43] - small LTL expressions that captured properties about a system that one might commonly seek to verify.

A similar approach was adopted in this thesis to resolve the problem of capturing hints. Although it is, in principle, possible for the developer to provide arbitrary constraints, we have compiled a set of common ‘hint templates’ that capture the sort of feedback that one might wish to provide. These hint-templates were collected through the frequent use of the tool, and could easily be expanded. An example is shown in Figure 4.13, a selection of these hints can be found in Appendix C.

### 4.6 Hint Generation Process

To provide a feedback, the developer is required to provide their own ‘hints’ in the form of additional constraints. These hints might be simple statements of facts (‘Class X does not play the ‘Adapter’ role’). Hints can also be in the form of more intricate statements about the underlying behaviour or about the structure of the system. In the present implementation, hints are added to the bottom of the text file that contains the other constraints. Accordingly, each iteration amasses an increasing number of constraints, which ought to guide the constraint solver towards more accurate results.

The user can browse the source code analysis results, e.g. determine whether a particular class can take on a particular role in a design pattern, the user could

---

**Figure 4.13: Example - Hint**

```
Hint: Class x define a method m1 invoking at least one abstract method defined in the same class.
class(x);
has-method(x,m1);
has-method(x,m2);
abstract-method(m2);
invoke(m1,m2);
```
state that a particular class or method certainly does or does not play a particular role within a design pattern structure. Furthermore, the user might wish to carry out a cursory dynamic analysis to, for example, determine whether a particular method really does override another method. Dynamic information is gathered during the program execution by debugging the program manually. The user can use the Eclipse debugger [1] to run the program interactively (dynamic analysis) in order to observe the source code and the variables during the execution process.

Dynamic analysis gives the developers a deeper insight into the sequence of the method calls from one class to another, expression and variable values, and parameter passing. As a result, dynamic analysis helps to verify the pattern candidates by eliminating any class that does not participate and match the specified pattern behaviour. This makes it possible to trace and verify the executional and the behavioural information of the design pattern candidates. The dynamic information is collected during the program execution by debugging the source code manually. Breakpoints should be set for the pattern-related methods and method traces are recorded at run time.

As a result of involving the user knowledge to the heart of the design pattern detection process, the user understands the source code structure and behaviour through the static and the dynamic analyses, verifies the design candidates, identifies the missing playing roles, and iterates the detection process for several times. This process of automated analysis and user-driven input can be repeated in an iterative manner and, thus, gradually producing more accurate detection results.

### 4.6.1 Methodology

We anticipate that hints could be added in four phases. In each phase, the user is permitted to provide only the hints corresponding to a specific type of hint template.
1. **No Hints:** hints are not provided. Therefore, the pattern detection results corresponded to those that would be produced by a conventional constraint satisfaction (CSP-based) pattern detection approach.

2. **Structural Hints:** the user could add hints pertaining to the structure or the inheritance of the system. The user could identify a common relation between the design pattern roles. These hints give the user more information related to a single role that can be constituted with various design patterns. These hints are used to reduce the search space to a number of candidates that are fittingly similar to the targeted pattern structure.

3. **Sequential Hints:** the user could add hints pertaining to the method invocations. These hints provide information about the pattern behaviour of the program that might be impossible to verify from a finite number of program executions. Sequential hints enable the user to add basic sequential information to refine the static analysis results. These hints can refine these candidates by adding sequential information to produce a short list of candidate patterns that approximately similar to the targeted pattern behaviour.

4. **Ad-hoc Hints:** the user could add ad-hoc hints. These hints help the user to identify or rule-out a particular candidate, e.g. the user could state that a particular class or method certainly does or does not play a particular role within a design pattern structure. The user might detect a class that has been mistakenly attributed to a particular role, and might wish to formulate a constraint that states otherwise. The user might also notice that a large number of false positive patterns are produced because the program analysis has failed to identify or rule-out a particular property of the source code.
4.7 Implementation

A proof-of-concept tool was developed to demonstrate the feasibility and the validity of the proposed approach. This tool enables the application of the proposed hint driven pattern detection approach to Java programs. The parsing and source code analysis is carried out by the inFamix parser\(^1\). It parses and analyses the source code to produce a large text file in a FAMIX format [37]. This constitutes a large text file that represents the essential information regarding the system (e.g. classes, methods, attributes, etc.). Figure 4.14 below provides an overview of the FAMIX file core elements.

![Famix Core elements](image)

Figure 4.14: Famix Core elements [42]

In order to feed this information into a CSP to be solved by a constraint solver, a parser was developed to extract the relevant information from the FAMIX file as a set of facts and constraints in order to create the corresponding static models (architecture, and behaviour). As shown in Figure 4.15, the basic parser parses and analyses the .MSE file. In particular, it extracts all classes, methods, attributes, invocations, inheritance, accessor, and parameter. This parser is also used to encode the relevant information from the static models as a set of facts and constraints with respect to the object-oriented relationships.

\(^1\)http://www.intooitus.com/products/infamix
Chapter 4. GUIDING DESIGN PATTERN DETECTION WITH HINTS

java InFamix Parser.mse
Basic Parser

map

Deduce main program relations as functions and map into constraints to run the constraint solver (python)

CSP input

# abst_class
def abst_class(n1):
  conds = {
    (cl_id,)
    ...
    (cl_id),
    and_conds = (and(n1=a) for a in conds)
    return Or(*and_conds)

# Override
def override(h1, h2):
  conds = {
    (cl_id, me_id),
    (cl_id, me_id),
    and_conds = (and(h1=a, h2=b) for a,b in conds)
    return Or(*and_conds)

Figure 4.15: Source code parser
Chapter 4. GUIDING DESIGN PATTERN DETECTION WITH HINTS

The InFamix parser exports the source code into a .MSE file format, which is a generic format to describe the model that is similar to XML. This file contains eight entities including namespaces, packages, classes, methods, attributes, access, invocations, and inheritance. Each of these entities has a unique identifier, e.g. id:3 and its properties. The properties can be either primitive, like (name 'classA'), or they can point to another entity, like in the case of (parentPackage (ref:5)) which denotes that the parent package property of ClassA points to the package with (id:5). Using these entities, it is possible to refer to the entities using the ref: tag. The file structure is as follows:

```
(FAMIX.Class (id: 3)
 (parentPackage (ref: 5))
 (sourceAnchor (ref: 13911))
 (name 'classA')
 (isStub true)
 (isAbstract true)
 (isInterface true))
```

In detail, as shown in Figure 4.15, we used a set of hash tables to store the extracted elements (classes, methods, attribute, etc.). These tables store the extracted information by using a unique key, which is a FAMIX unique identifier as discussed above. This key is then used as the index at which the extracted data associated with the key is stored. With respect to the above example: the index id:3 associated with the name:classA, Is-abstract:True, Is-Interface:True. The inheritance relationship between a one subclass and a one superclass can be represented using two fields sub:subclass, sup:superclass. The method hash table stores the associated fields with each method including method name, parent, declared, and method types including Is-Abstract, Is-Protected, and Is-Final.

As mentioned above, two different models (architecture and behaviour) are used to represent and to analyse the static data of the Java programs as well as to
describe the program elements. The architecture model contains all the classes, the methods, and the attributes. The behaviour model contains the method invocations list. These two models are used to create the corresponding model with respect to object-oriented relationships. In fact, the need for the object-oriented parser is due to the missing capability of the current source code analysis tools necessary to create an intermediate representation with respect to the object-oriented relationships.

The object-oriented relationships (i.e. the associations) are essential characteristics that are used to describe and implement a number of design patterns [54]. Current software modelling tools (i.e. UML tools) are able to differentiate these relations in forward engineering. However, unfortunately, they are still incapable to detect how these relations are implemented in the source code. Thus, we utilised the extracted source code elements (architecture and behaviour model) to identify these relationships as described below:

- The inheritance relationship is explicitly derived from the architecture model where the classes and the interfaces declare the classes or the interfaces they extend or implement. For example: in each FAMIX class contains super/sub types relationships. Thus, we represented these two types in the inheritance hash table as sub:subclass, sup:superclass.
- The association and the aggregation are not explicitly declared in the architecture model. Thus, the behaviour model (method invocation list) was utilised to approximately identify these relationships. For example, the association between ClassA and ClassB defines the ability of an instance of ClassA to invoke an instance of ClassB. On the other hand, the aggregation relationship between ClassA and ClassB exists if ClassA (the whole) contains instances of ClassB (the part). ClassA must define a an array field of the type of ClassB. Then, the instances of ClassA invoke the instances of ClassB.

The design patterns (meta-model) are provided as abstract descriptions as described in Gamma et al. [54]. A set of constraints were built that show the abstract model
for the design pattern. These constraints incorporate the classes, methods and the relationships between classes such as inheritance, association and method delegations. In particular, we subsequently developed a small design pattern repository. To do this, we took a set of behavioural patterns from Gamma et al.’s book [54] and encoded them using again the same syntax as was used in Section 4.4. The chosen patterns were the Template Method, Observer, State/Strategy, and Command (See Appendix B). As a constraint solver we selected the Z3 constraint solver [36]. Specifically, we used the Z3Py extension, which can directly produce solutions to text files of the format used above.

4.8 Related Work

The process used in the proposed work to acquire a set of constraints is related to the work by Bessiere et al. on query-driven constraint acquisition [22]. Their CONACQ approach addresses a similar problem, in the sense that they are attempting to reason with a constraint solver about an incomplete system of constraints, and seek further input from a user. However, whereas they formulate specific constraints as queries to the user, the approach used here presents the user with a full hypothesis solution (the proposed set of design patterns), and expects them to offer further constraints (using the hypothesised patterns as a basis).

In Machine Learning terminology, whereas they pose ‘membership-queries’ (requiring a simple ‘yes’ or ‘no’ as an answer), the approach proposed here uses ‘equivalence queries’ [12], requiring the user to provide at least one counter-example. There would certainly be a scope for the use of membership queries in this work (i.e. do modules x and y belong together?). However, in the software maintenance usage scenario, where the maintainer does not necessarily have the patience to answer what could be thousands of questions, it seems more practical to rely on their intuition to select those counter examples that seem most obvious.
Within software engineering, this general approach of gradually acquiring constraints has previously been successfully applied to the problem of software re-modularisation. Hall et al. [67] applied the same high-level framework to acquire small facts from the developer about which classes belong together in the same package. The relevant findings from this work are that (at least in the context of re-modularisation) it is possible to substantially improve the accuracy of a solution with a relatively small amount of manual input.

4.9 Summary

This chapter proposed an approach that enables the user to feed small ‘hints’ into the detection process. This is different to some of the existing tools that permit a limited degree of user-interaction. The proposed approach enables the developer to be much more expressive, whilst remaining easy to use. The approach is intended to act as a complementary ‘layer’ to enable for additional feedbacks from the developer. The proposed approach was founded on the observation that, in order to produce reliably accurate results, it is necessary to actively involve the developer in the pattern detection process. The developers must be provided with as much opportunity as possible to feed their knowledge about the system structure, its behaviour, and the possible implementation. To address this need, this approach has proposed a procedure that can complement the existing constraint-based pattern detection techniques. This procedure enables the user to supply their knowledge in the form of ‘hints’ as small facts about the source code or the pattern implementations that can enhance the results produced by the constraint solver. An application of the users’ feedbacks within an iterative detection process was proposed in this chapter. In this application, the user was able to gradually enhances the detection results. The proposed approach seeks to enable the user to feed his/her knowledge interactively into the detection process, and be able to decide, to stop the detection process, or to iterate it further.
Chapter 5

EXPERIMENTS

This chapter presents experiments of the capability of the proposed approach in integrating the user knowledge into the design pattern detection process. Section 5.1 introduces the case study research question. Section 5.2 outlines the methodology adopted in this evaluation. Section 5.3 discusses the acquired detection results. Section 5.4 presents the evaluation of the validity of the acquired detection results by comparing the results. Section 5.5 discusses the reasons for the false detection results. Section 5.6 covers the threats to validity. Finally, a summary is given in Section 5.7.

5.1 Case Study Research Question

The approach was applied to a selection of openly available software systems. The investigated software systems have all been used to evaluate other design pattern detection approaches. Three selected systems were presented to a user, i.e. the researcher. The user was asked to use our tool to identify patterns by supplying hints in a structured iterative process. After each iteration of hints, the accuracy was measured, and the user was given the opportunity to continue. This process
was repeated over four iterations. This case study seeks to establish the impact of the user knowledge on the design pattern detection process. The specific research question was:

**RQ 1**: To what extent does the use of hints lead to an increase in the accuracy of the design pattern detection results?

The systems selected for the evaluation process are JHotDraw [2], JUnit [3], and QuickUML [5]. These systems were selected for the evaluation of the proposed pattern detection approach as they are mature, openly-available frameworks, built with design-patterns in mind, and have been used as the basis for the evaluation of several existing design pattern detection techniques (e.g. [117], [60], [33]).

## 5.2 Methodology

- **Hint Selection** The fact that the proposed approach ultimately revolves around the input from the developer makes it challenging to evaluate. A skilled developer could choose a small selection of highly useful hints, which would be much more effective than those selected by an inexperienced developer, who might select irrelevant hints that have less of an effect. To limit such sources of bias, the hint process was systematised as much as possible. The hint selection process was fixed to four iterations. For each iteration, the developer was permitted to provide only the hints corresponding to a specific type of hint template. In the first iteration, hints are not provided. Therefore, the results corresponded to those that would be produced by a conventional constraint satisfaction (CSP-based) pattern detection approach. In the second iteration, the developer could add hints pertaining to the structure or the inheritance of the system. In the third iteration, the developer could add hints pertaining to
the method invocations. Finally, in the fourth iteration the developer could add ad-hoc hints.

• **Sequence of Hints** The sequence in which the hints were added in phases was fixed, to reflect the hint templates. Hints had to be chosen according to the methodology set out in Section 4.6.1. Firstly, as mentioned above, hints are not provided. Then, the user starts by adding structural hints, sequential hints, and finally ad-hoc hints. The structurally matched design pattern (candidate instances) in the source code are identified based on the results of the structural hints phase. The main goal of the structural hints is to reduce the search space to a number of candidates that are fittingly similar to the targeted pattern structure. Then, by adding the sequential hints, the user can refine these candidates by adding sequential information to produce a short list of candidate patterns that approximately similar to the targeted pattern behaviour. Finally, the user adds the ad-hoc hints to identify or rule-out a particular candidate. For example: the user could state that a particular class or method certainly does or does not play a particular role within a design pattern structure. The user might detect a class that has been mistakenly attributed to a particular role, and might wish to formulate a constraint that states otherwise. The user checks whether the detected whether the pattern candidates are actually implemented in the source code or not. This step is performed by browsing the source code and the user might wish to carry out a cursory dynamic analysis (e.g. with the help of a debugger).

• **Evaluation** To assess the accuracy of the acquired results at each stage, the ‘Role-based’ pattern evaluation approach proposed by Petterson et al. [95] was adapted. This approach adapts the classical precision recall metric [118]. Instead of focussing purely on the exact detection of an entire pattern instance, they assumed that the evaluation should be carried out in terms of the proportion of the correctly assigned roles per pattern. This enables us to account for the partial matches, and allows for a more granular comparison.
The role-based pattern evaluation is also used in the results of the approaches in [65], and [61]. The results were computed with respect to five well-known behavioural design patterns [54] that are problematic for the conventional pattern techniques: Template Method, State, Strategy, Observer, and Command. The lack of standard benchmarks of design pattern instances makes the evaluation process challenging [95]. We manually analysed the acquired results to assess the accuracy of the detected design pattern instances. We used the available documentation of the systems under analysis to determine the implemented patterns. However, since only the popular design patterns were mentioned in the documentation, we manually investigated the design of the frameworks of each system from the source code in order to identify any additional implemented patterns.

- **Validity** One (well-established) limitation of conducting research in design patterns is the lack of a “golden standard” by which to evaluate the retrieved patterns [95]. Systems are poorly documented, and often fail to fully document their patterns. Accordingly, when it comes to evaluation, one has to inevitably use a degree of human judgement to determine whether a particular pattern is accurate or not. The approaches in the literature were evaluated manually based on the evaluator judgement. However, same of the design pattern instance can be considered as a true positive and considered as a false negative by another evaluator. This invariably introduces a risk of bias. To attenuate this risk, a two-pronged evaluation approach was followed in this case study. On one hand, as it has already been discussed in the previous paragraph, a manual analysis was conducted to assess the veracity of the retrieved design patterns. On the other hand, the only available peer reviewed repository of design patterns P-MARt [59] was used as a comparison baseline.
5.2.1 Subject Systems

The selected systems are JHotDraw [2], JUnit [3], and QuickUML [5]. Table 5.1 illustrates the number of classes, the number of methods, the number of interfaces, the applied patterns, and the distribution of patterns per system. JHotDraw 5.1 [2] is a drawing editor application framework usually used for the creation of graphical editing applications. JHotDraw was selected for the evaluation process because it was built using well-known patterns. JHotDraw was developed to apply the design patterns concepts. JUnit 3.7 [3] is a unit testing framework for the Java programming language that is used to write and run repeatable tests. It is used by the developer who implements unit tests in Java. The design of JUnit is presented in a style of patterns [51]. It is developed to ease the implementation and running of the unit tests for Java applications. QuickUML 2001 [5] is an editor tool usually used to create and manipulate the class diagrams based on a small and simple subset of the standard UML notation. It is an object-oriented design tool that supports the design of a core set of UML models and has advanced features including the design namespaces for project organizations.

<table>
<thead>
<tr>
<th>System</th>
<th>Number of classes</th>
<th>Number of Interfaces</th>
<th>Number of Methods</th>
<th>Applied Patterns</th>
<th>Distribution of Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>JHotDraw 5.1</td>
<td>136</td>
<td>19</td>
<td>1393</td>
<td>Adapter</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Command</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Composite</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Decorator</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Factory Method</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Observer</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Prototype</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Singleton</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>State</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Strategy</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Template Method</td>
<td>2</td>
</tr>
<tr>
<td>JUnit 3.7</td>
<td>69</td>
<td>10</td>
<td>856</td>
<td>Composite</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Decorator</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Iterator</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Observer</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Singleton</td>
<td>2</td>
</tr>
<tr>
<td>QuickUML 2001</td>
<td>142</td>
<td>13</td>
<td>1264</td>
<td>Abstract Factory</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Builder</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Command</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Composite</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Observer</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 5.1: Subject systems
Chapter 5. EXPERIMENTS

5.3 Results & Discussion

The approach is evaluated by applying it to the detect design patterns in the above-mentioned three open-source systems. The detected patterns instances are examined and manually considered as true or false positives. The accuracy is calculated in terms of precision, recall, and F-score measures. The precision measures the degree of how much of the detected patterns are implemented (True Positives). On the other hand, recall measures how much of the implemented patterns are detected. F-score measures the combination of precision and recall.

Tables 5.2 and 5.3 show the evaluation results of the four phases where the accuracy was determined by manual validation of the detected patterns. TP refers to the number of true positives, FP refers to the number of false positives, FN refer to the number of false negatives and P, R, and $F_w$ refer respectively to the precision, the recall and the F-Score derived from these numbers, using Petterson’s metrics [95]. The tables should be read from left to right. Different types of hints were applied in phases, which yielded the final results on the right. The results indicate that the presentation of hints tends to increase the accuracy of the results. However, the precision values show that any increases in the accuracy are not necessarily monotonic, there can be substantial fluctuations.

Table 5.2 presents the results obtained from phase 1 and phase 2. In phase 1, no hints are presented in the detection process. The result are obtained based on the conventional constraint satisfaction pattern detection approaches. From the result of phase 1, there is an apparent that the low rate of true positives (i.e. Template Method in QuickUML). There is however, a high number of false positive (the case of Command and State/Strategy in JHotDraw). In phase 2, the user adds hints pertaining to the structure or the inheritance of the system domain.
Table 5.2: No hints/structural hints phase detection results

<table>
<thead>
<tr>
<th>Pattern</th>
<th>System</th>
<th>TP</th>
<th>FP</th>
<th>FN</th>
<th>P</th>
<th>R</th>
<th>Fw</th>
</tr>
</thead>
<tbody>
<tr>
<td>Template Method</td>
<td>JHotDraw</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>40%</td>
<td>40%</td>
<td>40%</td>
</tr>
<tr>
<td></td>
<td>JUnit</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>8%</td>
<td>50%</td>
<td>31%</td>
</tr>
<tr>
<td></td>
<td>QuickUML</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>State/Strategy</td>
<td>JHotDraw</td>
<td>0</td>
<td>37</td>
<td>6</td>
<td>0%</td>
<td>NA</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>JUnit</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>0%</td>
<td>NA</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>QuickUML</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>NA</td>
<td>NA</td>
<td>0</td>
</tr>
<tr>
<td>Observer</td>
<td>JHotDraw</td>
<td>2</td>
<td>9</td>
<td>2</td>
<td>18%</td>
<td>50%</td>
<td>42%</td>
</tr>
<tr>
<td></td>
<td>JUnit</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>0%</td>
<td>NA</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>QuickUML</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0%</td>
<td>NA</td>
<td>1</td>
</tr>
<tr>
<td>Command</td>
<td>JHotDraw</td>
<td>1</td>
<td>56</td>
<td>0</td>
<td>2%</td>
<td>100%</td>
<td>14%</td>
</tr>
<tr>
<td></td>
<td>JUnit</td>
<td>0</td>
<td>8</td>
<td>1</td>
<td>0%</td>
<td>NA</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>QuickUML</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0%</td>
<td>NA</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5.3: Sequential hints/Ad-hoc hints phase detection results

<table>
<thead>
<tr>
<th>Pattern</th>
<th>System</th>
<th>TP</th>
<th>FP</th>
<th>FN</th>
<th>P</th>
<th>R</th>
<th>Fw</th>
</tr>
</thead>
<tbody>
<tr>
<td>Template Method</td>
<td>JHotDraw</td>
<td>5</td>
<td>12</td>
<td>0</td>
<td>29%</td>
<td>100%</td>
<td>79%</td>
</tr>
<tr>
<td></td>
<td>JUnit</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>33%</td>
<td>100%</td>
<td>82%</td>
</tr>
<tr>
<td></td>
<td>QuickUML</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>State/Strategy</td>
<td>JHotDraw</td>
<td>3</td>
<td>19</td>
<td>3</td>
<td>14%</td>
<td>50%</td>
<td>38%</td>
</tr>
<tr>
<td></td>
<td>JUnit</td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>67%</td>
<td>100%</td>
<td>95%</td>
</tr>
<tr>
<td></td>
<td>QuickUML</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0%</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Observer</td>
<td>JHotDraw</td>
<td>4</td>
<td>30</td>
<td>0</td>
<td>12%</td>
<td>100%</td>
<td>54%</td>
</tr>
<tr>
<td></td>
<td>JUnit</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>50%</td>
<td>100%</td>
<td>90%</td>
</tr>
<tr>
<td></td>
<td>QuickUML</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>33%</td>
<td>100%</td>
<td>82%</td>
</tr>
<tr>
<td>Command</td>
<td>JHotDraw</td>
<td>1</td>
<td>11</td>
<td>0</td>
<td>8%</td>
<td>100%</td>
<td>45%</td>
</tr>
<tr>
<td></td>
<td>JUnit</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>50%</td>
<td>100%</td>
<td>90%</td>
</tr>
<tr>
<td></td>
<td>QuickUML</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>20%</td>
<td>100%</td>
<td>69%</td>
</tr>
</tbody>
</table>
Table 5.3 presents the results obtained from phase 3 and 4. In phase 3, the user could add hints pertaining the method invocations. The results from this phase show an increase of true positive values comparing to the precision values in phase 2 (i.e. State/strategy in JUnit). In addition, the results of phase 3 show a decrease of false positive (i.e. Template method in JHotDraw). Finally, in phase 4, the user could add ad-hoc hints. These hints depend on the user level of knowledge of the pattern and the domain.

The precision and the recall values in these two tables can be compared. Precision values show that any increases in the accuracy are not monotonic. For example, the precision values of the Template method in JHotDraw are 40 % in phase 1, 30 % in phase 2, 29 % in phase 3, and 100 % in phase 4. Recall values show any increases in the accuracy is monotonic. For example, the recall values of the Observer pattern in JHotDraw are 50 % in phase 1, 75 % in phase 2, and 100 % in phase 3 and 4. The substantial fluctuations in precision values between phase 1 and 2 due to breaking of the strict detection rules in phase 2 implies detection more patterns candidates. These candidates can either increase the true positive or increase the false positive, but, they will definitely decrease the false negatives. Consequently, increasing or decreasing the true positives and false positives causes the fluctuations in precision values. However, decreasing the false negatives causes an increase of recall values in a monotonic manner.

Hints do not only rule out false positives (thus increasing precision), but can also rule in false negatives. They add information about the system that may have been omitted by the preliminary program analysis and, therefore, enable the constraint solver to highlight the presence of additional design patterns. This can lead to a decrease in the precision but a corresponding increase in recall. In one case the use of hints was detrimental to accuracy. For the State/Strategy pattern in QuickUML, the addition of hints in phases 2 and 3 only led to the detection of false positives. Also, looking at the Observer pattern for JHotDraw, the hints added in phase 1 added 13 false positives, whilst only adding a single true positive.
At first glance, the jumps to 100% precision and recall for the ad-hoc selection of hints might appear suspicious. For the Command pattern in JHotDraw for example, the combination of structural and sequential hints only achieves 8% precision, whereas the inclusion of a selection of ad-hoc hints suddenly increases precision to 100% without compromising recall. First of all, it is important to bear in mind that the ultimate number of true positives patterns is very low (a maximum of 5 occurrences). Secondly, the reason for low precision is always due to a large number of false positives (e.g. 38 false positives for the Command pattern). This highlights a particular strength of the use of hints, it is often trivial to rule out a large number of false positives with a single, well-placed hint (e.g. ‘Class X is not a Command’). This ultimately explains the large leaps in precision at the end; it is because the developer inspects each pattern, and rules out the unlikely roles with a single hint.

5.4 Evaluation Against P-MARt Repository

As discussed earlier in Section 5.3, the lack of standard benchmarks makes the evaluation and the validation a difficult task. The approaches in the literature were evaluated manually based on the evaluator judgement. However, same of the design pattern instance can be considered as a true positive and considered as a false negative by another evaluator. Consequently, the only peer reviewed patterns repository is used as a comparison baseline. To attenuate the risk of the acquired results being biased, and to assess the validity of the proposed approach, we decided to re-run our experiments using the P-MARt repository [59]. Table 5.4 shows the comparison based on P-MARt, it shows the number of the pattern instances specified by P-MARt repository, in addition to the number pattern instances detected by our approach. This table also shows the number of shared instances with P-MARt repository.
### Table 5.4: Results comparison based on P-MARt

<table>
<thead>
<tr>
<th>Pattern: Role</th>
<th>JHotDraw</th>
<th>JUnit</th>
<th>QuickUML</th>
</tr>
</thead>
<tbody>
<tr>
<td>Template Method: Abstract Class</td>
<td>TP 2, FP 0, FN 0</td>
<td>TP 2, FP 0, FN 0</td>
<td>TP 2, FP 0, FN 0</td>
</tr>
<tr>
<td>Observer: Subject</td>
<td>TP 2, FP 0, FN 0</td>
<td>TP 2, FP 0, FN 0</td>
<td>TP 2, FP 0, FN 0</td>
</tr>
<tr>
<td>Observer: Observer</td>
<td>TP 0, FP 0, FN 0</td>
<td>TP 0, FP 0, FN 0</td>
<td>TP 0, FP 0, FN 0</td>
</tr>
<tr>
<td>State/Strategy: Context</td>
<td>TP 0, FP 0, FN 0</td>
<td>TP 0, FP 0, FN 0</td>
<td>TP 0, FP 0, FN 0</td>
</tr>
</tbody>
</table>

### Table 5.5: Results accuracy comparing to P-MARt

<table>
<thead>
<tr>
<th>Pattern: Role</th>
<th>JHotDraw</th>
<th>JUnit</th>
<th>QuickUML</th>
</tr>
</thead>
<tbody>
<tr>
<td>Template Method: Abstract Class</td>
<td>TP 2, FP 0, FN 0</td>
<td>TP 2, FP 0, FN 0</td>
<td>TP 2, FP 0, FN 0</td>
</tr>
<tr>
<td>Observer: Subject</td>
<td>TP 2, FP 0, FN 0</td>
<td>TP 2, FP 0, FN 0</td>
<td>TP 2, FP 0, FN 0</td>
</tr>
<tr>
<td>Observer: Observer</td>
<td>TP 0, FP 0, FN 0</td>
<td>TP 0, FP 0, FN 0</td>
<td>TP 0, FP 0, FN 0</td>
</tr>
<tr>
<td>State/Strategy: Context</td>
<td>TP 0, FP 0, FN 0</td>
<td>TP 0, FP 0, FN 0</td>
<td>TP 0, FP 0, FN 0</td>
</tr>
</tbody>
</table>

Chapter 5: EXPERIMENTS
Table 5.4 illustrates the key roles playing as reported in the P-MARt repository. This table compares our tools against the P-MARt repository based on how many instances we share with P-MARt; this can represent a useful indication to assess the validity of the obtained results. Table 5.4 also shows the obtained results in the three systems. It can be seen that in JHotDraw, for the TemplateMethod:AbstractClass role we detected five roles, and we shared the two roles as reported. The approach detected Observer:Subject role as reported by P-MARt. For the Observer:Observer we detected four instances but we did not share any of the reported roles.

Concerning the State/Strategy:Context the approach shared two instances and missed sixteen reported roles. On the other hand, for State/Strategy:Context we shared three roles out of five as reported. For the three key role playing in Command pattern, we shared the exact instance of Command:Command role, one out four for both Command:Invoker and Command:Receiver. In JUnit, the reported key role playing in P-MARt are only two Observer:Subject and three Observer:Observer. The approach shared one instance of both roles. Finally, in QuickUML the reported roles are one Observer:Subject and one Observer:Observer, we shared the exact roles as reported. On the other hand, three of the Command:Command roles in QuickUML were missed, and we only shared one instance. Overall, our approach is a viable for sharing key role playing classes as reported by P-MARt.

Table 5.5 shows the numbers of true positives TP, false positives FP, false negatives FN, the precision P, the recall R, and the F-score Fw values for each design pattern instance/role compared to the roles reported in the P-MARt repository. Besides, Tables 5.6, 5.7, and 5.8 show the actual roles detected in the three selected systems, along with the actual roles specified in P-MARt repository. As can be seen from these tables, there is a significant overlap between the patterns detected by our approach and those reported in the P-MARt repository. Although the performance of the detection results is highly accurate for QuickUML and JUnit, several roles remain undetected by the approach for JHotDraw. This is especially the case for the State/Strategy pattern. The inaccuracy comes down to an overly strict set of
constraints that were used to define the base-pattern, and the subsequent failure of hints to loosen this definition to a sufficient degree.

In JHotDraw, the results show that we are able to detect Template method pattern instances with good recall whilst having some false positives that affect the precision value. In particular, three of the detected instances of AbstractClass role were identified as false positives which affect the precision (40%). However, a subsequent manual analysis of the source code and the documentation, the analysis shows that these instances were in fact true positive instances. This is due to the fact that the Template pattern can be implemented with a variety of code implementation styles. For example, class LineConnection in CH.ifa.figures package plays the AbstractClass role. This is also the case for class LineDecoration in the same package.

The proposed approach works well in detecting the Observer:Subject role. The results show 100% precision and recall since the approach was capable to detect all the Observer:Subject roles specified in P-MARt. However, with the Observer:Observer role the approach was unable to match any reported role in the P-MARt repository. The results indicate that the four detected roles were considered as false positive which affects the precision. In addition, the approach missed two roles as reported in P-MARt repository, which are considered as false negatives and, hence, affect the recall of the detection of the Observer:Observer role.

Regarding the detection of the State/Strategy pattern in JHotDraw, there are too many missing role instances (16 instances) for the State/Strategy:Context role, which negatively affect the recall (11%). It is possible that these results are due to the highly dynamic behaviour of the State/Strategy pattern. In particular, the relation between the pattern roles depends mainly on the value of the field at the run time. Thus, the variations of the method delegation implementations between the State/Strategy:Context and State/Strategy:State/Strategy seems to be the main reason behind the undiscovered context role.
Concerning the Command pattern, the approach detected the Command:Command role with 100% precision and recall (CH.if.draw.util.Command). However, there are some missing Command:Invoker and Command:Receiver roles which affect the recall rate (25%). The reason is due to the multiple inheritance constraints; some roles do not belong to any inheritance hierarchy with the Command:Command role. A possible explanation for this might be that with adding the constraints at different level of inheritance hierarchy, we can detect these roles. Whereas the detection of the common interface (a class named CH.if.draw.util.Command) can help to identify the other Command:Invoker and Command:Receiver roles, where all command inherits from the common abstract interface.

In JUnit, the proposed approach was able to detect the Observer:Subject role with a 100% precision and a 50% recall. This is because the approach missed one of the reported roles. The P-MARt reported only two instances of the observer:Subject role and three instances of the Observer:Observer role. However, with Observer:Observer role, the proposed approach detected two instances, one was considered as a true positive (junit.framework.TestListener) and the other was considered as a false positive (junit.framework.Test), which affects the acquired precision by 50%. The proposed approach also missed two of the reported roles, which affect the recall 33% (java.awt.event.MouseListener and java.awt.event.KeyListener).

In QuickUML, the results show 100% in both, the precision and the recall of the Observer:Subject and the Observer:Observer roles. However, with the Command pattern, the proposed approach detected one instance of the Command:Invoker role that was considered false positive which affect the precision. In addition, our approach missed one instance reported in P-MARt (false negatives) which affect the recall. For the Command:Command role, our approach detected one instance (uml.ui.ExportAction) out of the four instances reported in P-MARt. This gives 100% precision but with low recall (25%) since the proposed approach has missed three reported roles.
### Table 5.6: Detected/shared pattern instances with P-MARt in JHotDraw

<table>
<thead>
<tr>
<th>Subject</th>
<th>Observer</th>
<th>Observer</th>
</tr>
</thead>
<tbody>
<tr>
<td>CH.ifa.draw.framework.Figure</td>
<td>CH.ifa.draw.standard.CompositeFigure</td>
<td>CH.ifa.draw.framework.Drawing</td>
</tr>
<tr>
<td>CH.ifa.draw.framework.Figure</td>
<td>CH.ifa.draw.standard.DecoratorFigure</td>
<td>CH.ifa.draw.framework.Drawing</td>
</tr>
<tr>
<td>CH.ifa.draw.framework.Figure</td>
<td>CH.ifa.draw.framework.Figure</td>
<td>CH.ifa.draw.framework.Drawing</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Invoier</th>
<th>Command</th>
<th>Receiver</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Command</th>
<th>Receiver</th>
</tr>
</thead>
<tbody>
<tr>
<td>CH.ifa.draw.framework.ConnectionFigure</td>
<td>CH.ifa.draw.framework.Connector</td>
</tr>
<tr>
<td>CH.ifa.draw.framework.DrawingView</td>
<td>CH.ifa.draw.framework.PointConstraint</td>
</tr>
<tr>
<td>CH.ifa.draw.framework.DrawingView</td>
<td>CH.ifa.draw.framework.Tool</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Context</th>
<th>$/$</th>
</tr>
</thead>
<tbody>
<tr>
<td>CH.ifa.draw.framework.ConnectionFigure</td>
<td>CH.ifa.draw.framework.DrawingView</td>
</tr>
<tr>
<td>CH.ifa.draw.framework.ConnectionFigure</td>
<td>CH.ifa.draw.framework.PointConstraint</td>
</tr>
<tr>
<td>CH.ifa.draw.framework.ConnectionFigure</td>
<td>CH.ifa.draw.framework.Tool</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Command</th>
<th>Observer</th>
</tr>
</thead>
<tbody>
<tr>
<td>CH.ifa.draw.framework.ConnectionFigure</td>
<td>CH.ifa.draw.framework.Connector</td>
</tr>
<tr>
<td>CH.ifa.draw.framework.ConnectionFigure</td>
<td>CH.ifa.draw.framework.Connector</td>
</tr>
<tr>
<td>CH.ifa.draw.framework.ConnectionFigure</td>
<td>CH.ifa.draw.framework.Connector</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$/$</th>
<th>Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>CH.ifa.draw.framework.Painter</td>
<td>CH.ifa.draw.framework.Painter</td>
</tr>
<tr>
<td>CH.ifa.draw.framework.Painter</td>
<td>CH.ifa.draw.framework.Painter</td>
</tr>
<tr>
<td>CH.ifa.draw.framework.Painter</td>
<td>CH.ifa.draw.framework.Painter</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Command</th>
<th>Observer</th>
</tr>
</thead>
<tbody>
<tr>
<td>CH.ifa.draw.framework.Painter</td>
<td>CH.ifa.draw.framework.Painter</td>
</tr>
<tr>
<td>CH.ifa.draw.framework.Painter</td>
<td>CH.ifa.draw.framework.Painter</td>
</tr>
<tr>
<td>CH.ifa.draw.framework.Painter</td>
<td>CH.ifa.draw.framework.Painter</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$/$</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>CH.ifa.draw.framework.Painter</td>
<td>CH.ifa.draw.framework.Painter</td>
</tr>
<tr>
<td>CH.ifa.draw.framework.Painter</td>
<td>CH.ifa.draw.framework.Painter</td>
</tr>
<tr>
<td>CH.ifa.draw.framework.Painter</td>
<td>CH.ifa.draw.framework.Painter</td>
</tr>
</tbody>
</table>
### Table 5.7: Detected/shared pattern instances with P-MARt in JUnit

<table>
<thead>
<tr>
<th>Subject</th>
<th>Observer</th>
<th>Subject</th>
<th>Observer</th>
<th>Subject</th>
<th>Observer</th>
</tr>
</thead>
</table>

### Table 5.8: Detected/shared pattern instances with P-MARt in QuickUML

<table>
<thead>
<tr>
<th>Subject</th>
<th>Observer</th>
<th>Subject</th>
<th>Observer</th>
<th>Subject</th>
<th>Observer</th>
</tr>
</thead>
</table>
5.5 Reasons for False Detection

Overall, it can be seen that the overlap between the acquired results and the P-MARt repository is encouraging. The results indicate that the use of hints in an iterative manner shows a significant improvement in the detection results, not only reducing false positive but also detecting more patterns, which reduces false negatives. However, there are several possible explanations for false positives and false negatives detection results, the major reasons are as follows:

5.5.1 Design Pattern Variants

Design pattern variants are one of the main problems that affect the accuracy of the design pattern detection approaches [102]. Some patterns can be implemented in a variety of different ways, and it is not possible to define all the patterns implementation variants. The proposed approach provided a flexibility in customising (breaking) the detection constraints by the user. However, there are still some implementation variants that produce some false negatives. A future research should therefore concentrate on investigating possible variants of the design patterns and on creating a catalogue of all common pattern implementations with proper hints types. There will, of course, never be a complete catalogue, but our approach provides a facility of easy customization of the detection constraints by the user.

5.5.2 Dynamic Data Flow

The lack of the dynamic (runtime) data flow analysis affects the detection of the patterns that rely on method delegations at runtime. Dynamic data flow analysis is the analysis of the data sequence of actions during the program execution [26]. Method delegation is the concept of passing tasks between methods during the program execution [108]. For example: a given method invokes the other methods
at the program runtime. Method delegation is the most important property used to implement a number of design patterns [54]. In particular, in object-oriented programming, delegation is used to describe the case where one object allocates a task to another object (the delegate). However, the identification of the method delegation information is challenging. The reason due the fact that the delegation can be implemented by different ways in different source systems [103]. Therefore, providing hints at the method delegation needs additional analysis at the program execution time. These hints rely on the dynamic analysis of the execution traces captured at program runtime. Further work is needed focusing on providing hints based on the method delegation information. This would be interesting since these hints will establish a greater degree of accuracy to handle the challenge of detecting patterns that rely on the method delegation information.

5.5.3 Parser Capabilities

Another possible explanation is the source code analysis and parser limitations. The parser limitations rise due to missing capability of the current source code analysis tools to create an intermediate representation with respect to the object-oriented relationships. The object-oriented relationships (i.e the associations) are essential characteristics that are used to describe and implement a number of design patterns [54]. Current software modelling tools (i.e UML tools) are able to differentiate these relations in forward engineering. However, unfortunately, they are still incapable to detect how these relations are implemented in the source code. Thus, we utilised the extracted source code elements (architecture and behaviour model) to identify these relationships. A parser was developed to extract the relevant information from the source code to create the corresponding models (architecture, and behaviour). It was also used to create the relevant information as a set of facts and constraints with respect to the object-oriented relationships. For example, the inheritance relationship is explicitly derived from the architecture model where the classes and
the interfaces declare the classes or the interfaces they extend or implement. On the other hand, the association relationships are not explicitly declared in the architecture model. Thus, the behaviour model (method invocation list) was utilised to identify approximately these relationships.

The association relationships are difficult to identify because their lack of precise definitions [57]. They require the dynamic information as they do not appear in the source code explicitly. However, the accuracy for identifying these relationships may differ depending on the program execution. The identification of the association relationships is subject to the common limitations of dynamic data flow analyses. Therefore, there are, of course, an entire common program features that the source code parser is unlikely ever to be able to detect. Thus, some roles of design patterns could be missed in the recovered instances. A robust and flexible parser is an important issue that helps in identifying the main feature of the patterns for a more precise detection. The limitation of the source code parser requires additional effort. The development of a robust parser requires an intensive effort and time. In the future, it is recommended to enhance the source code parser with more capabilities for the detection of the object-oriented relationships.

5.6 Threats to validity

5.6.1 Internal Validity

This validity threats concerns with the relation between the observed improvements in the detection results and the selection of hints; The hints used for the evaluation were selected by the principal author. The selection of hints is critical to the efficacy of the approach. The fact that the hints were selected by a single author does raise the possibility that the selection was biased by the author’s extensive familiarity with the proposed technique. The risk of bias was attenuated by the fact that the selection
of hints was systematised by using a set of hint-templates. Furthermore, Chapter 5 presents a controlled user study that involves a broader range of participants. This study more carefully explores the selection of hints with a view to determining which hints are especially useful.

5.6.2 External Validity

The subject systems are not representative of the broad family of software systems. Two of them are GUI-driven diagramming software systems. And there are only three systems in total. This makes it difficult to generalise from these results to broader classes and sized systems. This leads to the threat that the conclusions drawn from the acquired validation results do not generalise to a sufficiently broad range of systems. Further evaluation with different systems may reduce this threat. Furthermore, a user study in Chapter 6 may give an insight onto how the users are able to determine hints that can feed the design pattern detection process. The fact that the investigated systems are relatively diverse in purpose does attenuate this threat to a certain extent. However, it is part of the future work to further evaluate the proposed approach using wider range of systems.

5.6.3 Conclusion validity

As discussed before, the hints were fixed into four iterations, and the evaluation was conducted by the researcher. However, with a real developer this might cause a risk if the users who provided the hints were not experts in the domain system or in the design patterns. This will affect the result collected based on the user intervention. In fact, the user error has an impact on all systems that require a user intervention [70]. In this case, the consequences of the user errors lead to incorrect detection results. Users with minimum programming skills might find it difficult to understand the design pattern through the code. This risk was attenuated by the
hints templates that help in providing a good deal of information to the users with minimum skills by providing predefined hints to ease the supplement. However, it is part of the future work to rerun the experimentation with expert developers. There would be a useful utility of the approach if the users were actually experts.

5.7 Summary

Three open-source systems were selected for the evaluation process. These systems selected for this purpose as they are openly-available frameworks, built with design-patterns in mind, and have been used as the basis for the evaluation of several existing design-pattern detection techniques. During the evaluation process, the accuracy was measured, and the user was given the opportunity to continue the detection process. The hint selection process was systematised as much as possible to limit the potential of bias in the results. The detection process was limited to four iterations, where for each iteration the developer was only permitted to provide hints corresponding to a specific type of hint template. The detection results indicated that the use of hints substantially increases the accuracy of the design pattern detection process. Furthermore, the overlap between the acquired results and the P-MARt [59] repository is encouraging.
The use of empirical studies in software engineering has matured considerably in the last two decades [94]. Software engineering researchers are becoming more familiar with the concept of stating their scientific hypotheses [79]. Empirical studies in software engineering are usually conducted to gather and utilise evidence to advance software engineering methods, processes, techniques, and tools [18]. There are three major strategies for performing empirical studies including; surveys, case studies, and experiments [105]. The selection of an appropriate strategy depends mainly on the purpose of the study, and on the conditions of the investigation.

The user survey strategy was adopted in the present study. If the user survey is conducted properly, it allows the researcher to generalise about the beliefs and the opinions of many people by studying a subset of them [73]. The user survey is a method for collecting information from or about people to describe, compare or explain their knowledge, attitudes and behaviour [48]. It helps to gain an insight into the people minds or the problems within a specific study.
Chapter 6. *USER STUDY: PATTERN DETECTION AND USER KNOWLEDGE*

This chapter presents a user study that involves a broader range of expert users of design patterns. Section 6.1 introduces the aims and objectives, and presents the research question and hypothesis of this study. The overall methodology and study protocol are discussed in Section 6.3. The data analysis results are presented in Section 6.4. The study threats to validity is discussed in Section 6.5. Finally, Section 6.6 presents a summery of this chapter.

### 6.1 Aims and Objectives

The research findings in Chapter 5 indicate that the expert user can gradually enhance the detection results with the use of hints. Therefore, for the purpose of highlighting this knowledge, a more user-oriented study was conducted to consider the impact of user knowledge on the detection process. This study investigated whether it is realistic that the expert user can identify useful hints to the design pattern detection process. This involved a broad range of participants, and carefully explored the selection of hints, with clear view to determining which hints are especially useful for the related patterns and contexts.

There is no widely accepted framework to study the combination of automatic detection tools with user inputs. Tonella *et al.* [115] proposed some criteria for classifying the empirical studies in reverse engineering. Thus, under those circumstances they classified the study of the user guidance in terms of whether the tool requires from the user, or what guidance is required from the user. This criteria benefit this research study to gauge factors such as the amount of user input required, and the nature of that input. Therefore, an empirical study was conducted to collect, process, and analyse the users’ responses. The data was collected from several expert users in design patterns (researchers, developers, maintainers, etc.).
6.2 Research Question & Hypotheses

In this study, the opinions extracted from different expert users of design patterns are considered and analysed by conducting a survey. This will explore the selection of hints with a view to determining which hints are especially useful. This survey will investigate the impact of user knowledge in the detection process. Furthermore, the survey will evaluate the users’ level of experience and its effect on the amount of the provided knowledge. The overall research question was:

**RQ 2.** Are users certainly aware of which hints that do and do not constitute a useful feedback for a given type of pattern?

The results of this survey are intended to evaluate the claim that users are likely to identify useful hints that can help in the detection of pattern instances in the source code. The study in this chapter will verify the following research hypothesis:

**H1:** Users can identify useful hints that feed the design pattern detection process.

Participants who are considered as experienced users are able to think effectively in regards to design patterns. Understanding the structure of the design pattern is important because it provides insights into the intent of such a pattern and into its implementation. The results analysis will show how the identification of useful hints can differentiate expert users from novice users. Participants’ knowledge affects how they understand the design pattern structure and implementations and how they identify the useful feedback in the selected scenarios. This, in turn, reflects their programming experience, and their abilities to solve design problems.

We conjecture that participants who are considered as experienced users can identifying more useful hints comparing to novice users. Therefore, for the purpose of extensively investigating the difference between the participants’ level of experience and its effect on the identification of hints, the following hypothesis will be tested:
Chapter 6. USER STUDY: PATTERN DETECTION AND USER KNOWLEDGE

**H2**: Respondents perform differently according to their experience.

We expect the novice users to be more familiar with structural and creational patterns than expert users. The reasons behind these familiarity come from our understanding and experience with design patterns. These two categories rely on the inter-classes relationships which can be recognised statically. This indicates that novice users are familiar with the structural hints, because these hints characterise some structural information that can feed in detecting the structural and the creational patterns. On the other side, behavioural patterns are problematic patterns because these patterns rely on the runtime data and the implementation of these patterns may vary. Therefore, identifying the implementation variants for the behavioural patterns requires a high level of experience and programming skills. Experienced user can identify these variants comparing to novice users based on his/her knowledge. This indicates that expert users are more familiar with sequential hints.

### 6.3 Methodology

A user-oriented study was performed to investigate the impact of user knowledge on the detection process. This involved surveying a broad range of participants, and carefully explored the selection of hints, with a clear view of determining which hints are especially useful for the related patterns and contexts. The survey starts by presenting each user with a particular pattern detection scenario from the three open source systems that were used in RQ1 (JHotDraw [2], JUnit [3], and QuickUML [5]), and then asking them to rate the usefulness of a selection of hints to support the detection of a given pattern instance in the source code.

This study of the design patterns aim to investigate the experiences of professionals in design patterns. In order to perform a more comprehensive investigation of the design patterns and to fulfil the goal of this study, only expert users in design patterns
and domain were identified as the targeted sampling frame. Their responses formed the data for the analysis process. This sample provides a further insight and helps in electing useful knowledge that is drawn upon the expert experiences.

Each participant was asked to decide on the hints that were considered as useful or not useful for a given design pattern in the selected scenarios. For each scenario, half of the hints are known to be useful (positive) and the other half are known to be useless (negative). This categorisation was not disclosed to the users. The participants were asked to evaluate the hints using the ordinal Likert scale [85]: Very Negative (VN), Negative (N), Not Significant (NU), Positive (P), and Very Positive (VP). The process of performing the survey includes three main stage [125]: Survey Design, Survey Operation, and Data Analysis. Furthermore, Kitchenham et al. [79] proposed guidelines for conducting empirical research in software engineering that were followed in this study.

6.3.1 Survey Design

This study is surveying the experience of expert users in GoF design patterns [54] in order to capture whether a user can provide useful knowledge to support the automated detection of pattern instances in the source code. A set of experts were identified as the sampling frame of the survey, and their responses formed the appropriate data for the analysis process. This section describes three important parts of survey design process including the population & sampling, the questionnaire form & protocol, and the pilot test.

6.3.1.1 Population and Sampling

Designing surveys requires careful selection of participants. The target population is the group or individuals to whom the survey applies. Sampling is the process of selecting a subset from the entire population under study. Ideally, the selected
sample should be representative in order to generalise the findings to the entire target population [78]. The number of expert users (population) in the area of design pattern community is not well-known. The LinkedIn community and other science and research networks like ResearchGate were used to find the expert users. The following specific groups were joined in LinkedIn:

- **Design Patterns in Java** (1,468 members): A group of users in design patterns to discuss topics related to the design patterns in Java.
- **Learning Design Patterns and OOAD** (3,530 members): This group is an open forum to discuss different Design Patterns, OOAD techniques and UML presentations.
- **Software Design Patterns and Architecture** (18,210 members): A group for Design pattern enthusiasts (including GoF Design Patterns) and people who can talk Software Architecture.

Different sampling techniques are applied in this survey. The aforementioned groups of experts formed the basis for a *cluster-based* sampling [78]. This is a sampling technique where the entire population is divided into defined groups and a researcher wants to survey individual members of these groups. This type of sampling was used because it’s impossible to get a complete list of all the expert users in design patterns, and because experts users are widely scattered in different districts. Furthermore, The participants who have participated in the survey were asked to nominate and pass the survey request to other people they believe they would be keen to participate in the survey. This process of selection and nomination formed a *snowball* sampling [78]. This sampling technique is often used when the population is difficult to identify as mentioned before.

### 6.3.1.2 Questionnaire Form and Protocol

Determining the appropriate questions asked is the core aspect of every survey [100]. In the conducted survey, the questions were derived from three selected scenarios
Chapter 6. USER STUDY: PATTERN DETECTION AND USER KNOWLEDGE

(See Appendix D.2). The selected scenarios were extracted from the three open source systems (JHotDraw, JUnit, and QuickUML). The final survey contained three main parts as shown in Figure 6.1. In the following, we provide a brief description for each part:

A) **Introduction:** The questionnaire starts with a description of the purpose of the study, followed by a description of the questionnaire structure and the required time to complete the survey.

B) **Demographic:** This part intended to investigate the participants contact details, their background, and experience of design patterns. The questions are as follows:

- **Question 1:** Contact information: This question asks the survey participants to provide their names and email addresses. These information were used to keep tracking the user responses, to prevent duplicate responses, and to distinguish the respondent profiles. SurveyMonkey\(^1\), which is the online survey website used in the conducted survey, also keeps the IP addresses and the date and time taken to complete the questionnaire by each participant. The data and any other information used in this questionnaire were treated anonymously and they will not be disclosed to any third party under any circumstances.

- **Question 2:** Participant experience: (How do you rate your experience in design patterns?). The purpose of this question is to classify the participants based on their experience in design pattern. This question was used to investigate the participants’ experience in design patterns. Because this research study focuses on the design patterns, it was important to know the level of experience with design patterns. This question has three ranges to rate the users experience: Beginner, Good, and Expert.

\(^1\)http://www.surveymonkey.com
• Question 3: Job role: (What is your job role?). This question is intended to identify the participant’s job role. Four choices were provided to select from; a Software developer, a Researcher, a student, and other option, which allows the participants to insert the exact description of their primary role.

C) Selected Scenarios: This part contains the three scenarios selected from the three open source systems and a set of facts for each scenario. A scenario is a real case where a design pattern is implemented in the subject systems. Each scenario contains a class diagram to capture this case, followed by a selection of eight facts “Hints” that were derived from a given case. The participants were asked to rate the usefulness of these facts according to their knowledge of design patterns. The questions are as follows:

• Questions 4-6: Three selected scenarios: For each scenario, a description from the three open source system, namely: JHotDraw, JUnit, and QuickUML, were respectively given to the participants with a link to the source code repository. For each scenario, a set of eight hints that were derived from the scenario and the selected system documentations were presented. The participants were asked to provide their assessment of the usefulness of the hints that support the detection of pattern instances in the source code. Based upon the participants’ experience of using design patterns, they were asked to rate each hints according to the extent by which it supports the detection of a design pattern in question, as:
  - Very Negative (Not at all Useful)
  - Negative (Not very useful)
  - Not Significant
  - Positive (Somewhat Useful)
  - Very Positive (Very useful)
1. Contact Information:
   - Name
   - Email Address

2. How do you rate your experience with design patterns?
   - Very Negative
   - Negative
   - Not Significant
   - Positive
   - Very Positive

3. What is your job role?
   - Software Developer
   - Researcher
   - Student
   - Other (please specify)

In the following section you are asked to provide us with your assessment of the usefulness of some facts that support our detection of pattern instances in the source code. Based upon your experiences with using design patterns, please rate each fact as:
- Very Negative
- Negative
- Not Significant
- Positive
- Very Positive

Survey on Design Pattern Detection and User Knowledge
Welcome to our Survey

Thank you for participating in our survey.
As part of a research study, we are investigating expert users with GoF design patterns.
The purpose of this survey is to investigate how an expert user can supply useful hints in the
design pattern detection process.
We begin by asking about your experiences with using design patterns, and about your views.
We then ask you to provide us with a little information about three selected scenarios.
We really appreciate your generosity of sharing your experiences and perspectives about design patterns.

Please press the "Next" button below to start this survey, which will take approximately
20 to 25 minutes to complete.

---

We know the following:
- A subclass of ToolListener contains the Transformed class and one or more Tool instances.
- Each Tool instance implements the Tool and ToolListener interfaces and has one or more ToolListeners.
- ToolListeners are called when a tool has reacted to an event

Regarding the two methods intended to be overridden in the subclasses

- First, a method runTest which is an abstraction provided as a very simple entry point for the test.*
- Then a method setUp which is an abstraction provided as an very simple entry point for the test.*

- JHotDraw provides some facts that support our suspicion:
  - It contains one CommandMenu class which is an abstraction provided as an very simple entry point for the tests.

- Consider the following facts and rate them according to the extent by which they support our suspicion:

  Please state your agreement level with each phrase:
  A. AbstractTool is an ancestor of the AbstractFacet.
  B. All concrete observers (classes of Tool) inherit (unless) from ToolListener.

---

We begin by asking about your experiences with using design patterns and about your views.
We then ask you to provide us with a little information about three selected scenarios.
We really appreciate your generosity of sharing your experiences and perspectives about design patterns.

Please press the "Next" button below to start this survey, which will take approximately
20 to 25 minutes to complete.

---

We know the following:
- A subclass of ToolListener contains the Transformed class and one or more Tool instances.
- Each Tool instance implements the Tool and ToolListener interfaces and has one or more ToolListeners.
- ToolListeners are called when a tool has reacted to an event

Regarding the two methods intended to be overridden in the subclasses

- First, a method runTest which is an abstraction provided as a very simple entry point for the test.*
- Then a method setUp which is an abstraction provided as an very simple entry point for the test.*

- JHotDraw provides some facts that support our suspicion:
  - It contains one CommandMenu class which is an abstraction provided as an very simple entry point for the tests.

- Consider the following facts and rate them according to the extent by which they support our suspicion:

  Please state your agreement level with each phrase:
  A. AbstractTool is an ancestor of the AbstractFacet.
  B. All concrete observers (classes of Tool) inherit (unless) from ToolListener.

---

We begin by asking about your experiences with using design patterns and about your views.
We then ask you to provide us with a little information about three selected scenarios.
We really appreciate your generosity of sharing your experiences and perspectives about design patterns.

Please press the "Next" button below to start this survey, which will take approximately
20 to 25 minutes to complete.

---

Survey on Design Pattern Detection and User Knowledge
Welcome to our Survey

Thank you for participating in our survey.
As part of a research study, we are investigating expert users with GoF design patterns.
The purpose of this survey is to investigate how an expert user can supply useful hints in the
design pattern detection process.
We begin by asking about your experiences with using design patterns, and about your views.
We then ask you to provide us with a little information about three selected scenarios.
We really appreciate your generosity of sharing your experiences and perspectives about design patterns.

Please press the "Next" button below to start this survey, which will take approximately
20 to 25 minutes to complete.

---

We know the following:
- A subclass of ToolListener contains the Transformed class and one or more Tool instances.
- Each Tool instance implements the Tool and ToolListener interfaces and has one or more ToolListeners.
- ToolListeners are called when a tool has reacted to an event

Regarding the two methods intended to be overridden in the subclasses

- First, a method runTest which is an abstraction provided as a very simple entry point for the test.*
- Then a method setUp which is an abstraction provided as an very simple entry point for the test.*

- JHotDraw provides some facts that support our suspicion:
  - It contains one CommandMenu class which is an abstraction provided as an very simple entry point for the tests.

- Consider the following facts and rate them according to the extent by which they support our suspicion:

  Please state your agreement level with each phrase:
  A. AbstractTool is an ancestor of the AbstractFacet.
  B. All concrete observers (classes of Tool) inherit (unless) from ToolListener.

---

We begin by asking about your experiences with using design patterns and about your views.
We then ask you to provide us with a little information about three selected scenarios.
We really appreciate your generosity of sharing your experiences and perspectives about design patterns.

Please press the "Next" button below to start this survey, which will take approximately
20 to 25 minutes to complete.

---

We know the following:
- A subclass of ToolListener contains the Transformed class and one or more Tool instances.
- Each Tool instance implements the Tool and ToolListener interfaces and has one or more ToolListeners.
- ToolListeners are called when a tool has reacted to an event

Regarding the two methods intended to be overridden in the subclasses

- First, a method runTest which is an abstraction provided as a very simple entry point for the test.*
- Then a method setUp which is an abstraction provided as an very simple entry point for the test.*

- JHotDraw provides some facts that support our suspicion:
  - It contains one CommandMenu class which is an abstraction provided as an very simple entry point for the tests.

- Consider the following facts and rate them according to the extent by which they support our suspicion:

  Please state your agreement level with each phrase:
  A. AbstractTool is an ancestor of the AbstractFacet.
  B. All concrete observers (classes of Tool) inherit (unless) from ToolListener.

---

We begin by asking about your experiences with using design patterns and about your views.
We then ask you to provide us with a little information about three selected scenarios.
We really appreciate your generosity of sharing your experiences and perspectives about design patterns.

Please press the "Next" button below to start this survey, which will take approximately
20 to 25 minutes to complete.

---

We know the following:
- A subclass of ToolListener contains the Transformed class and one or more Tool instances.
- Each Tool instance implements the Tool and ToolListener interfaces and has one or more ToolListeners.
- ToolListeners are called when a tool has reacted to an event

Regarding the two methods intended to be overridden in the subclasses

- First, a method runTest which is an abstraction provided as a very simple entry point for the test.*
- Then a method setUp which is an abstraction provided as an very simple entry point for the test.*

- JHotDraw provides some facts that support our suspicion:
  - It contains one CommandMenu class which is an abstraction provided as an very simple entry point for the tests.

- Consider the following facts and rate them according to the extent by which they support our suspicion:

  Please state your agreement level with each phrase:
  A. AbstractTool is an ancestor of the AbstractFacet.
  B. All concrete observers (classes of Tool) inherit (unless) from ToolListener.
6.3.1.3 Pilot Test

A pilot test is a simulation of the survey implementation that is carried out on a small scale of the members of the targeted population. It is conducted to expose the problems or the weaknesses in the questions, the time required, the structure, as well as to determine whether the survey objectives are clear and of a value [73]. An email invitation was sent to five selected expert users in design pattern detection. The invitation contained a clear description about the purpose of the study and the instructions for conducting the pilot test. The pilot test participants were selected based on their experience in programming, software architecture, and design. Three of the selected five experts are senior Java developers and the other two experts are researchers with a good experience in Java and software design and architecture. The selected expert users are considered as typical experts in design patterns and can, therefore, help in improving the final survey. By conducting the pilot test on this group of experts, we were able to derive tangible benefit helps to ensure a successful pilot test.

The pilot test questionnaire contained the three selected scenarios and hints, followed by a set of questions. These questions were designed and aimed to investigate; (i) whether the length of the scenarios was suitable; (ii) whether the descriptions of the scenarios were clear and easy to understand; and (iii) whether the hint choices were comprehensive. Furthermore, the pilot participants were asked to provide their comments on any problems or difficulties they found in answering the questions. After the completion of the pilot test, the evaluation forms were collected automatically. Tables (6.1, 6.2, and 6.3) summarise the feedback from the five assessors. The resulting feedbacks were used to make a number of improvements to the survey, especially in relation to the time taken for each question and the types of hints used for each scenario. This pilot test was used to evaluate the survey structure, and how long it takes to complete the questionnaire. It was also used to determine whether the questions were clear and easy to understand.
### Table 6.1: Pilot Testing - Scenario 1

<table>
<thead>
<tr>
<th>Question</th>
<th>Assessor 1</th>
<th>Assessor 2</th>
<th>Assessor 3</th>
<th>Assessor 4</th>
<th>Assessor 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scenario 1: Is the length of the scenario suitable?</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Is the scenario description clear and easy to understand?</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>None</td>
<td>Yes</td>
</tr>
<tr>
<td>Here in the preliminary information, you have said: &quot;The command calls its execute() method. The execute() method is an abstraction provided as a very simple entry point, for external classes to use the commands without knowing unnecessary details about them. execute() constitutes the sole contract that client classes need to depend on, and commands could be orchestrated, no matter if clients hold a collection of references to them they are instantiated on the fly (by means of some configuration based mechanism). The important thing is that client classes only need to know that object implementing command interface, have a simple method to be called in order to consume its provided responsibilities. Clients don't need to know the concrete data types of the commands. They will always &quot;talk&quot; to them as instances of an higher level abstraction (The command), which provide an execute() method.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>So, this is how I understood the question. First the question gives some basic overview of Figure hierarchy in JHotDraw. Then it gives several smaller facts about the library, all of which are also true. What I need to do is to point out which facts are evidences of the use of Command pattern (or evidences of that Command pattern is not used).</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>None</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Are the hint choices comprehensive?</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I don't even feel at ease answering question 7 because I don't know if information given in question 6, could be used as a fact when answering question 7. For the Command pattern, it is significant that commands are passed around as black boxes (as references to an interface or a base class). The facts don't really say whether this is the case. E.g., in &quot;Method actionPerformed() in Class commandButton invokes method execute() in Concrete commandMenu's classes.&quot;, it is not clear, whether a CommandButtons holds a reference to an abstract command (CommandMenu?) or to an instance of a concrete class. The former would be evidence in favour of CommandPattern, the later does not seem to be.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The problem is well defined. It just not clear if you are asking question, or checking the answer. Anyway, if you are trying to support your suspicious, it seems the facts are actually ordered by their support.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Yes</td>
<td>None</td>
<td></td>
<td></td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>

**Chapter 6: USER STUDY: PATTERN DETECTION AND USER KNOWLEDGE**
<table>
<thead>
<tr>
<th>Scenario 2</th>
<th>Question</th>
<th>Assessor 1</th>
<th>Assessor 2</th>
<th>Assessor 3</th>
<th>Assessor 4</th>
<th>Assessor 5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Is the length of the scenario suitable?</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Is the scenario description clear and easy to understand?</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>I consider the base class doesn't even need to be abstract. - Plus there is no need to have a &quot;set&quot; of subclasses directly in the current implementation. The most important element is the &quot;method&quot; that is defined as a &quot;template&quot;. At present I could even have a single non-ab...</td>
<td>None</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Are the hint choices comprehensive?</td>
<td>I have doubts about your interpretation of the verb &quot;define&quot; all over the questions. AFAICS, &quot;defines&quot; <em>(in the context of OOP design)</em> means &quot;implements&quot;, while &quot;specifies&quot; might mean that a method is only declared with no implementation provided (abstract). Re-read question one to make sure if that's the meaning you attempt to give, i.e an abstract method could not be &quot;defined&quot; in the same class, but in its subclasses, the abstract class does nothing but specifying it.</td>
<td>None</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>There's some mismatch between my understanding of the question and the wording of the facts. E.g., one of the facts says &quot;The class TestCase must be declared abstract.&quot; My understanding of the question says that this should instead be formulated as: &quot;The class TestCase is abstract&quot;.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Considering every fact stands for its own: A and B are a clearly implies that this is a template method pattern (TMP), but not necessarily. C also implies about TMP, but can occurs in other scenarios as well. D, E and F can be also considered as supporting the TMP, but are very common in other scenarios as well. G so you can't really know. Gathering all the facts together, they really imply that this is the TMP.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Table 6.2: Pilot Testing - Scenario 2**
<table>
<thead>
<tr>
<th>Scenario 3</th>
<th>Question</th>
<th>Assessor 1</th>
<th>Assessor 2</th>
<th>Assessor 3</th>
<th>Assessor 4</th>
<th>Assessor 5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Is the length of the scenario suitable?</td>
<td>Yes</td>
<td>None</td>
<td>Yes</td>
<td>Yes</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>Is the scenario description clear and easy to understand?</td>
<td>None</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>Are the hint choices comprehensive?</td>
<td>Yes</td>
<td>None</td>
<td>Yes</td>
<td>Yes</td>
<td>None</td>
</tr>
</tbody>
</table>

I would change this “All Concrete observers classes of ToolListener” by this “All concrete observers (subclasses of ToolListener)”, because as written, it might make one think that there is a trap… and you are trying to say that “concrete observers” “of” ToolListener, as if ToolListener would play the role of “observer”.

Table 6.3: Pilot Testing - Scenario 3
6.3.2 Survey Operation

The next stage in the process of performing the survey is the survey operation [125]. In this phase, the data survey answers are collected and organised. The two main steps in the operation process are: (i) Survey preparation, where the participants are selected and the questionnaire forms are distributed; (ii) Data collection, which is carried out after the participants answer the questionnaire forms.

6.3.2.1 Preparation

A survey request letter was posted on LinkedIn and ResearchGate to request the members of the selected groups to participate in the survey. This letter included a description of the study, with a link for the on-line survey (See Appendix D.1). Finding people who can be considered as experts and willing to participate was a challenge. Therefore, every participant was asked to nominate and pass the on-line survey link to colleagues who can be considered as familiar users of design patterns. Some guidelines and factors were considered in order to motivate the participants to take the exercise seriously, and to enhance the response rate [111]. Furthermore, a set of ethical aspects were considered in the participants’ motivation process [125].

6.3.2.2 Collection

After posting the survey, it was opened on the 11th of July 2014 and closed on the 31st of August 2014. During the data collection process, the survey responses were checked through a preliminary analysis to make sure the data were collected correctly [125]. The result of the preliminary analysis of the data showed that the collected answers were reasonable, and the survey was conducted in the desired way. Sixty users participated in this empirical study. Forty-eight of them completed the on-line survey successfully. In the final data analysis, twelve responses were eliminated due to non-completion. The response rate was not calculated because, as mentioned
above in Section 6.3.1.1, the size of the population was unknown. The data were collected and stored by the SurveyMonkey website database.

6.3.3 Data Analysis

The data collected in the previous step need to be analysed to draw a conclusion based on this data. Typically, survey analysis is conducted from the quantitative and qualitative data that have been generated by collecting the responses in the operation phase. The quantitative data consists of numbers, while qualitative data consists of words. In this case, the respondents responses were quantitatively analysed. A quantitative interpretation was carried out in order to draw the conclusion. The quantitative data analysis includes the analysis of the descriptive statistics and hypothesis testing [125]. This section introduces the statistical methods used to analyse the collected answers. To understand the analysis procedure, it is important to understand the most common measurement scales and data types. The data can be described as either categorical data which can be analysed using the nominal scale, or as a continuous data which can be analysed using the ordinal, the interval, and the ratio scales. [25, 73, 125].

- The nominal scale maps the response categories to names or symbols. The numbers represent the categories, and have no meaning except as the assignment for the text response. For example, the arbitrary numbers/labels assignments for the gender (e.g. 1 = male, 2 = female).
- The ordinal scale is a more powerful scale that ranks the elements in order. The numbers only specify the order, and each number has a value in a relationship to the other. Examples of the ordinal scale are the measures of “low”, “medium”, “high”, and “greater than”.
- The interval scale uses numbers to specify the order and the distance between the attributes. It is usually used when the data has a range and there is a
meaningful difference between the attributes. For example, the temperature ranges in Celsius or Fahrenheit.

- The ratio scale also uses numbers to specify the order and represents a meaningful relative distance between the attributes on a scale. For example, the measurement of the duration, the age, and the years of experience.

The main reason to understand these classification, is to underpin the choice of the type of the statistical test [86]. Table 6.4 shows the measurement scale used in the survey. Typically, survey analysis is conducted on the quantitative or the qualitative data collected from the participants’ responses. The quantitative data consists of numbers, while the qualitative data consists of words. In this study, the respondents’ responses were quantitatively analysed. The quantitative data analysis includes the analysis of the Descriptive Statistics and Hypothesis Testing [125].

<table>
<thead>
<tr>
<th>Question</th>
<th>Scale</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q2: Experience</td>
<td>Nominal</td>
</tr>
<tr>
<td>Q3: Job Role</td>
<td>Nominal</td>
</tr>
<tr>
<td>Q4: Scenario 1 Facts</td>
<td>Ordinal</td>
</tr>
<tr>
<td>Q5: Scenario 2 Facts</td>
<td>Ordinal</td>
</tr>
<tr>
<td>Q6: Scenario 3 Facts</td>
<td>Ordinal</td>
</tr>
</tbody>
</table>

**Table 6.4: Measurement scale**

### 6.3.3.1 Data Coding

Coding is the assignment of a number value to respondent answer, it facilitates the data manipulation, and allows for quantitative analysis [73]. For coding the questions (Q2 and Q3), a code has been assigned to each category as follows:

- Experience: (Beginner = 1, Good = 2, Expert = 3)
- Job role: (Software developer = 1, Researcher = 2, Student = 3, other = 4)

For questions (Q4 - Q6), each fact was evaluated using the five-point Likert scale [85]. For every fact in the selected scenario, the participants were asked to evaluate
the facts using the ranked choices: Very Negative, Negative, Not Significant, Positive, and Very Positive. Table 6.5 presents the coding used for the ranking of the usefulness of facts in the conducted survey.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Very Negative (VN)</th>
<th>Negative (N)</th>
<th>Not Significant (NU)</th>
<th>Positive (P)</th>
<th>Very Positive (VP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 6.5: The assessment of the usefulness of facts

As discussed above in Section 6.3, for each scenario, half of the hints are known to be useful and the other half are known to be useless. Therefore, the provided answers implicitly have hints that were considered as positive, and hints that were considered as negative. With this in mind, the responses for each participant were imported into SPSS [32]. Each hint was assigned by a variable. For example, S1_H1_P: denotes Scenario 1, Hint 1 was considered as a Positive hint in the analysis, and S2_H3_N: denotes Scenario 2, Hint 3 was considered as a Negative.

6.3.3.2 Descriptive Statistics

Descriptive statistics focus on the quantitative data analysis. It can be used to visualise and describe the data graphically using tables and charts [87]. The goal of descriptive statistics is to present how the data set is distributed, and to provide a general view of the collected data [125]. With this in mind, the main task in the descriptive analysis was to manifest the frequency distribution of the responses of each question in the survey. The term distribution means the occurrence of each value selected by the respondents over the range of all valid values [73]. These frequencies help the researcher to gain knowledge about how many respondents answered at a specific level of the variable [86]. The use of Web-based questionnaires eased the data collection and organisation by automating these tasks. Section 6.4 starts by presenting the descriptive statistics for the participants profiles based on the level of experience and on the primary job role, followed by the frequency of the selected hints.
Chapter 6. USER STUDY: PATTERN DETECTION AND USER KNOWLEDGE

6.3.3.3 Hypothesis Testing

Hypothesis testing is a statistical method used to evaluate the hypothesis and to assist in making the decision by using a sample data. The objective of hypothesis testing is to see whether it is possible to reject a certain null hypothesis $H_0$ based on the sample from some statistical distribution [125]. The null hypothesis that represents the case is generally assumed true until an evidence indicates otherwise. In that case, the null hypothesis $H_0$ is rejected and the alternative hypothesis $H_A$ is accepted. The process followed for testing the hypothesis is as follows:

1. Define the scientific hypothesis that covers the aim of the conducted investigation.
2. Derive the statistical hypothesis, including the null $H_0$ and the alternative hypothesis $H_A$.
3. Decide what kind of testing is more appropriate to test the case.
4. Select the test technique to apply.
5. Accept or reject the null hypothesis based on the data analysis.

6.3.3.4 Test of Normality

Tests can be classified into a parametric and a non-parametric test [125]. The parametric test assumes that the data is normally distributed. The non-parametric test can be used when the data does not meet the assumptions needed to use the parametric test [86]. The selection of different testing methods is carried with respect to the type of the test either parametric or non-parametric. Therefore, it is necessary to check whether the observed distribution fits the normal distribution based on the respondents’ responses in the three selected scenarios. Let’s assume the following:

\[ H_0: P > 0.05. \text{ the observed distribution fits the normal distribution.} \]
\[ H_A: P \leq 0.05. \text{ the observed distribution does not fit the normal distribution.} \]
Chapter 6. **USER STUDY: PATTERN DETECTION AND USER KNOWLEDGE**

### Table 6.6: Normality Testing - Scenario 1

<table>
<thead>
<tr>
<th></th>
<th>Kolmogorov-Smirnov&lt;sup&gt;a&lt;/sup&gt;</th>
<th>Shapiro-Wilk</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Statistic</td>
<td>df</td>
</tr>
<tr>
<td>Hint 1</td>
<td>.279</td>
<td>48</td>
</tr>
<tr>
<td>Hint 2</td>
<td>.251</td>
<td>48</td>
</tr>
<tr>
<td>Hint 3</td>
<td>.212</td>
<td>48</td>
</tr>
<tr>
<td>Hint 4</td>
<td>.253</td>
<td>48</td>
</tr>
<tr>
<td>Hint 5</td>
<td>.307</td>
<td>48</td>
</tr>
<tr>
<td>Hint 6</td>
<td>.306</td>
<td>48</td>
</tr>
<tr>
<td>Hint 7</td>
<td>.214</td>
<td>48</td>
</tr>
<tr>
<td>Hint 8</td>
<td>.204</td>
<td>48</td>
</tr>
</tbody>
</table>

<sup>a</sup> Lilliefors Significance Correction

### Table 6.7: Normality Testing - Scenario 2

<table>
<thead>
<tr>
<th></th>
<th>Kolmogorov-Smirnov&lt;sup&gt;a&lt;/sup&gt;</th>
<th>Shapiro-Wilk</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Statistic</td>
<td>df</td>
</tr>
<tr>
<td>Hint 1</td>
<td>.265</td>
<td>48</td>
</tr>
<tr>
<td>Hint 2</td>
<td>.212</td>
<td>48</td>
</tr>
<tr>
<td>Hint 3</td>
<td>.192</td>
<td>48</td>
</tr>
<tr>
<td>Hint 4</td>
<td>.338</td>
<td>48</td>
</tr>
<tr>
<td>Hint 5</td>
<td>.284</td>
<td>48</td>
</tr>
<tr>
<td>Hint 6</td>
<td>.378</td>
<td>48</td>
</tr>
<tr>
<td>Hint 7</td>
<td>.208</td>
<td>48</td>
</tr>
<tr>
<td>Hint 8</td>
<td>.198</td>
<td>48</td>
</tr>
</tbody>
</table>

<sup>a</sup> Lilliefors Significance Correction

### Table 6.8: Normality Testing - Scenario 3

<table>
<thead>
<tr>
<th></th>
<th>Kolmogorov-Smirnov&lt;sup&gt;a&lt;/sup&gt;</th>
<th>Shapiro-Wilk</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Statistic</td>
<td>df</td>
</tr>
<tr>
<td>Hint 1</td>
<td>.228</td>
<td>48</td>
</tr>
<tr>
<td>Hint 2</td>
<td>.255</td>
<td>48</td>
</tr>
<tr>
<td>Hint 3</td>
<td>.180</td>
<td>48</td>
</tr>
<tr>
<td>Hint 4</td>
<td>.225</td>
<td>48</td>
</tr>
<tr>
<td>Hint 5</td>
<td>.312</td>
<td>48</td>
</tr>
<tr>
<td>Hint 6</td>
<td>.272</td>
<td>48</td>
</tr>
<tr>
<td>Hint 7</td>
<td>.226</td>
<td>48</td>
</tr>
<tr>
<td>Hint 8</td>
<td>.268</td>
<td>48</td>
</tr>
</tbody>
</table>

<sup>a</sup> Lilliefors Significance Correction
Chapter 6. USER STUDY: PATTERN DETECTION AND USER KNOWLEDGE

Tables 6.6, 6.7, and 6.8 show that the results have significant differences ($p \leq 0.05$). Thus, the null hypothesis is rejected. The observed distribution from the three scenarios do not fit the normal distribution. Therefore, a non-parametric test was used in the analysis of the hypothesis in this study. This is because the collected data did not show a normal distribution based on the normality testing of the three scenarios.

6.4 Results & Discussion

Before conducting the data analysis, it was essential to be acquainted with the distributions of the respondents. Thus, analysing the collected responses began with the respondents profiles based on the level of experience and on the primary job role. In detail, the composition of the respondents was analysed by using the demographic questions (Q2 - Q3). The majority of the respondents were software developers (61%) and researchers (25%). The distribution of the respondents based on the level of experience is; beginner (33%), good (52%), and expert (15%). Figure 6.2 and 6.3 summarise the distribution of the respondents based on their job role and level of experience.

![Figure 6.2: Profile of respondents: Job Role](image)

112
Chapter 6. **USER STUDY: PATTERN DETECTION AND USER KNOWLEDGE**

![Figure 6.3: Profile of respondents: Level of experience](image)

### 6.4.1 The Frequency of Hints

The frequencies of the selection of useful hints in each scenario (Q4 - Q6) are presented as follows. Table 6.9 shows the frequency of the selection of hints in percentage of the number of respondents ranking. The left column of Table 6.9 shows the hints in the three scenarios, along with the ranking scale values of these hints from left to right (VN,..., VP). The results show that the participants were able to successfully rate the positive hints as useful hints, and the negative hints as useless hints.

In the first scenario, the positive hints are H2, H3, H5, and H6. On the other hand, the negative hints are H1, H4, H7, and H8. It can be seen the positive group were ranked significantly more useful than the negative group. For example, the positive hint H2 was rated in a percentage of 43.8 % as useful, and 14.6 as very useful (58.4 % useful). The negative hint H1 was rated with percentage of 16.7 % as not at all useful, and 45.8 % as not very useful (62.5 % useless).

In the second scenario, the positive hints are H1, H2, H4, and H6. On the other hand, the negative hints are H3, H5, H7, and H8. Similarly in the first scenario, the overall ranking of positive hints group as useful was very positive. Furthermore, the ranking of the negative hints group as useless shows that the respondents were able to define these hints as useless for a given type of pattern. Finally, the third
Chapter 6. *USER STUDY: PATTERN DETECTION AND USER KNOWLEDGE*

scenario, the positive hints group are H2, H4, H6, and H8. The negative hints group are H1, H3, H5, and H7. It was also clear that the respondents were very aware of the useful and the useless hints in this scenario.

<table>
<thead>
<tr>
<th>Hints</th>
<th>VN</th>
<th>N</th>
<th>NU</th>
<th>P</th>
<th>VP</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1_H1_N</td>
<td>16.7%</td>
<td>45.8%</td>
<td>20.8%</td>
<td>12.5%</td>
<td>4.2%</td>
</tr>
<tr>
<td>S1_H2_P</td>
<td>0.0%</td>
<td>10.4%</td>
<td>31.3%</td>
<td>43.8%</td>
<td>14.6%</td>
</tr>
<tr>
<td>S1_H3_P</td>
<td>0.0%</td>
<td>8.3%</td>
<td>31.3%</td>
<td>37.5%</td>
<td>22.9%</td>
</tr>
<tr>
<td>S1_H4_N</td>
<td>4.2%</td>
<td>43.8%</td>
<td>35.4%</td>
<td>10.4%</td>
<td>6.3%</td>
</tr>
<tr>
<td>S1_H5_P</td>
<td>2.1%</td>
<td>4.2%</td>
<td>18.8%</td>
<td>54.2%</td>
<td>20.8%</td>
</tr>
<tr>
<td>S1_H6_P</td>
<td>2.1%</td>
<td>8.3%</td>
<td>18.8%</td>
<td>52.1%</td>
<td>18.8%</td>
</tr>
<tr>
<td>S1_H7_N</td>
<td>8.3%</td>
<td>35.4%</td>
<td>31.3%</td>
<td>20.8%</td>
<td>4.2%</td>
</tr>
<tr>
<td>S1_H8_N</td>
<td>16.7%</td>
<td>31.3%</td>
<td>22.9%</td>
<td>25.0%</td>
<td>4.2%</td>
</tr>
<tr>
<td>S2_H1_P</td>
<td>2.1%</td>
<td>14.6%</td>
<td>22.9%</td>
<td>43.8%</td>
<td>16.7%</td>
</tr>
<tr>
<td>S2_H2_P</td>
<td>2.1%</td>
<td>14.6%</td>
<td>25.0%</td>
<td>33.3%</td>
<td>25.0%</td>
</tr>
<tr>
<td>S2_H3_N</td>
<td>10.4%</td>
<td>31.3%</td>
<td>29.2%</td>
<td>20.8%</td>
<td>8.3%</td>
</tr>
<tr>
<td>S2_H4_P</td>
<td>4.2%</td>
<td>0.0%</td>
<td>22.9%</td>
<td>60.4%</td>
<td>12.5%</td>
</tr>
<tr>
<td>S2_H5_N</td>
<td>4.2%</td>
<td>47.9%</td>
<td>29.2%</td>
<td>14.6%</td>
<td>4.2%</td>
</tr>
<tr>
<td>S2_H6_P</td>
<td>0.0%</td>
<td>16.7%</td>
<td>16.7%</td>
<td>62.5%</td>
<td>4.2%</td>
</tr>
<tr>
<td>S2_H7_N</td>
<td>6.3%</td>
<td>35.4%</td>
<td>35.4%</td>
<td>16.7%</td>
<td>6.3%</td>
</tr>
<tr>
<td>S2_H8_N</td>
<td>18.8%</td>
<td>31.3%</td>
<td>27.1%</td>
<td>18.8%</td>
<td>4.2%</td>
</tr>
<tr>
<td>S3_H1_N</td>
<td>22.9%</td>
<td>35.4%</td>
<td>22.9%</td>
<td>14.6%</td>
<td>4.2%</td>
</tr>
<tr>
<td>S3_H2_P</td>
<td>4.2%</td>
<td>16.7%</td>
<td>18.8%</td>
<td>39.6%</td>
<td>20.8%</td>
</tr>
<tr>
<td>S3_H3_N</td>
<td>6.3%</td>
<td>29.2%</td>
<td>33.3%</td>
<td>27.1%</td>
<td>4.2%</td>
</tr>
<tr>
<td>S3_H4_P</td>
<td>0.0%</td>
<td>4.2%</td>
<td>31.3%</td>
<td>41.7%</td>
<td>22.9%</td>
</tr>
<tr>
<td>S3_H5_N</td>
<td>10.4%</td>
<td>52.1%</td>
<td>20.8%</td>
<td>16.7%</td>
<td>0.0%</td>
</tr>
<tr>
<td>S3_H6_P</td>
<td>0.0%</td>
<td>8.3%</td>
<td>29.2%</td>
<td>47.9%</td>
<td>14.6%</td>
</tr>
<tr>
<td>S3_H7_N</td>
<td>6.3%</td>
<td>37.5%</td>
<td>29.2%</td>
<td>20.8%</td>
<td>6.3%</td>
</tr>
<tr>
<td>S3_H8_P</td>
<td>6.3%</td>
<td>4.2%</td>
<td>27.1%</td>
<td>45.8%</td>
<td>16.7%</td>
</tr>
</tbody>
</table>

Table 6.9: Selection of Hints (in percentage of number of respondents)

Because of the similarities between (VN & N) and (P & VP) answers, we adopted the established approach [76] of aggregated the answers as: (Negative: VN and N), (Neutral: NU), and (Positive: P and VP). The frequencies of the respondents answers on each fact in the selected scenarios was computed using the three-point Likert scale. Figures 6.4 and 6.5 show the analysis results of the selected user hints in percentage of the number of the respondents in the three selected scenarios. This clearly shows that there was a tendency to rate positive hints as useful, and to rate
negative hints as useless. This indicates that the users are certainly aware of which hints do and do not constitute a useful feedback for a given type of pattern.

![Positive Hints](image1)

**Figure 6.4: Positive Hints Frequencies**

Figure 6.4 shows the positive hints in the three scenarios numbered from 1 to 12 (4 positive hints in each scenario). It can be seen from the green bars that the rating of positive hints as useful is trending in the three scenarios with the values ranging between 60% - 75%. The blue bars show the rating of these positive hints as useless is low. This indicates that the respondents have clearly identified the proper hints for the selected scenarios.

![Negative Hints](image2)

**Figure 6.5: Negative Hints Frequencies**

Similarly, Figure 6.5 shows the negative hints in the three scenarios. It can be seen that the blue bars were upward trending because the user have considered and
identified those hints as useless with a frequencies range of 45% - 62%. Overall, Table 6.9 and figures 6.4 and 6.5 provide important insights that the user can identify useful hints that feed the design pattern detection process. These results provide a proof for the hypothesis $H_1$ in Section 6.2.

6.4.2 The Relation between the Experience and Useful Hints

The relation between the user level of experience (Q3) and the average of selecting useful hints in questions (Q4 - Q6) is reported in this section. We conjecture that user who is considered as a more expert can provide more useful hints compared to the other users. Therefore, the percentage of the assessment for the positive facts in each level of experience category was counted. Tables (6.10 - 6.12) show the percentage of the selecting useful hints in the selected scenario with respect to the different level of experience.

<table>
<thead>
<tr>
<th>Experience</th>
<th>Hint 1</th>
<th>Hint 2</th>
<th>Hints 3</th>
<th>Hint 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beginner</td>
<td>22.75%</td>
<td>21.0%</td>
<td>18.28%</td>
<td>17.94%</td>
</tr>
<tr>
<td>Good</td>
<td>23.96%</td>
<td>23.76%</td>
<td>26.76%</td>
<td>25.96%</td>
</tr>
<tr>
<td>Expert</td>
<td>30.43%</td>
<td>35.14%</td>
<td>30.64%</td>
<td>34.29%</td>
</tr>
</tbody>
</table>

Table 6.10: The relation between the Experience and the Hints in Scenario 1

<table>
<thead>
<tr>
<th>Experience</th>
<th>Hint 1</th>
<th>Hint 2</th>
<th>Hints 3</th>
<th>Hint 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beginner</td>
<td>18.75%</td>
<td>16.78%</td>
<td>24.09%</td>
<td>24.56%</td>
</tr>
<tr>
<td>Good</td>
<td>24.32%</td>
<td>26.62%</td>
<td>23.08%</td>
<td>23.58%</td>
</tr>
<tr>
<td>Expert</td>
<td>38.29%</td>
<td>34.57%</td>
<td>30.50%</td>
<td>27.64%</td>
</tr>
</tbody>
</table>

Table 6.11: The relation between the Experience and the Hints in Scenario 2

<table>
<thead>
<tr>
<th>Experience</th>
<th>Hint 1</th>
<th>Hint 2</th>
<th>Hints 3</th>
<th>Hint 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beginner</td>
<td>21.16%</td>
<td>19.31%</td>
<td>17.69%</td>
<td>22.31%</td>
</tr>
<tr>
<td>Good</td>
<td>24.48%</td>
<td>25.12%</td>
<td>25.52%</td>
<td>24.40%</td>
</tr>
<tr>
<td>Expert</td>
<td>32.21%</td>
<td>34.14%</td>
<td>36.43%</td>
<td>29.86%</td>
</tr>
</tbody>
</table>

Table 6.12: The relation between the Experience and the Hints in Scenario 3

From these tables, it can be clearly seen that the percentage of participants identifying the useful hints increases gradually with users’ level of experience. They show
that there are no significant differences on the identifying of useful hints between
different level of experience. As mentioned in Section 6.2, we expect the beginner
users are more familiar with structural and creational patterns. This indicates they
will be familiar with the structural hints. On the other hand, more experienced user
can identify more sequential hints comparing to beginner. Within the three levels of
experience, only the results collected from the users considered as expert is slightly
different from those acquired from the other two levels. Expert users can, therefore,
provide more positive hints compared to the beginner and good users. Therefore,
for the purpose of extensively investigating the difference between the participants’
level of experience and its effect on the selection of hints, hypothesis $H_2$ in Section
6.2 was tested ”Users can identify useful hints that feed the design pattern detection
process”.

\[
H_0: P > 0.05. \text{ The three levels of experience will not have any significant differences.}
\]
\[
H_A: P \leq 0.05. \text{ The three levels of experience will differ significantly.}
\]

In order to test this hypothesis, a Kruskal-Wallis [86] test was conducted to examine
the significant differences between the levels of experience in term of identifying
positive hints. The Kruskal-Wallis test is a non-parametric version of analysis of
variance. It is used to compare the scores on a continuous variable by a levels of
a categorical variable with three levels or more [86]. In this case, the continuous
variables are the hints for each scenario in (Q4 - Q6) and the categorical variables
are the three levels of experience (Beginner, Good, and Expert).

As shown in Table 6.13, for half of the selected hints $P$ value is $\leq 0.05$, this means
rejecting the null hypothesis. This indicates that there are significant differences in
the participants’ level of experience. The three levels of experience will differ signifi-
cantly in identifying useful hints. This is influenced by users’ level of experiences.
Overall, these results show that the percentage of identifying positive feedback in-
creases with the experience on design patterns.
### Table 6.13: kruskall-Wallis Test

<table>
<thead>
<tr>
<th>Hints</th>
<th>Chi-Square</th>
<th>df</th>
<th>Asymp. Sig.</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1_H2_P</td>
<td>1.749</td>
<td>2</td>
<td>0.417</td>
</tr>
<tr>
<td>S1_H3_P</td>
<td>5.655</td>
<td>2</td>
<td>0.059</td>
</tr>
<tr>
<td>S1_H5_P</td>
<td>6.244</td>
<td>2</td>
<td><strong>0.044</strong></td>
</tr>
<tr>
<td>S1_H6_P</td>
<td>8.526</td>
<td>2</td>
<td><strong>0.014</strong></td>
</tr>
<tr>
<td>S2_H1_P</td>
<td>10.582</td>
<td>2</td>
<td><strong>0.005</strong></td>
</tr>
<tr>
<td>S2_H2_P</td>
<td>9.752</td>
<td>2</td>
<td><strong>0.008</strong></td>
</tr>
<tr>
<td>S2_H4_P</td>
<td>2.033</td>
<td>2</td>
<td>0.362</td>
</tr>
<tr>
<td>S2_H6_P</td>
<td>0.617</td>
<td>2</td>
<td>0.743</td>
</tr>
<tr>
<td>S3_H2_P</td>
<td>3.309</td>
<td>2</td>
<td>0.191</td>
</tr>
<tr>
<td>S3_H4_P</td>
<td>6.287</td>
<td>2</td>
<td><strong>0.043</strong></td>
</tr>
<tr>
<td>S3_H6_P</td>
<td>10.446</td>
<td>2</td>
<td><strong>0.005</strong></td>
</tr>
<tr>
<td>S3_H8_P</td>
<td>1.611</td>
<td>2</td>
<td>0.447</td>
</tr>
</tbody>
</table>

#### 6.5 Threats to validity

The important question is that how valid the results acquired from the conducted survey are. The results should be valid for the population from which the sample is drawn. In addition, the acquired results should generalise to a broader population. Three main types of threats to validity including an internal, an external, and an conclusion validity.

#### 6.5.1 Internal Validity

This validity focusses on the effectiveness of the responses [125]. In this study, three scenarios were selected, and the users were asked to provide their knowledge. The main threat is the design of the asked questions (selected scenarios). In fact, the selected scenarios and facts were derived from three well-known systems. This raises the possibility that the selection of the scenarios and facts was specific to these systems. However, with other systems, the user might require more efforts to decide the proper hints. This risk was attenuated by the fact that the selected scenarios are relatively diverse from three different systems for three different design patterns.
However, it is part of the future work to conduct a study using a wider range of systems and design patterns.

### 6.5.2 External Validity

This validity concerns about to which extent the results of a study can be held to be true for other people, places, or times. In particular, external validity focuses on the completeness of the responses [125]. The main point is how does the sample was selected as a representative for the population. As was discussed in Section 6.3.1.1, there is no available group or a well-known population for researchers or experts in the area of design pattern detection. Thus, the current research networks were used to simulate the sampling process. In the current study, the bias in selecting participants was mitigated by posting the survey on selected groups and asking the participants to nominate others to participate. In future work, it would be interesting to select expert users from a well-known population (i.e. software students, developers, etc.).

### 6.5.3 Conclusion validity

This validity concerns with the issues that affect the ability to draw the correct conclusion [125]. As discussed before, this study of design patterns aimed to investigate the experiences of professionals with design patterns in the domain of software engineering. However, if the users who have participated were not experts in the domain system or the design patterns, this will affect the conclusion that was derived upon the collected data from their responses. There would be a useful input on the utility of the approach if the users were actually experts. The results might have differed if students were surveyed as part of the study. The students’ responses reflect the students’ ability in understanding design problems with patterns and interpreting the impact of patterns. Students having minimum programming skills might find it
difficult to understand the design pattern through the code. The students’ responses are associated with the effectiveness of teaching them the design patterns in a software engineering course. Some design patterns are less popular than others. This indicates that software students spend more time on patterns, which are fundamentally difficult to understand [28]. Students might find it difficult to relate the applied design patterns to specific source code elements. This is possibly due to the fact that most patterns are presented in Gamma et al. [54] as solutions to known implementation problems rather than as a treatment of design issues in reverse engineering manner. In other words, design patterns are presented as a consequence rather than a goal of pattern application. This observation can be valuable in redesigning and in enhancing the design pattern courses.

6.6 Summary

An empirical study was conducted in this chapter. Sixty expert users participated in this study with forty-eight users successfully completed the survey (incomplete surveys were discarded). This study aimed to highlight the importance of involving user knowledge in the design pattern detection. The survey was conducted to investigate the relation between the user and the design pattern detection process, and how does the expert users can supplement useful hints in the detection process. The main reason for this study is to provide evidence of the importance of users’ involvements in the design pattern detection process. The collected data were quantitatively analysed using descriptive statistics and hypothesis testing. A non-parametric method test was applied for the hypothesis testing, which aimed to study the relation between the user level of experience and the amount of the provided useful hints. The results indicated that the user can feed the detection process with useful hints to enhance the detection of the design patterns in the source code. In addition, the amount of this knowledge is influenced by users’ level of experiences.
Chapter 7

CONCLUSIONS AND FUTURE WORK

This chapter outlines the conclusions and the future work of the research presented in the thesis. Section 7.1 presents a summary of the research and its outcomes. Section 7.2 discusses the conclusions of the conducted research. Finally, Section 7.3 lists some suggestions for further research directions.

7.1 Overall Summary

The work in thesis highlights the important role that can be played by the experts in the design pattern detection process. It argues that this process needs to be considered as an iterative and user-driven process. In order to achieve this, a constraint-based framework was built to provide the user knowledge in the form of a small amount of useful hints. The proposed approach is compatible with the existing constraint-based pattern detection approaches, and can act as a complementary ‘layer’ to enable additional feedback from the user. The proposed approach was evaluated using an experimental case study that applied it to three open-source
systems. A more substantive controlled user study that involved a broader range of participants was also conducted for the evaluation of the proposed approach. It explored the process of hint selection, with a view to determining which hints are especially useful in which contexts.

- Chapter 2 introduced the history of design patterns, and explained how design patterns have been interpreted in the context of object-oriented design. The general concept of design patterns, their classification, and relevance were discussed in this chapter. Finally, this chapter discussed the impact of using design patterns, the design pattern implementation, and the identification of design patterns.

- Chapter 3 covered the main concepts related to the design pattern detection process, and reviewed some of the work related to this area. Different approaches were built to date based on different detection techniques. Some of these approaches are tool-supported. The key challenges and problems that currently face the available approaches of design pattern detection are highlighted. These challenges constitute the main motivation for the research topic in this thesis.

- Chapter 4 presented the proposed approach which seeks to enable the user to feed small ‘hints’ into the detection process. This is unlike most of the existing tools, which permit a limited degree of user-interaction. The proposed approach enables the user to be much more expressive, whilst remaining easy to use. Hints can provide information about the dynamic behaviour, or provide partial knowledge about the pattern implementations themselves. The use of hints can gradually guide the constraint solver towards a more accurate set of detected pattern instances.

- Chapter 5 presented the experiments of the proposed approach. The proposed approach was applied to a selection of openly available software systems. The specific research question was “To what extent does the use of hints lead to an increase in the accuracy of the design pattern detection results?”. 

122
Chapter 7. CONCLUSIONS AND FUTURE WORK

The evaluation results indicated that this approach can yield a significant improvement in the detection accuracy based on a relatively small amount of input fed by the developer.

• Chapter 6 covered the user study, which also aimed to elaborate upon the results of Chapter 5. The specific research question was “Are users certainly aware of which hints that do and do not constitute a useful feedback for a given type of pattern?”. The study findings indicated that users can supplement the design pattern detection process with a positive feedback that can enhances the detection of design pattern.

7.2 Conclusions

Aims This thesis was founded on the observation that, in order to produce more accurate results, it is necessary to actively involve the expert user in the pattern detection process. The aim was to provide a structured process involving the user expert to complement the existing constraint-based pattern detection techniques. It proposed a framework that enables the user to supply their knowledge in the form of ‘hints’, fed as small facts about the source code or the pattern implementations that can enhance or curtail the results.

Findings The approach was applied to three openly available systems for evaluation. The evaluation case study results indicated that user knowledge enhances the detection process and can eliminate the detection of false positives. The use of hints can, therefore, substantially increase the accuracy of the detected patterns. A user study was also conducted for evaluation. This study involved a broader range of expert users. The results of this study indicated that there was a tendency to rate positive hints as useful and to rate negative hints as useless. This suggests that users are certainly aware of which hints do and do not constitute a useful feedback for a given type of pattern.
The evaluation case study findings enhance our understanding of how the user knowledge feeds the detection process. On the other hand, the user study findings could be used to show that the expert user can supplement the design pattern detection process with useful hints. These hints can enhance the detection of design pattern instances in the source code. These hints can include the knowledge about the structure, the behaviour, and the possible implementation deviations from traditional design pattern specifications. Therefore, it can be concluded that pattern detection must necessarily be a user-driven process. The results of this research support the idea that the user must be provided with as much opportunity as possible to contribute their knowledge about design pattern occurrences.

7.2.1 Findings Significance in Design Pattern Detection

Fully automated pattern detection techniques are intrinsically limited, especially when it comes to the detection of the behavioural patterns. Current design pattern detection techniques and tools do not produce accurate results because they do not enable sufficient intervention from users. This research has shown that a limited amount of user input can complement automated detection. It enhances our understanding of the important role of the user expert in the detection process. The user-driven approach proposed in this thesis provides a suitable basis for bridging the gap, by enabling the user to supply the information that is not available to those techniques.

This research strengthens the idea that the use of micro-structures has a global usefulness to the design pattern detection process [49, 92, 93, 112, 113]. However, micro-structures alone are not enough to enhance the detection results. The integration of the user in an iterative user-driven process helps to refine and gradually enhances the detection results. The contribution is the iterative process where the user can refine the detection results in an incremental manner. Future detection techniques should therefore concentrate on considering the role of the user expert.
Chapter 7. CONCLUSIONS AND FUTURE WORK

in the detection process. This research confirm that there is a need for hybrid approaches that combines the automatic tools with the user inputs.

The general approach of gradually acquiring constraints has previously been successfully applied to software engineering [67]. It has shown that it is possible to significantly improve the accuracy of a solution (in the context of software remodularisation area) with a relatively small facts from the developer. In the same manner, this research will serve as the basis for future studies in the design pattern detection area. This research provides an evidence that in order to produce more accurate detection results, the detection process cannot be considered as a fully automated process.

7.2.2 Findings Significance in Software Maintenance

In object-oriented legacy systems, the source code can contain thousands of classes and millions of lines of codes. Furthermore, design documents are often missing, or may not accurately match the source code [90]. Therefore, the absence of a valid documentation has a bad influence on the process of understanding and comprehending these systems. This research extends our knowledge of the use of design patterns, which can provide a better understanding of a legacy system program code. If the software maintainer is aware of the patterns used in the code, this will assist in understanding the system requirements and in considering new requirements from the reversed design patterns.

The findings of this thesis could be used to provide insights into the source code, which can help the maintainer to understand the analysed system. The recovery of the source code design and architecture makes the maintenance task easier. It helps inspecting the source code artefacts in order to gain a sufficient level of knowledge about the analysed system design and architecture. Furthermore, understanding and organising the source code as a set of well-known structures (GoF Design patterns)
gives the software maintainer a faster firm grasp of the whole system structure and organisation. This research helps in organising the legacy system around recurring patterns (represented as a set of classes and objects), and will assist in the program comprehension. This gives the maintainers and the developers the opportunity to be more productive in the process of understanding legacy systems. Moreover, understanding legacy system as a set of design patterns facilitates the communication between software designers, software developers, and software maintainers by using a common design patterns vocabulary.

The research findings confirm previous findings and contribute additional evidence that the detection of design patterns enhances the documentation process. Furthermore, the detection of design patterns facilitates the modifications process by adding new functionality. It also accelerates the software development process by grouping the software based on well-known patterns. Future research should, therefore, focus on the integration of the proposed framework with the software development process in a reengineering cycle. This will augment the software development and makes this approach more useful in facilitating the software reengineering tasks.

7.3 Future Research Directions

This research revealed many questions that needs to be investigated in further studies. The following is a summary for the possible open research directions:

- **Evaluation:** The approach was evaluated only on three systems. Those systems may not be sufficiently representative. It is, therefore, part of the future work to apply the proposed approach to a wider range of different software systems. Moreover, it would be interesting to extend the conducted evaluation experiments to evaluate the approach against all the GoF [54] design patterns.
• **Hint Templates:** The selection of hints was systematised by using a set hint-templates. However, hint-templates are not meant to be complete or perfect. In the future work, the top priority is to facilitate the addition of useful hints. It would be interesting to generate more useful hint templates. For example: collecting more hints from different expert users. This could be achieved by conducting another survey with more open questions. The participants of this study would be expected to select a specific pattern and use their knowledge to create hints.

• **Detection Workflow:** This research highlights the role of the user expert in the detection process. Further studies need to be carried out in order to create more concrete guidelines for the involvement of the user expert in the design pattern detection process, typically in the form of an organised workflow. This can be achieved by providing not only templates of frequent solutions, but also a straightforward approach for deciding which hints to select. Moreover, it would be of a great help to provide a facility that allow the users to navigate through the pattern candidates and go back to the code editor where the pattern is implemented.

• **Pattern Variants Catalogue:** As discussed in Section 5.5.1, each pattern can be represented by different code styles. It would be useful to provide a definition for the possible common pattern implementations variants. A greater focus on design pattern specifications and its implementations could produce interesting findings that account more for creating a catalogue of design pattern implementations variants. This will enhance the user awareness of different alternatives for each design pattern.

• **Empirical Directions:** The empirical study in Chapter 5 raised many questions that need further investigations. *(i)* Investigate whether the users want to use the framework with other source code related tools to facilitate their programming tasks; *(ii)* Some hints were considered as not being useful, it would
be interesting to investigate how these hints influence the user decisions. The investigation of these two questions will help to create a more concrete hint selection guidance.

- **Expert Users Population:** In future research, it would be interesting to select expert users from a well-known population (i.e. software students, developers, etc.). This will help to extract a more accurate generalisation from the acquired results.

- **Expressing Constraints:** The user feedback is provided in the form of hints. This feedback needs to take the form of additional constraints that can be added to the constraint solver. In further research, one might consider some other techniques used to support the user feedback such as *Aspect-Oriented Software Development (AOSD)* as a way of expressing constraints.

- **P-MARt Patterns:** Some new extra patterns are identified as true positive instances during the manual validation of the acquired results in the chosen systems in Chapter 5. These pattern instances need to be checked with P-MARt [59] team in order to verify if they are true positives or false positives.
Appendix A

Implementation

A.1 InFamix (.mse) file
Figure A.1: Example of .MSE file produced by InFamix
### Appendix F. Implementation

#### A.2 Basic-parser

```
Figure A.2: Example of Basic Parser output
```
Appendix F. Implementation

A.3 CSP Input

Figure A.3: Example of CSP input
Appendix B

Detection Constraints

B.1 Observer Design Pattern

Figure B.1: Observer Design Pattern
Observer Design Pattern:

super-class (observer-class))
super-class(subject-class))
concrete-class(concrete-Observer))
concrete-class(concrete-Subject))
abst-method(update-method))
override(concrete-Observer,update-method))
avbst-class(observer-class))
inb(concrete-Observer,observer-class))
inb(concrete-Subject,subject-class))
has-method(observer-class,update-method))
has-method(subject-class,notify-method))
has-method(subject-class,getState-method))
has-method(concrete-Observer,concreteObserver-methods))
asso(subject-class,observer-class))
asso(concrete-Observer,subject-class ))
invokes(notify-method, update-method))
invokes (concreteObserver-methods,getState-method))
Not(observer-class == concrete-Observer))
Not(observer-class == concrete-Subject))
Not(subject-class == concrete-Observer))
Not(concrete-Observer == concrete-Subject))
Not(subject-class == concrete-Subject))
Not(observer-class == subject-class))
Not(inh (observer-class,subject-class)))
Not(inh (subject-class,observer-class)))
B.2 State/Strategy Design Pattern

Figure B.3: State/Strategy Design Pattern

State/Strategy Design Pattern:

- super-class(state-strategy-role))
- inh(concrete-class-role,state-strategy-role))
- aggre(context-class-role,state-strategy-role))
- has-method(state-strategy-role,abstr-method))
- abst-method(abstr-method))
- override(concrete-class-role,abstr-method))
- Not(state-strategy-role == context-class-role))

Figure B.4: State/Strategy Pattern - Detection Constraints
B.3 Template Method Design Pattern

Figure B.5: Template Method Design Pattern

Template Method Design Pattern:

super-class(abstr-class))
concrete-class(conc-class))
override(conc-class, abstr-method))
inh(conc-class, abstr-class))
has-method(abstr-class, template-method))
has-method(abstr-class, abstr-method))
abst-method(abstr-method))
final-method(template-method))
invokes(template-method, abstr-method))

Figure B.6: Template Method - Detection Constraints
### Appendix E. Detection Constraints

#### B.4 Command Design Pattern

**Figure B.7: Command Design Pattern**

<table>
<thead>
<tr>
<th>Command Design Pattern:</th>
</tr>
</thead>
<tbody>
<tr>
<td>super-class(command)</td>
</tr>
<tr>
<td>sub-class(concreteCommand)</td>
</tr>
<tr>
<td>inh(concreteCommand, command)</td>
</tr>
<tr>
<td>has-method(command, execute-method)</td>
</tr>
<tr>
<td>abst-method (execute-method)</td>
</tr>
<tr>
<td>override(concreteCommand, execute-method)</td>
</tr>
<tr>
<td>aggre(invoker, command)</td>
</tr>
<tr>
<td>has-method(invoker, invoker-method)</td>
</tr>
<tr>
<td>invokes(invoker-method, execute-method)</td>
</tr>
<tr>
<td>asso(concreteCommand, receiver)</td>
</tr>
<tr>
<td>has-method(receiver, action-method)</td>
</tr>
<tr>
<td>has-method(concreteCommand, concreteCommand-method)</td>
</tr>
<tr>
<td>invokes(concreteCommand-method, action-method)</td>
</tr>
</tbody>
</table>

**Figure B.8: Command Pattern - Detection Constraints**
Appendix C

Hint Templates

C.1 Structural Hints
### Structural Hints

<table>
<thead>
<tr>
<th>Hint</th>
<th>Definition in Z3py API</th>
</tr>
</thead>
</table>
| Class $x$ and Class $y$ are never play the same role in the specified pattern. | $\text{Role}(x) = x \implies \text{Role}(y) = y$
| $\text{Role}(y) = y \implies \text{Role}(x) = x$ |
| Class $x$ must have at least one abstract/protected/final method. | $\text{Abstract-method}(m) \lor \text{Final-method}(m) \lor \text{Interface-method}(m) \lor \text{Protected-method}(m)$
| $\text{Class}(x) \implies \text{Has-method}(x,m)$ |
| Class $x$ is either abstract/Interface | $\text{Class}(x) \implies \text{Abstract-class}(x) \lor \text{Interface}(x)$ |
| Class $x$ must have at least one object of type $y$. | $\text{Class}(x)$; $\text{Class}(y)$; $\text{Obj}(x)$; $\text{Par-Declared}(y,v)$; $\text{Par-Paren}(x,v)$ |
| Class $x$ keep a reference to all concrete classes $x_1,...,x_n$. | $\text{Abstract-class}(x)$; $\text{Concrete-class}(x_1,...,x_n)$; $\text{Parameter}(p)$; $\text{Par-Declared}(x,p)$; $\text{Par-Paren}(x,p)$ |
| Class $x$ define a method $m_1$ invoking at least one abstract method defined in the same class. | $\text{Class}(x)$; $\text{Has-method}(x,m_1)$; $\text{Has-method}(x,m_2)$; $\text{Abstract-method}(m_2)$; $\text{Invoke}(m_1,m_2)$ |
| Classes $y_1,...,y_n$ override method $m_1$. | $\text{Abstract-class}(y)$; $\text{Abstract-method}(m)$; $\text{Has-method}(y,m)$; $\text{Concrete-class}(y_1,...,y_n)$; $\text{Has-method}(y_1,...,y_n,m_1)$; $\text{Override}(y_1,...,y_n,m_1)$; $\text{Same-signature}(m_1,m_1)$ |
| For a given pair of inheritance relationships, Class $x_1$ extends class $x$ and Class $y_1$ extends class $y$. These two pairs are said to be manually dependent if Class $y_1$ is never being as a concrete class of $x$, and Class $x_1$ is never being as a concrete class of $y$. | $\text{Super-class}(x)$; $\text{Super-class}(y)$; $\text{Unh}(y_1,y)$; $\text{Unh}(x_1,x)$; $\text{Not} \left( \text{Unh}(y_1,x) \right)$; $\text{Not} \left( \text{Unh}(x_1,y) \right)$ |
| Classes $y_1,...,y_n$ must keep reference back to class $x$. | $\text{Class}(y_1,...,y_n)$; $\text{Parameter}(p)$; $\text{Par-Declared}(x,p)$; $\text{Par-Paren}(y_1,...,y_n,p)$ |
| Method $m_1$ in class $x$ must be called in method $m_2$ in class $y$. | $\text{Class}(x)$; $\text{Has-method}(x,m_1)$; $\text{Class}(y)$; $\text{Has-method}(y,m_2)$; $\text{Invoke}(m_1,m_1)$ |
| Method $t_1$ should not be overridden. | $\text{Class}(x)$; $\text{Has-method}(t_1)$; $\text{Classes}(x_1,...,x_n)$; $\text{Unh}(x_1,...,x_n,x)$; $\text{Not} \left( \text{Override}(x_1,...,x_n,t_1) \right)$; $\text{Has-method}(x_1,...,x_n,t_2)$; $\text{Not} \left( \text{Same-signature}(t_1,t_2) \right)$ |
| Method $t_2$ invokes method $t_2$ in the same class. | $\text{Class}(x)$; $\text{Has-method}(x,t_1)$; $\text{Has-method}(x,t_2)$; $\text{Invoke}(t_1,t_2)$ |
| Class $x_1$ overrides a method $m_1$ that must call methods $m_2$ defined in the abstract class $x$. | $\text{Abstract-class}(x)$; $\text{Abstract-method}(m_1)$; $\text{Has-method}(x,m_1)$; $\text{Has-method}(x,m_2)$; $\text{Concrete-class}(x_1)$; $\text{Unh}(x_1,x)$; $\text{Has-method}(x_1,t_1)$; $\text{Override}(x_1,m_1)$; $\text{Same-signature}(t,m_1)$; $\text{Invoke}(t,m_2)$ |

**Table 1: Structural Hints**
## C.2 Sequential Hints

<table>
<thead>
<tr>
<th>Sequential Hints</th>
<th>Definition in Z3py API</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method ( t_1 ) must be called before method ( t_2 ) and must be terminated after ( t_2 ) in the same class.</td>
<td>( \text{Class}(x); \text{has-method}(x.t_1); \text{has-method}(x.t_2); \text{invoke}(t_1, t_2); \text{dominate}(t_1, t_2); )</td>
</tr>
<tr>
<td>Method ( t_1 ) must be called after method ( t_2 ) and must be terminated before ( t_2 ) in the same class.</td>
<td>( \text{Class}(x); \text{has-method}(x.t_1); \text{has-method}(x.t_2); \text{invoke}(t_2, t_1); \text{dominate}(t_2, t_1); )</td>
</tr>
<tr>
<td>Method ( t_1 ) must be called after method ( t_2 ) and must be terminated before ( t_2 ) in different classes.</td>
<td>( \text{Class}(x); \text{Class}(y); \text{has-method}(x.t_1); \text{has-method}(y.t_2); \text{invoke}(t_1, t_2); \text{dominate}(t_1, t_2); )</td>
</tr>
<tr>
<td>Method ( t_1 ) must be called before method ( t_2 ) and must be terminated after ( t_2 ) in different classes.</td>
<td>( \text{Class}(y); \text{Class}(y); \text{has-method}(x.t_1); \text{has-method}(y.t_2); \text{invoke}(t_2, t_1); \text{dominate}(t_2, t_1); )</td>
</tr>
<tr>
<td>Method ( m_1 ) must be always invoked before method ( m_2 ).</td>
<td>( \text{dominate}(m_1, m_2) )</td>
</tr>
<tr>
<td>Method ( m_1 ) must be always invoked after method ( m_2 ).</td>
<td>( \text{dominate}(m_2, m_1) )</td>
</tr>
<tr>
<td>Method ( t_1 ) in Class ( x ) must invoke method ( t_2 ) in Class ( y ) before class ( y ) invokes method ( t_3 ) in class ( z ).</td>
<td>( \text{Class}(x); \text{Class}(y); \text{Class}(z); \text{has-method}(x.t_1); \text{has-method}(y.t_2); \text{has-method}(z.t_3); \text{dominate}(t_1, t_2); \text{invoke}(t_3, t_2); \text{invoke}(t_2, t_3); )</td>
</tr>
<tr>
<td>Method ( t_1 ) in Class ( x ) must invoke method ( t_2 ) in Class ( y ) after class ( y ) invokes method ( t_3 ) in class ( z ).</td>
<td>( \text{Class}(x); \text{Class}(y); \text{Class}(z); \text{has-method}(x.t_1); \text{has-method}(y.t_2); \text{has-method}(z.t_3); \text{dominate}(t_1, t_3); \text{invoke}(t_2, t_1); \text{invoke}(t_2, t_3); )</td>
</tr>
<tr>
<td>Class ( x_1 ) overrides method ( m ) then calling method ( m_2 ) that belonging to class ( y ).</td>
<td>( \text{Abstract-class}(x); \text{class}(y); \text{concrete-class}(x_1); \text{Abstract-method}(m); \text{has-method}(x,m); \text{has-method}(y,m); \text{has-method}(x_1,m_1); \text{new}(x_1.x); \text{override}(x_1,m_1); \text{Same-signature}(m_1,m); \text{invoke}(m_1,m_2); \text{dominate}(m_2,m_2); )</td>
</tr>
</tbody>
</table>

Table 2: Sequential Hints
Appendix D

Survey Request and Questionnaire Form

D.1 Survey Request Letter

Dear Members,

We are conducting a survey on: Design Pattern Detection and User Knowledge as part of a research study that investigates expert users with GoF design patterns. The purpose of this survey is to study how an expert user can supply useful hints to assist the design pattern detection process.

I’ve invited you to fill out a form of a questionnaire. Kindly complete this questionnaire and at the end of each part, feel free to write your valuable comments on any problem or difficulty you found in answering the questions. This questionnaire will take approximately 20-25 minutes to complete. I really appreciate, in advance, your generosity of sharing your experiences, your time, and cooperation.

To access the questionnaire, please visit: https://www.surveymonkey.com/s/65FBCWP.

Please feel free to forward this link to anyone you think may be able to help.

Kind Regards,

Researcher
Appendix A. Questionnaire Form

D.2 Questionnaire Form

D.2.1 Introduction

Survey on Design Pattern Detection and User Knowledge

Thank you for participating in our survey.

As part of a research study, we are investigating expert users with GoF design patterns.

The purpose of this survey is to investigate how an expert user can supply useful hints in the design pattern detection process.

We begin by asking about your experiences with using design patterns. We then ask you to provide us with a little information about three selected scenarios.

We really appreciate your generosity of sharing your experiences and perspectives about design patterns.

This survey will take approximately 20 to 25 minutes to complete.
Appendix A. Questionnaire Form

D.2.2 Demographic

1. Contact Information:

   • Name: ..........................................
   • Email address: .............................

2. How do you rate your experience with design patterns?

   • Beginner
   • Good
   • Expert

3. What is your job role?

   • Software Developer
   • Researcher
   • Student
   • Other (please specify)
D.2.3 Selected Scenario

In this section you are asked to provide us with your assessment of the usefulness of some facts that support our suspicion of pattern instances in the source code. Based upon your experiences with using design patterns, please rate each fact as:

Very Negative
Negative
Not Significant
Positive
Very Positive

D.2.3.1 Scenario 1

We know the following:
JHotDraw provides menu items, which allow the user to tell the program what action to perform. Each choice in a menu bar is an instance of a CommandMenu class. This class provides a method execute(), which is an abstraction provided as a very simple entry point for the external classes (Clients) to use the concrete commands without knowing unnecessary details about them.

CommandButton class works as a command enabled button where the client selects a menu item by clicking the button to execute the command, it causes an action-Performed event. The method gets a selected menu item and the corresponding Command (CutCommand, copyCommand).

We suspect that the above characterises the behaviour of the Command pattern, consider the following facts and rate them according to the extent by which they support our suspicion.
Appendix A. Questionnaire Form

Please state your agreement level with each phrase.

1. Class commandMenu must have at least one final method.
   - Very Negative
   - Negative
   - Not Significant
   - Positive
   - Very Positive

2. Class commandButton holds a reference to an abstract class (commandMenu)
   - Very Negative
   - Negative
   - Not Significant
   - Positive
   - Very Positive

3. All Concrete commandMenus classes override a method execute().
   - Very Negative
   - Negative
   - Not Significant
   - Positive
   - Very Positive

4. Class copyCommand invokes all public methods in class DrawingView.
   - Very Negative
   - Negative
   - Not Significant
   - Positive
   - Very Positive

5. The client only need to know that object implementing command interface (commandMenu), have a simple method(execute()) to be called in order to consume it’s provided responsibilities.
   - Very Negative
   - Negative
   - Not Significant
   - Positive
   - Very Positive

6. Method actionPerformed() in Class commandButton must invoke method execute() in commandMenus classes before execute() invokes method selectionCount() in DrawingView class.
   - Very Negative
   - Negative
   - Not Significant
   - Positive
   - Very Positive
Appendix A. *Questionnaire Form*

7. Method `actionPerformed()` in Class `commandButton` invokes an abstract method in the same class.
   - Very Negative  
   - Negative  
   - Not Significant  
   - Positive  
   - Very Positive

8. All Concrete classes `commandMenus` override all the abstract methods of class `commandMenu`.
   - Very Negative  
   - Negative  
   - Not Significant  
   - Positive  
   - Very Positive

D.2.3.2 Scenario 2

We know the following:
In JUnit, the class `TestCase` defines the fixture to run multiple tests. It contains one or more tests, and groups together the tests that exercise the common behaviours. This class provides a method `run()` calling `setUp()` followed by `runTest()` and `tearDown()`. `TestCase` is declared abstract and follows a very specific sequence of events when invoking the tests. The execution of this sequence, however, will remain the same for all tests, no matter how the fixture code is written or how the testing code is written. First, it constructs a new instance of the test case for each test method. After constructing all of the test case objects, JUnit follows these steps for each test method:

- Call the test cases `setUp()` method.
- Call the `runTest()` method.
- Call the test cases `tearDown()` method.

These three methods contain a default implementation for each test, and they are intended to be overridden in the subclasses, so they were declared as protected methods.
Appendix A. Questionnaire Form

We suspect that the above is an instance of the Template Method pattern. Consider the following facts and rate them according to the extent by which they support our suspicion.

**Figure D.2: Questionnaire - Scenario 2**

Please state your agreement level with each phrase.

1. Class TestCase specifies a method that invoking at least one abstract/protected method defined in the same class.
   - Very Negative
   - Negative
   - Not Significant
   - Positive
   - Very Positive

2. Class TestCase must have at least one public/final method, and this method should not be overridden in the concrete classes.
   - Very Negative
   - Negative
   - Not Significant
   - Positive
   - Very Positive

3. The class TestCase must be declared abstract.
   - Very Negative
   - Negative
   - Not Significant
   - Positive
   - Very Positive

4. Method run() in TestCase class invokes method setUp() followed by runTest() and tearDown() in the same class.
   - Very Negative
   - Negative
   - Not Significant
   - Positive
   - Very Positive

5. All Concrete classes of TestCase should override all inherited abstract/protected methods.
   - Very Negative
   - Negative
   - Not Significant
   - Positive
   - Very Positive

6. All Concrete classes of TestCase class override at least one of the following methods. setUp(), runTest() and tearDown().
   - Very Negative
   - Negative
   - Not Significant
   - Positive
   - Very Positive
Appendix A. Questionnaire Form

7. Method run() in TestCase class invokes method runTest() defined in different class.

☐ Very Negative ☐ Negative ☐ Not Significant ☐ Positive ☐ Very Positive

8. Method run() in TestCase class invokes all the methods defined in the same class.

☐ Very Negative ☐ Negative ☐ Not Significant ☐ Positive ☐ Very Positive

D.2.3.3 Scenario 3

We know the following:

In QuickUML, there are two interfaces, Tool and ToolListener. These two interfaces interact on different aspects. The default implementation for the Tool interface is provided in the AbstractTool class. The AbstractTool implements the interface Tool and knows the ToolListener, and can have any number of ToolListener objects, and provides an interface for attaching (add) and detaching (remove) ToolListener objects. The ToolListener defines an update interface for objects that should be notified of changes in the Tool. This class provides two methods toolStarted() and toolFinished(). These two methods are called when a tool has reacted to an event, and when a tool has completed its work, and they are overridden in the derived classes.

We suspect that the above is an instance of the observer pattern, consider the following facts and rate them according to the extent by which they support our suspicion.

```
addToolListener(toolListener) {   
  listener.add(); 
}
removeToolListener(toolListener) {   
  listener.remove(); 
}
```

\[\text{AbstractTool} \rightarrow \text{Subject} \rightarrow \text{ToolListener} \rightarrow \text{Observer}\]

\[\text{addToolListener} \rightarrow \text{removeToolListener}\]

\[\text{toolStarted} \rightarrow \text{toolFinished}\]

Figure D.3: Questionnaire - Scenario 3
Appendix A. Questionnaire Form

Please state your agreement level with each phrase.

1. Method addToollistner() in the AbstractTool must call all the abstract methods in the AbstractTool class.
   - Very Negative
   - Negative
   - Not Significant
   - Positive
   - Very Positive

2. Class AbstractTool must have at least one object of type ToolListener.
   - Very Negative
   - Negative
   - Not Significant
   - Positive
   - Very Positive

3. Class AbstractTool invokes all the public methods in the ToolListener class.
   - Very Negative
   - Negative
   - Not Significant
   - Positive
   - Very Positive

4. All concrete observers (subclasses of ToolListener) must override at least one method belonging to its super class.
   - Very Negative
   - Negative
   - Not Significant
   - Positive
   - Very Positive

5. Method toolStarted() should invokes an abstract method in the same class.
   - Very Negative
   - Negative
   - Not Significant
   - Positive
   - Very Positive

6. Method in class AbstractTool class invokes method toolStarted() and toolFinished in ToolListener class.
   - Very Negative
   - Negative
   - Not Significant
   - Positive
   - Very Positive

7. All concrete observers (subclasses of ToolListener) should override all inherited abstract methods.
   - Very Negative
   - Negative
   - Not Significant
   - Positive
   - Very Positive

8. All concrete observers (subclasses of ToolListener) override toolStarted() and/or toolFinished() that belonging to the ToolListener class.
   - Very Negative
   - Negative
   - Not Significant
   - Positive
   - Very Positive

149
Bibliography


Bibliography


design patterns and software quality. GEODES–Research Group on Open, Dis-
tributed Systems, Experimental Software Engineering, University of Montreal.

In Knowledge-Based Intelligent Information and Engineering Systems, pages 384–
393. Springer.

populations and samples. ACM SIGSOFT Software Engineering Notes, 27(5):17–
20.

research in software engineering. Software Engineering, IEEE Transactions on,

structural design patterns in object-oriented software. In Reverse Engineering,

programming. In Proceedings of the 2002 ACM SIGPLAN workshop on Rule-
based programming, pages 1–14. ACM.

that aids reverse engineering. International Journal of security and applications,
2(1).

reverse engineering. In Software Engineering Research, Management & Applica-
IEEE.

[85] Likert, R. (1932). A technique for the measurement of attitudes. *Archives of psychology*.


