CODE PATTERN ANALYSIS OF OBJECT-ORIENTED PROGRAMMING LANGUAGES

by

XUBO MIAO

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Master of Science

Queen’s University
Kingston, Ontario, Canada
November 2016

Copyright © Xubo Miao, 2016
Abstract

Code patterns, including programming patterns and design patterns, are good references for programming language feature improvement and software re-engineering. However, to our knowledge, no existing research has attempted to detect code patterns based on code clone detection technology. In this study, we build upon the previous work and propose to detect and analyze code patterns from a collection of open source projects using NiPAT technology. Because design patterns are most closely associated with object-oriented languages, we choose Java and Python projects to conduct our study. The tool we use for detecting patterns is NiPAT, a pattern detecting tool originally developed for the TXL programming language based on the NiCad clone detector. We extend NiPAT for the Java and Python programming languages. Then, we try to identify all the patterns from the pattern report and classify them into several different categories. In the end of the study, we analyze all the patterns and compare the differences between Java and Python patterns.
Acknowledgments

First of all, I would like to express my thanks to my professor Dr. James R. Cordy for all his guidance and support during my study in Queen’s University.
I would like to thank my friend Ashiqur M. Rahman for sharing his pattern detecting tool NiPAT and all his help during my research.
I also want to express my gratitude to my family for all their support and encouragement.
Statement of Originality

I hereby certify that the research presented in this dissertation is my own, conducted under the supervision of Dr. James R. Cordy. Ideas and techniques that are not a product of my own work are cited, or, in cases where citations are not available, are presented using language that indicates they existed prior to this work.
# Contents

Abstract i

Acknowledgments ii

Statement of Originality iii

Contents iv

List of Tables vii

List of Figures viii

Chapter 1: Introduction 1

1.1 Motivation ........................................ 2
1.2 Contributions ...................................... 3
1.3 Outline of the Thesis ............................... 4

Chapter 2: Background 5

2.1 Code Clones ......................................... 5
2.1.1 Definition of Code Clones ....................... 6
2.1.2 Four Types of Code Clones ..................... 6
2.2 NiCad: Clone Detector ............................ 9
2.2.1 TXL Programming Language .................... 10
2.2.2 How NiCad Works ............................... 13
2.2.3 How to Use NiCad ............................... 20
2.3 NiPAT: Pattern Detector ........................... 21
2.3.1 How NiPAT Works ............................... 22
2.4 Summary ............................................ 30

Chapter 3: Overview 31

3.1 Experimental Setup ............................... 34
3.1.1 Choice of Open Source Projects ................. 34
3.1.2 NiCad Clone Detector Settings ........................................ 35
3.1.3 Project collections for the Study ..................................... 35
3.2 Extending NiPAT ................................................................. 35
  3.2.1 A Class Extractor for NiCad ........................................... 36
  3.2.2 Fixing Broken XML Files from NiCad ............................... 36
  3.2.3 Pattern Detection Changes ........................................... 36
  3.2.4 Changes to the Pattern Report ...................................... 37
3.3 Pattern Detection and Analysis ......................................... 37
  3.3.1 Extracting files from Java and Python projects .................. 38
  3.3.2 Detecting code clones using NiCad ................................. 38
  3.3.3 Detecting patterns using extended NiPAT ......................... 38
  3.3.4 Pattern analysis ....................................................... 39
3.4 Summary ........................................................................ 39

Chapter 4: Extending NiPAT ................................................. 40
  4.1 Adding Class Extractor for NiCad ....................................... 40
  4.2 Fixing Broken XML Files Clone Report ................................ 41
  4.3 Changes to Pattern Detection ............................................ 43
  4.4 Changes to the Pattern Report .......................................... 45
     4.4.1 Exporting a summary report of patterns found ................ 45
     4.4.2 Exporting individual Pattern Reports ......................... 47
     4.4.3 Adding a Hyperlink to the Source File ....................... 47
  4.5 Summary ....................................................................... 50

Chapter 5: Experimental Setup .............................................. 51
  5.1 Choosing Appropriate Open Source Projects ....................... 51
     5.1.1 Java Projects in Our Study ...................................... 52
     5.1.2 Python Projects in Our Study .................................... 53
  5.2 Settings in the NiCad Clone Detector ............................... 53
     5.2.1 Clone Granularities ............................................... 53
     5.2.2 Maximum difference threshold .................................. 54
     5.2.3 The Size of Clones ............................................... 55
     5.2.4 Code Normalization ............................................. 56
     5.2.5 Code clone report ............................................... 57
  5.3 Settings in NiPAT Pattern Detector ................................. 58
     5.3.1 File Extractions ................................................... 58
     5.3.2 Number of Files in Set .......................................... 58
  5.4 Summary ...................................................................... 59

Chapter 6: Pattern Analysis .................................................. 60
  6.1 Patterns in Java Projects .............................................. 61
List of Tables

5.1 The influence of Threshold for Java projects . . . . . . . . . . . . . . 55
5.2 The influence of threshold for Python projects . . . . . . . . . . . . . 55
5.3 The influence of normalization (Blind Renaming) for Java projects . . 57
5.4 The influence of normalization (Blind Renaming) for Python projects 57
5.5 The influence of filtering No. of files in set for Java projects . . . . . 59

6.1 All factory method patterns detected from Java projects . . . . . . . 67
6.2 All abstract factories detected from Java projects . . . . . . . . . . . 71
6.3 A summary of all Java design patterns under different thresholds . . 98
6.4 A summary of all emergent Java patterns under different thresholds . 99
6.5 A summary of all Python design patterns under different thresholds . 126
6.6 A summary of all emergent Python patterns under different thresholds 127
List of Figures

2.1 An example of a Type I Clone in the Java programming language . . 7
2.2 An example of a Type II Clone in the Java programming language . 7
2.3 An example of a Type III Clone in the Java programming language . 8
2.4 An example of a Type IV Clone in the Python programming language 9
2.5 TXL process (Adapted from TXL Cookbook [22], Figure 1) . . . . . . 10
2.6 A TXL calculator (Adapted from the Guided Tour of TXL [5]) . . . 12
2.7 The TXL execution process (Adapted from the Guided Tour of TXL [5]) 13
2.8 The Nicad process (Adapted from Nicad Clone Detector [23]) . . . 14
2.9 An Example of two code fragments used for clone detection . . . . 15
2.10 Code fragments after parsing stage . . . . . . . . . . . . . . . . . . . 16
2.11 Code fragments after normalization stage . . . . . . . . . . . . . . 18
2.12 An example of clone report . . . . . . . . . . . . . . . . . . . . . . . 19
2.13 The whole working process of pattern detection . . . . . . . . . . 22
2.14 The program execution of NiPAT during the first stage . . . . . . . 23
2.15 An summary report for TXL grammar file retrieval . . . . . . . . 24
2.16 Two options for detecting clone classes . . . . . . . . . . . . . . . 25
2.17 Broken tags in Nicad clone report . . . . . . . . . . . . . . . . . . . 26
2.18 Parsing tree generated by NiPAT [30] . . . . . . . . . . . . . . . . . 27
2.19 Possible file combinations during pattern detection [30] . . . . . . 28
2.20 An example of pattern report - Part 1 [29] ................. 29
2.21 An example pattern report - Part 2 [29] ................. 30

3.1 The whole process of our study ............................ 32

4.1 An example of broken XML files ......................... 42
4.2 Fixing broken XML tags - Part 1 ......................... 43
4.3 Fixing broken XML tags - Part 2 ......................... 43
4.4 An example of pattern report with function names only .... 44
4.5 An example of pattern summary report (Threshold = 5) .... 46
4.6 An example of the file directory hyperlink ................ 48
4.7 The “AntClassLoader.java” file appears after click the hyperlink ... 49

5.1 An example of Java compiler generated patterns ............ 56

6.1 An example of factory method pattern from pattern report .... 66
6.2 An example of abstract factory pattern from pattern report .... 70
6.3 An example of builder pattern from pattern report .......... 74
6.4 An example of adapter pattern from pattern report .......... 77
6.5 An example of parser generated pattern from pattern report .... 83
6.6 An example of class polymorphism pattern from pattern report ... 90
6.7 An example of data operation pattern from pattern report .... 97
6.8 An example of singleton pattern from pattern report .......... 102
6.9 An example of decorator pattern from pattern report .......... 106
6.10 An example of file handling pattern from pattern report ....... 110
6.11 An example of data operation pattern from pattern report .... 114
6.12 An example of package management pattern from pattern report ... 118
6.13 An example of testing and checking pattern from pattern report . . . 122
6.14 An example of API pattern from pattern report . . . . . . . . . . . 125
Chapter 1

Introduction

Programming patterns, including both coding patterns [20] and design patterns [34] are good references for programming language feature improvement and software re-engineering. NiPAT [30], a code pattern detection tool developed by Ashiqur M. Rahman and James R. Cordy, is an efficient approach to detect TXL code patterns. The tool was originally developed for improving coding features of the TXL programming language, but it has never been used for other languages. Therefore, we propose to extend the utilization of NiPAT to some popular programming languages and try to find some useful code patterns.

In our study, we continue the previous work[30] and propose to apply NiPAT to some open source projects which are implemented by object-oriented languages Java and Python. NiPAT was originally developed for detecting code patterns from TXL programs, which are relatively less complicated than Java and Python projects. Therefore, we have to extend NiPAT for our study before applying it to code pattern detections in Java and Python projects.

The thesis consists of three parts. First, we introduce the extensions to NiPAT to make it suitable for our project. Second, we illustrate all the different experimental
settings of our study to get better pattern reports for the next stage. In the end, we categorize and analyze all the code patterns and compare the differences between code patterns found in Java and Python projects.

1.1Motivation

Software design patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system [11]. Researchers have proposed different ways to identify design patterns from open source projects. Tsantalis, Nikolaos, et al. [37] proposed a similarity scoring algorithm to detect Java design patterns from Java projects. Stoianov, Alecsandar [35] proposed a methodology to detect Java design patterns using Prolog rules. However, no previous research has been done using code clone techniques.

Recently, Rahman and Cordy [30] have developed a tool called NiPAT to detect TXL programming patterns for the purpose of improving TXL programming features based on code clones observed in TXL programs. This technique works very well on TXL programs, and has uncovered some useful coding patterns. Therefore, we propose to extend the use of NiPAT to object-oriented programming languages to search for object-oriented patterns, including both known design patterns and previously unknown emergent patterns.

In our project, we continue Ashiqur Rahman’s previous work [30]. Ashiqur developed a pattern detector tool called NiPAT, which is being used to detect TXL programming patterns for the purpose of improving coding features of the TXL programming language. NiPAT is a pattern detector tool based on the NiCad code clone detector. It takes the code clone report as input and detects the multi-appearance of
1.2. CONTRIBUTIONS

Since it is an efficient tool to detect programming patterns, we propose to apply it to other projects and try to find some interesting code patterns including both design patterns and emergent code patterns that frequently appear in the pattern report. Considering that Java and Python are the top two popular programming languages and the theory of design patterns is based on object-oriented programming languages, we choose open source projects in the Java and Python languages to conduct our empirical study. If some useful patterns are found, we may take them as feature recommendations to programmers.

1.2 Contributions

First, we extend the use of NiPAT to Java and Python projects and these extensions will be covered in Chapter 4.

Second, besides detecting some design patterns in Java and Python projects, we also identify some new emergent code patterns that appear frequently in pattern reports, such as data operation patterns and file operation patterns. The pattern analysis including design patterns and emergent code patterns for both Java and Python projects will be introduced in Chapter 6.

Last but not the least, we also compare the difference between Java patterns and Python patterns. For example, design patterns are more widely used in Java projects compared to Python projects. Apart from that, the size of the code patterns from Python projects are relatively smaller and less complicated compared to Java projects. More comparisons will be shown in Section 6.3.
1.3 Outline of the Thesis

In Chapter 2, we introduce the background knowledge of the project, including the definition of code clones, the NiCad code clone detector and the NiPAT code pattern detector. A brief overview of our work is covered in Chapter 3. In this chapter, we illustrate the whole process of the study, which consists of three main stages. The first stage of our study is to extend the existing NiPAT for Java and Python projects, described in Chapter 4. It is followed by a discussion of our experimental settings in Chapter 5. The last stage of our study is to analyze and compare the patterns we get from the pattern reports, in Chapter 6. In Chapter 7, we conclude all the findings and point out the limitations and the future work of our study.
Chapter 2

Background

This chapter provides background for our study. There are three aspects we want to discuss in the following sections. In Section 2.1, we describe the definition of code clones and four different types of code clones with examples. After that, the code clone detector NiCad that we use in our study is introduced in Section 2.2. Finally, NiPAT, the code pattern detector we use for our study is covered in Section 2.3.

2.1 Code Clones

Code pattern detection is actually detecting the co-appearance of code clone classes from different files. Thus, code pattern detecting is based on code clones. In order to give a better understanding of clone detections, we firstly introduce the definition of code clones in Section 2.1.1. Then, we give an overview of four different types of code clones, namely type-I, type-II, type-III and type-IV [33] code clones in Section 2.1.2.
2.1. CODE CLONES

2.1.1 Definition of Code Clones

In general, the definition of code clone is a code fragment is exactly the same as or similar to the other code fragment. However, there is not a universal definition for code clones mainly because different methods are used to detect code clones and each has its own definitions and threshold to decided whether two code fragments are clone or not [28]. For example, in our thesis, we use the NiCad clone detector to find code clones, using thresholds (the maximum percentage of code difference) from 0.1 (10%) to 0.3 (30%) difference. We consider clone fragments with the maximum threshold 0.3 as code clones in our study.

2.1.2 Four Types of Code Clones

There are basically two different kinds of code similarities, which are textual and functional similarities. Textual similarity means two code fragments are similar in their program texts. The code fragments could be an exact copy of the other one or a copy just with some minor changes of the identifiers. Functional similarity means two code fragments implement similar or same functions but in different ways. Among the following four type of code clones, Type I, Type II and Type III clones are defined based on code textual similarities. Type IV clone is defined based on code functional similarities.

Type I Clone: It is the simplest one among all these four types of code clones because the code fragment is just the exact same clone of another one. There are maybe some differences in spaces or lines but the actual text part is exactly the same between two code fragments [33]. We provide an example of a Type I Java code clone in Figure 2.1.
2.1. CODE CLONES

(a) Original code fragment - Type I Clone

(b) Cloned code fragment - Type I Clone

Figure 2.1: An example of a Type I Clone in the Java programming language

From the two code fragments above, we can find that although there are some minor differences (blanks, new lines, comments etc.) between them, the expressions and functions are exactly the same. Both of them are implementing the same output functions under certain conditions.

**Type II Clone:** This kind of clone is more general than the Type I Clone. Instead of completely copying the original code fragment, there are some changes in the names of identifiers (variables, constants, class etc) or the values of variables between them [33]. However, the textual structure of the code is still the same as the original code fragment. The following two code fragments in Figure 2.2 are examples of Type II Clone.

(a) Original code fragment - Type II Clone

(b) Cloned code fragment - Type II Clone

Figure 2.2: An example of a Type II Clone in the Java programming language
2.1. CODE CLONES

In this example, the code fragments still look similar textually. The differences between the two code fragments are the variable “a” is renamed by “b” and the value is also changed. However, the whole structure and function of the program has not changed.

**Type III Clone:** Based on the two types of code clones above, there can be more modifications in a Type III code clone. Apart from some changes with name and value of the identifiers, some statements may also be changed a little bit [33]. The cloned code fragment may add or delete some statements from the original code. The following Figure 2.3 is an example of Type III clone. Not only the identifiers are changed in the cloned code fragment, but also a new statement “b = b + 1” is added.

```
int a = 3;
if (a < 1) {
    System.out.println("a is less than 1");
}
else if (a > 1 && a < 5) {
    System.out.println("a is between 1 and 5");
}
else {
    System.out.println("a is bigger than 5");
}
```

(a) Original code fragment - Type III Clone

```
int b = 10;
if (b < 1) {
    b = b + 1;
    System.out.println("b is less than 1");
}
else if (b > 1 && b < 5) {
    System.out.println("b is between 1 and 5");
}
else {
    System.out.println("b is bigger than 5");
}
```

(b) Cloned code fragment - Type III Clone

Figure 2.3: An example of a Type III Clone in the Java programming language

**Type IV Clone:** This is the most complicated code clone among all these four types. The functionalities could be identical or similar between two code fragments, however, the ways to implement the functions are different [33]. For instance, basically, there are five different kinds of sorting algorithms, quick sort, insertion sort, bubble sort, merge sort and selection sort. All these five algorithms implement the same function, however, the implementations are quite different.
2.2. NICAD: CLONE DETECTOR

We provide two different sorting algorithms implemented in Python programming language as follows which could be considered as Type IV clone in Figure 2.4. The first code fragment is the implementation of selection sorting algorithm using “for” loops and the second one is the implementation of insertion sorting algorithm using “while” loops. Both of the code fragments are able to sort the given arrays. However, the way to implement the sorting algorithm are very different which is hard to detect whether these code fragments are cloned or not. In our study, we use NiCad to do the code clone detection, and it supports code clone detections of Type I, Type II and Type III clones. Thus, type IV code clones are not included in our study.

2.2 NiCad: Clone Detector

Code clone detection is the very first step for pattern detection. Therefore, a good understanding of NiCad [32] [23], the code clone tool we use, is necessary for our study. In this Section, we firstly make a brief introduction to the TXL Programming language which is used to implement the NiCad clone detector in Section 2.2.1. The basic concepts and the working principles of NiCad are described in Section 2.2.2.
Finally, we illustrate how to use NiCad to do the clone detection in Section 2.2.3.

2.2.1 TXL Programming Language

TXL is a unique programming language specifically designed to support computer software analysis and source transformation tasks [24]. For instance, it is a very efficient tool for code migration from one language to another different language. In the following Figure 2.5, We provide an example to better illustrate how TXL works.

![TXL process diagram](image)

Figure 2.5: TXL process (Adapted from TXL Cookbook [22], Figure 1)

Firstly, the input text “blue fish” which is parsed into a parse tree by TXL program. Then, the program transforms the input parse tree into another parse tree which is consist of the new text “marlin” instead. Finally, the tree is unparsed and output the text “marlin”. The whole working process could be conclude into “parse, transform and unparse”. Every Txl program consists of two parts, grammar definitions and transformation rules in order to implement the whole process in the example.

The following Figure 2.6 is an implementation of a simple calculator in TXL,
which is able to implement “add”, “minus”, “multiply” and “divide” operations. At
the beginning, the grammars are all defined in the first part for the parsing stage and
then all the expressions are replaced by the NewE in the transformation stage and
output the result. The following Figure 2.7 shows the working process of the whole
program.
2.2. NICAD: CLONE DETECTOR

(a) Grammar definition

(b) Transformation rule

Figure 2.6: A TXL calculator (Adapted from the Guided Tour of TXL [5])
2.2. NICAD: CLONE DETECTOR

2.2.2 How NiCad Works

The NiCad Clone Detector [32][23] is a clone detection tool which is designed for detecting near-miss intentional clones using a hybrid structural/textual technique. The tool has been shown to perform with both high precision and high recall in the detection of near-miss intentional clones [36]. The process of the code clone detection with NiCad can be divided into three stages: parsing, normalization and comparison (Figure 2.8).
In Figure 2.8, the original code base is taken by the extractor of NiCad in the first stage. Then, the pretty-printed code fragments are normalized in the second stage. In this stage, NiCad users are able to rename, filter and normalize the code for comparison, according to their own needs. Finally, all the code fragments are compared line by line and clustered into several clone classes. The final output of the NiCad clone detector is a set of clone reports of all the clone classes in both XML and HTML formats.

For a good understanding of all the three working stages of NiCad, a Java example program of insertion sorting algorithm is provided as follows in Figure 2.9. Both of the two fragments are the implementations of the same algorithm, however, some identifiers and comments are slightly different. We will illustrate how NiCad Clone Detector works by doing clone detection in function granularity using these two code fragments.
Figure 2.9: An Example of two code fragments used for clone detection

**Parsing:** In this stage, all codes are broken into the given granularity, either block or function. In order to increase the precision of clone detection, all the comments and extra spaces or tabs are removed from the original code fragments. The code fragments after parsing are shown in Fig 2.10 as an XML file.
From the example above, four nice-printed functions are extracted from the original files because we set the granularity to function. Also, each function is labelled with the directory, the number of start line and end line of code fragments. Then
these code fragments will be normalized in the next stage.

**Normalization:** In this stage, we are able to do some code normalizations after parsing. Users are able to either write their own normalization files in TXL or just use the default normalization options from NiCad. In our study, we make use of the blind renaming normalization to get more different patterns to analyze. The code fragments after blind renaming could be found in Figure 2.11 and all function names and identifiers are blinded. It is apparent that the possibilities of finding code clones will be increased after applying blind renaming.
2.2. NICAD: CLONE DETECTOR

Comparison: The code fragments we got from the two stages above are compared line by line using an optimized LCS algorithm in comparison stage. The clone pairs will be detected and printed with the given threshold. The threshold limits the
maximum difference between two clone pairs. For instance, if threshold is set to 0.2, only clone pairs with 80% similarity or more will be left after the comparison. Then clone pairs will be clustered into several clone classes according to the similarities. Finally, NiCad will export a clone report (Figure 2.12).

**NiCad Clone Report**

System: Example  
Granularity: functions-blind  
Max difference threshold: 30%  
Clone size: 5 - 1500 lines  
Total functions-blind: 4  
Clone pairs found: 2  
LCS compares: 2  
CPU time: 0 min 0.28 sec  
Number of classes: 2  

**Clone class 1, 2 fragments, nominal size 8 lines, similarity 87%**

```java
public static void main(String a[]){
    int[] arr1 = {10,34,2,56,7,67,88,42};
    int[] arr2 = doInsertionSort(arr1);
    for(int i=arr.length;
        System.out.print(i);
        System.out.print(" ");
    }
}
```

```java
public static void main(String b[]){  
//Comment 1
    int[] arr1 = {1,3,2,6,7,8,4};
    int[] arr2 = doInsertionSort(arr1);
//Comment 2
    for(int j=arr.length;
    System.out.print(j);
    System.out.print(" ");
    }
```  

Figure 2.12: An example of clone report

In this report, some basic configurations could be found at the beginning of the
2.2. NICAD: CLONE DETECTOR

report. For example, granularities and max difference threshold. After that, a brief summary of the clone detection is listed, such as number of clone pairs and clone classes. The clone classes are listed in the third part. Besides, we are also able to identify the normal size of the code fragment and the similarities between several clone pairs.

All the parameters for these three stages could be changed in configuration files with the extension “.cfg” in NiCad and users are able to change the value of each parameter in configuration files according to their needs.

2.2.3 How to Use NiCad

NiCad can be used with simple line commands on several different platforms. NiCad users need to enter the granularities they want to extract from the source code, the language needed to parse the directory of source files and the name of the configuration file in the following format:

./nicad granularity language directory [config]

For instance, if we want to do a function granularity clone detection for java project under the directory “examples/javafiles” using the “default” configuration, the command line will be as follows:

./nicad4 functions java examples/javafiles default

In the next four paragraphs, we will provide some details about all the five parts of the command.

Invoking: First of all, the Nicad Clone Detector must be invoked at the beginning of the command line. In the example above, “./nicad4” means Nicad Clone Detector Version 4 is invoked for clone detection.
Granularity: Clone Detections in block and function granularities are supported in NiCad. However, users are able to add their own code extractor. In our study, we implement a class extractor for Java language in order to find the class-based design patterns and more information about the class extractor will be provided in Section 4.1. The extractor files are all named in “language-extract-granularity.txl” format under the folder txl. Thus, the class extractor we have is named “java-extract-granularity.txl”

Programming Languages: The C, C#, Python and Java programming languages are currently supported by NiCad. However, if a programming language is not supported in NiCad, users are also able to define the grammar of a language and add it as a TXL plugin to support other languages.

Directory: NiCad searches all directories in the source file path it is given to find source files in the language specified.

Configuration: There are many options that can be changed. For example, threshold, size of code fragments and some other code normalizations.

2.3 NiPAT: Pattern Detector

NiPAT [30], a code pattern detector originally designed for improving the features of the TXL programming language by Ashiqur Rahman in his MSc thesis [29]. It provides a full environment for the whole process of the TXL code pattern detection including source file extraction, clone detection and pattern detection. In the following Section 2.3.1, we will introduce the whole working process of NiPAT.
2.3.1 How NiPAT Works

The whole working process of NiPAT can be divided into four stages (Figure 2.13). First, NiPAT collects all the source files with the given extensions for code clone detection. Second, NiCad is invoked for code clone detection in the second stage. And in this stage, the identical and similar code fragments are clustered into several clone classes and exported as a clone report in both XML and HTML formats. Third, before taking the clone report as an input for pattern detection, some broken XML tags are needed to be fixed. Finally, NiPAT takes the fixed clone report as input and generates all the patterns. More details about each stage will be provided in the following four sections.

Figure 2.13: The whole working process of pattern detection
2.3. NIPAT: PATTERN DETECTOR

Source File Collection

NIPAT is originally designed for detecting code patterns for TXL programming language. And all the TXL projects are consist of two parts, grammars and transformation rules as mentioned in Section 2.2.1. Thus, the first stage of the code pattern detection is to extract the source files needed for the pattern detection and there are three available options in NIPAT “Grammar Only”, “Rule Only” and “Grammar and Rule”.

In most cases, TXL grammars and transformation rules are written in different files. Also, programmers are allowed to give any file extension names to grammar and rule files even for the same TXL project. Therefore, files are extracted according to the file extensions that are provided and different file extensions could be entered with spaces between them as seen in Figure 2.14.

In this example, we use three TXL projects downloaded from TXL website [17]. The source program directory path should be entered at the beginning as seen in Figure 2.14. Then, users are able to choose whether to extract grammar, rule files
or both in the next step. For this example, we only extract grammar files with the extensions "txl, Txl, and Grammar". As a result, 13 files are found and they are all copied into a folder called "GRMfiles".

--- Summary Report on TXL Grammar File Retrieval ---
Grammar Extension searched for: ['.txl', '.Txl', '.Grammar']

Retrieved File Stored in: C:\txl_example\GRMfiles
- Original Source File Path
  explicatecontains.gzm <- C:\txl_example\Txl_projects\ROX\Txl\ExplicateContains.txl <CopySuccess>
  explicatesequences.gzm <- C:\txl_example\Txl_projects\ROX\Txl\ExplicateSequences.txl <CopySuccess>
  generateTags.gzm <- C:\txl_example\Txl_projects\ROX\Txl\GenerateTags.txl <CopySuccess>
  getCleanTags.gzm <- C:\txl_example\Txl_projects\ROX\Txl\GetCleanTags.txl <CopySuccess>
  getcreateStatements.gzm <- C:\txl_example\Txl_projects\ROX\Txl\GetCreateStatements.txl <CopySuccess>
  getdbnameiform.gzm <- C:\txl_example\Txl_projects\ROX\Txl\GetDBInXMLForm.txl <CopySuccess>
  getinsertstatements.gzm <- C:\txl_example\Txl_projects\ROX\Txl\GetInsertStatements.txl <CopySuccess>
  getspecificTags.gzm <- C:\txl_example\Txl_projects\ROX\Txl\GetSpecificTags.txl <CopySuccess>
  marktags.gzm <- C:\txl_example\Txl_projects\ROX\Txl\MarkTags.txl <CopySuccess>
  sql.gzm <- C:\txl_example\Txl_projects\ROX\Txl\SQL.Grammar <CopySuccess>
  xml_element_v8.gzm <- C:\txl_example\Txl_projects\ROX\Txl\XML_Element_v8.Grammar <CopySuccess>
  pic.gzm <- C:\txl_example\Txl_projects\SDDS\Txl\pic.Txl <CopySuccess>

Total file found: 13
Copied Successfully: 13
Same File found: 0
Unknown Copy Error: 0

Figure 2.15: An summary report for TXL grammar file retrieval

A brief summary report on TXL Grammar File Retrieval is also generated during the file extraction process in Figure 2.15. Apart from the information we got during the execution of the program such as "total files found" and "same files found", all the file directories are also shown in the report. It is easy to find and check the source files using the summary report.

Clone Detection

In this stage, NiCad is used for doing code clone detection. NiCad is currently available for installation as a command-line tool on Linux, Mac OS X, and Cygwin only but NiPAT was developed based on Windows operating system. Therefore, two options are provided in NiPAT for clone detection in the following Figure 2.16.
Use NiCAD command to generate clones...
Press:
1> Minimize NiPAT and run NiCAD separately on collected files.
2> Run NiCAD in Cygwin environment in windows.

Collected files have been stored in  C:\txl_example\GRMfiles
Minimize NiPAT and return after running NiCAD on collected files, and press enter...

Figure 2.16: Two options for detecting clone classes

One option is “Minimize NiPAT and run NiCAD separately on collected files”. By choosing this option, users are able to run NiCad separately by using the collected files as the input for clone detection. The other option is “Run NiCAD in Cygwin environment in Windows”. Cygwin virtual machine will be invoked for running NiCad if users go for this option. It does not matter which method is chosen because only the NiCad clone report in XML format is needed for the next stage.

Fixing Broken XML Files

After the code clone detection stage, we get two clone reports in both HTML and XML formats from NiCad. NiPAT is only able to parse the XML files for the next stage but there are some symbols that are illegal or reserved for XML tags (i.e. “<”) which are needed to be replaced by some other symbols just as shown in Fig 2.17 before detecting code patterns.
2.3. NIPAT: PATTERN DETECTOR

The source code between the first “<source>” and the last “</source>” will be parsed by NiPAT. However, there are also another two “<” tags in the source code that we highlighted in Figure 2.17 which are reserved by XML. If we just use this file for the next stage, a grammar error will occur when the file is parsed by NiPAT. Thus, all the “<” tags are replaced by “&lt”. Apart from that, there are also some other symbols that are conflict with the XML grammar. And more details will be included in Section 4.2.

**Pattern Detection**

After all the XML tags in the NiCad clone report are fixed, NiPAT takes the fixed report as input and generates the parse tree as seen in Figure 2.18 [30]. Then, NiPAT will detect all the unique files which have several clone classes in common from the parse tree and find all the possible combinations of source files. In this example, there...
are four possible file combinations between files “F1, F2 and F3”.

Figure 2.18: Parsing tree generated by NiPAT [30]
2.3. NIPAT: PATTERN DETECTOR

(a) File combination (F1, F2)

(b) Cloned code fragment (F2, F3)

(c) Original code fragment (F1, F3)

(d) Cloned code fragment (F1, F2, F3)

Figure 2.19: Possible file combinations during pattern detection [30]

After that, NiPAT will map four different file sets one by one through the parsing tree as seen in Figure 2.19 [30]. Then a pattern will be formed if there are some clone classes in common from each file set. Finally, a pattern report will be generated as
seen in Figure 2.20 [29] and Figure 2.21 [29].
### 2.4 Summary

In this Chapter, we first introduced the definitions of code clones and gave an overview of all the four types of code clones. Then, we introduced the code clone detector NiCad, including a brief introduction of the TXL programming language, the way that NiCad works and how to use NiCad. Finally, we described the working process of the code pattern detector NiPAT.
Chapter 3

Overview

We introduced four different categories of code clones, the NiCad code clone detector and the NiPAT code pattern detector we used as the background knowledge in Chapter 2. However, the code patterns generated from the previous work [30] are all in the TXL programming language, which is a functional language used by only a few hundred users for specific purposes. Therefore, we propose to apply NiPAT to some projects implemented in more popular and widely used programming languages to explore more general code patterns. After careful consideration, we chose open source Java and Python projects to study code patterns, including design patterns and emergent code patterns using NiPAT. Our study can be divided into three stages as seen in Figure 3.1.

Stage 1: We detect code clones in the first stage using the NiCad clone detector and get the code clone report for the next stage. During this stage, we propose to detect code clones in both class and function granularities mainly because most design patterns are class or function based patterns. However, only clones at function and block granularity are supported in NiCad. Thus, we add a class extractor to NiCad in order to get a code clone report at class granularity for the NiPAT pattern detector.
Stage 2: The second stage of the experiment is to detect code patterns from the code clone report we get from the first stage, using NiPAT technique. As we mentioned before, NiPAT has been only applied to TXL programming language,
which is a relatively simple language compared to either Java or Python. And the code patterns generated from TXL projects are generally short and small. However, in this study, we propose to apply NiPAT to Java and Python projects, which have relatively larger and more complicated functions compared to TXL projects. Therefore, in order to better analyze these patterns, we also extend NiPAT to make it more suitable for Java and Python projects before pattern detections. We not only did some changes to the process of pattern detecting, such as fixing broken XML files and copying function names instead of copying the whole functions into the pattern report, but also added some changes to the pattern report such as exporting a summary report of all the patterns, adding hyperlinks to the source files and generating individual pattern reports. All the extensions of NiPAT will be further introduced in Chapter 4.

**Stage 3:** The last stage is to categorize and analyze all the patterns reported in the pattern reports. During this stage, we identify patterns one by one and try to categorize them into two main types, known design patterns, and new emergent code patterns. Apart from categorizing and analyzing these patterns from NiPAT pattern reports, we also compare the difference between Java and Python patterns. More details of this stage will be covered in Chapter 6.

The following paragraphs detail the stages of our experiment. First, we describe how we set up the whole experiment in Section 3.1. And then we introduce the extending of NiPAT to better apply it to Java and Python projects in Section 3.2. Finally, we introduce how we conduct the experiment and analyze the patterns in Section 3.3.
3.1 Experimental Setup

All the settings of the empirical study will be covered in this section, including choice of projects, NiCad clone detector settings, and the project collections we use in our study.

3.1.1 Choice of Open Source Projects

In order to find some interesting code patterns or design patterns, we decided to use two popular object-oriented languages, Java and Python, to conduct the experiment for the following reasons.

The projects we choose should be programmed by object-oriented languages because the theory of patterns (or design patterns) is already well established for object-oriented programming languages. Among all those languages, Java is the most popular programming language [25] and it is a general-purpose computer programming language that is concurrent, class-based, and object-oriented [27]. Java has been fully developed for more than twenty years and it is easy to find a large collections of open source Java projects.

Apart from Java, we also bring Python into the study mainly for two reasons. Firstly, Python is the second most popular language apart from Java according to the website [7] and it is designed to be object-oriented, which is good for finding some design patterns. The other reason is that Python is not an exclusively class-based language, which can also be used to find some other emerging code patterns.
3.1.2 NiCad Clone Detector Settings

In our study, the most important configuration parameters for NiCad clone detector are “renaming”, “size of clone fragment”, “maximum difference threshold” and “granularity”.

For the setting of “renaming”, we apply “blind renaming” to NiCad and try to find the difference before and after applying it. And also after a lot of experiments, we find that setting the “size of clone fragment” as 20 to 800 LOC for Java and Python projects will give us the best results.

Apart from that, we set the “maximum difference threshold” of the clone detector from 0.1 to 0.3 to find different patterns and make some comparisons between them. The last one parameter is “granularity”. The default clone detection granularities could be blocks or functions. However, most of the design patterns are class-based patterns. Thus, we also implement the class-level clone detection and we finally do the code clone detection in class and function granularities.

3.1.3 Project collections for the Study

As we mentioned above, two collections of Java and Python projects are needed for the study. For Java projects, we used the 79 projects of the Qualitas Corpus [9] for the study. For Python, we collected the 30 most trending Python projects [14] on Github for detecting code patterns.

3.2 Extending NiPAT

NiPAT, the pattern detection tool based on NiCad, is originally developed for detecting programming patterns in TXL language to give users some programming feature
3.2. EXTENDING NIPAT

recommendations [30]. Therefore, some changes are necessary in order to make it applicable to other languages (in our case, Python and Java). The following five sections provide an overview of the extensions to NiPAT we make.

3.2.1 A Class Extractor for NiCad

Extracting clone fragments from programs is the very first step for pattern detection. Block and function granularity extractors are already exist in NiCad, however, class extractors are still needed to be implemented for the sake of doing class-granularity clone detections.

3.2.2 Fixing Broken XML Files from NiCad

The output of NiCad is the clone report in XML format. In fact, some XML files are always broken after the clone detection, which cannot be parsed by NiPAT. Therefore, we also add the function to fix the broken XML files.

3.2.3 Pattern Detection Changes

NiPAT is originally designed for TXL. Most TXL functions are relatively short compared to either Python or Java functions. When copying the clone report to NiPAT, the whole function is written to the program, which is reasonable for TXL. However, for other languages which have large and complicated functions, it will cost quite a long time to write whole functions to NiPAT. In this study, we make some changes to NiPAT so that users are able to export pattern reports with either the names of functions or the whole functions.
3.3. PATTERN DETECTION AND ANALYSIS

3.2.4 Changes to the Pattern Report

Exporting a summary of patterns found: We extend the NiPAT to export a summary of all the patterns found. In this report, users are able to easily find some basic properties for all patterns, such as “number of clone classes”, “total files in set”, “internal or external pattern” and so on.

Exporting individual Pattern report: In NiPAT, all the patterns are generated into one HTML file. But for Java projects, we get about 500 patterns in one report, which are quite difficult to analyze or identify. To address this, we also implement the function to generate each pattern into one file.

Adding Hyperlink to the Source File: When hand analyzing patterns, it is always necessary to go through source files rather than the report itself. Before this study, all the source files are extracted into one folder and users can only find files by searching for the file names from this folder. To address this issue and make analysis more efficient, we added hyperlinks for each pattern in the report to easily access source files.

3.3 Pattern Detection and Analysis

After setting up the experiments and extending the NiPAT, we move on to pattern detection and analysis. In this section, we will give an overview of how we detect patterns and the way we identify and categorize them. At the beginning, we extract all files with the file extension “.java” or “.py” from the project collections. Then, we use all these source files as the input for NiCad to find all the clone classes and get the clone report in XML format which has several clone classes in it. After that, NiPAT will take the code clone report as input to find multi-appearance patterns and
export the pattern report in HTML format. In the end, we identify and categorize those patterns. More details will be provided in the following four sections.

3.3.1 Extracting files from Java and Python projects

Files with the given extensions (.java and .py) should be extracted for the clone detection. NiPAT only extracts one copy of the files with the same name for the purpose of reducing redundancies. However, we found that in Java and Python, the source code from those files can be different in some ways, which means they are also useful for finding some patterns. Thus, we change the NiPAT to retain all the files, including those with the same name during file extractions.

3.3.2 Detecting code clones using NiCad

We take all the files from the first step as an input for NiCad clone detector. When we perform the clone detection, various clone classes are found due to the different configuration parameters we set in NiCad. Consequently, we got six Java code clone reports and six Python code clone reports. And these clone classes will be further discussed in Chapter 5.

3.3.3 Detecting patterns using extended NiPAT

After getting the code clone reports from NiCad in XML format, we make use of the extended NiPAT to detect code patterns. According to the twelve code clone reports, we also get corresponding pattern reports for the next stage. More details of the pattern reports are provided in Chapter 5.
3.3.4 Pattern analysis

In our study, all patterns are identified and categorized manually. Some patterns are relatively easily to identify, such as parser generated patterns and data operation patterns (update, delete, write,...). For these patterns, we are able to identify and categorize them just by their functionalities. However, other patterns, especially design patterns, are more difficult to identify because we also have to work on the inheritance relationships between classes.

For instance, although it is possible to identify a factory method design pattern if a factory class is initiated, we still need to go back to the source files and try to find which class are these functions from. After that, we also need to find the parent class (factory class) and the subclasses (concrete factory method classes) of the class and learn about the relations of these classes to finally decide whether it is a factory pattern or not. More techniques about how we identify and categorize patterns will be provided in Chapter 6.

3.4 Summary

In this chapter, we have given an overview of our study. We discussed how we set up the experiment and how we extended NiPAT to make it applicable to Java and Python projects. Finally, we briefly described the analysis of patterns we found during the study.
Chapter 4

Extending NiPAT

As we mentioned in Chapter 2, NiPAT was originally developed for TXL, a special-purpose functional language. However, in our study, we are trying to explore code patterns in some projects implemented in object-oriented programming languages, such as Java and Python. Thus, some changes are needed. In this chapter, we introduce several features we added to NiPAT to make it suitable for detecting programming patterns in Java and Python projects.

In Section 4.1, we introduce the class extractor we added to NiCad for clone detection at class granularity. Some changes we made to fix the broken XML for Java and Python projects will be covered in Section 4.2. In Section 4.3, we describe the changes we made to the process of pattern detection. Finally, some changes to the code pattern report are illustrated in Section 4.4.

4.1 Adding Class Extractor for NiCad

Our aim is to find design patterns and emergent code patterns in object-oriented programs using NiPAT. Most object-oriented design patterns are class-based. However, NiCad only supports clone detection at function and block granularity. Therefore,
the first step for doing the pattern detection in class granularity is to extend NiPAT with class extractors for both Java and Python using TXL.

As we mentioned in Section 2.2.1, all TXL programs are consist of two parts, grammars and transformation rules. We go through the existing function and block extractors in NiCad and find the way to implement the class extractor as follows: For grammars, we redefine classes in Java and Python source files to collect source coordinates as parsed input, and then to allow for XML markup of class definitions as output. For transformation rules, we write a main rule and several subrules to extract and mark up class definitions from parsed input source files.

After extracting all classes from Java and Python projects, we find that it is not a good way to detect code clones in class granularity mainly for two reasons. First, Java project files are organized in a class-based way, and there is only one class in each Java file in most cases. However, the working mechanism of NiPAT is to detect co-appearance of several clone classes in one file. Thus, class level clone detection does not work very well for Java projects. Second, classes are not necessary for Python projects. Therefore, a lot of code clones are not included when we detect Python code clones at class granularity. Considering these two reasons, we decided to abandon detecting code clones at class granularity, concentrating instead on using the pattern detector to recognize the groups of methods that form a class rather than the class itself.

4.2 Fixing Broken XML Files Clone Report

The clone report that is taken as input for NiPAT is in XML format, which is a markup language that defines a set of rules for encoding documents in a format that
is both human-readable and machine-readable [19]. The characters making up an XML document are divided into markup and content. A markup construct that begins with "<" and ends with ">" and tags come in three ways:

- Start-tags: i.e. `<section>`
- End-tags: i.e. `</section>`
- Empty-element tags: i.e. `<line-break />`

However, there are always some symbols which conflict with XML tags in NiCad clone reports. For example, there will be an error when the following report in Figure 4.1 is parsed by NiPAT directly because there are some reserved tags in the highlighted part. In this situation, we have to replace the illegal symbols with blank spaces or some other characters to allow it to parse.

Figure 4.1: An example of broken XML files

From the clone reports we got from NiCad, "<" and ">" are always the most
4.3. CHANGES TO PATTERN DETECTION

commonly conflicting symbols. However, in the original NiPAT, only “<” are replaced by “&lt” as follows in Figure 4.2.

```python
if '<' in line and '<source ' not in line and '</source>' not in line:
    line = line.replace('<', '&lt;')

if 'S' in line:
    line = line.replace('S', '&amp;')

if 'Ê' in line:
    line = line.replace('E', '&E')

if '§' in line:
    line = line.replace('S', '&S')
```

Figure 4.2: Fixing broken XML tags - Part 1

For Java and Python projects, more symbols are needed to be replaced for parsing successfully in NiPAT. In Figure 4.3, we attach some code snippets showing replacement of those conflicting tags. After fixing all the broken XML files, the files are ready to be used by NiPAT.

```python
if '<source name>' in line:
    line = line.replace('<source name>', '$source_name$')

if '<source_location>' in line:
    line = line.replace('<source_location>', '$source_location$')

if '<user>' in line:
    line = line.replace('<user>', '$user$')

if '<password>' in line:
    line = line.replace('<password>', '$password$')

if '>' in line and '<source ' not in line and '</source>' not in line:
    line = line.replace('>', '&gt;')
```

Figure 4.3: Fixing broken XML tags - Part 2

4.3 Changes to Pattern Detection

During pattern detection, original NiPAT copied the entire function source into the clone report. This may be reasonable for TXL projects because they are usually
small. However, for Java and Python projects, it may take way more time to do the pattern detection when copying all the source code from NiCad. In our study, 79 Java projects are used for clone detection and it takes more than 18 hours to get the pattern report by using original NiPAT. Therefore, we added an another option to copy only the function names during the pattern detection instead of the whole functions. In this way, some simple patterns will be easier to identify as soon as possible.

Figure 4.4: An example of pattern report with function names only
As an example, we are able to identify the pattern as a JavaCC parser generated pattern simply by function names from the example in Figure 4.4. Some other patterns can also be easily identified by their function names, such as polymorphism. In this study, we export the report with only function names to identify some simple patterns such as parser patterns and polymorphisms first. And then we go through the details of pattern report to identify other patterns.

4.4 Changes to the Pattern Report

In this section, we introduce some changes we make in order to get a more readable pattern report. First of all, more than 100 patterns are generated by NiPAT in most cases. Therefore, a brief summary of all the patterns as we describe in Section 4.4.1 would facilitate a high level understanding of all the patterns we have. Besides, when there are too many patterns generated from NiPAT, it becomes more difficult to analyze each pattern from a long pattern report. Therefore, we also extend NiPAT to generate several pattern reports with only one code pattern in each in Section 4.4.2. The last change we make is adding a hyperlink for each file in the pattern report which will be covered in Section 4.4.3.

4.4.1 Exporting a summary report of patterns found

In our study, more than 500 design and emergent coding patterns are found from NiPAT. Therefore, it is not easy to get an overview from the pattern report when there are too many patterns generated at the same time. Thus, we add the function to export a brief summary report of all the patterns found. In the summary report, we are able to take a glimpse of some basic information of patterns from NiPAT and
4.4. CHANGES TO THE PATTERN REPORT

we can also use it as a guideline to identify these patterns. Apart from that, we also add the option to set the threshold for “No. of files in set” to change the export of the pattern report. For example, if the threshold is set to 3, only the patterns that appear in more than three files will be seen.

Figure 4.5: An example of pattern summary report (Threshold = 5)

Figure 4.5 shows an example of the pattern summary report with the threshold 5. In this report, we can easily find the ID of patterns and clones classes. We are
4.4. Changes to the Pattern Report

also able to learn about the number of files in each pattern set, the number of clone classes and the name of the involved projects. Apart from that, a pattern can also be categorized as an internal pattern (files in a pattern are from the same project) or external pattern (files in a pattern are from different projects). This information can also be found in the summary report. All the information will help the researchers to better analyze and identify these code patterns.

4.4.2 Exporting Individual Pattern Reports

We add this option mainly because in our case study, we get hundreds of patterns in one pattern report and it is difficult to analyze each pattern from a long report efficiently. Thus, we add an option to export individual report which has only one pattern in each report named by pattern ID.

4.4.3 Adding a Hyperlink to the Source File

Most of the patterns need to be analyzed by examining their original source files. However, it is not easy to check source files only through file directories we got from the summary report of file collections. Therefore, we added a hyperlink to each file directory in code pattern reports as seen in Figure 4.6 for an easy access to all the source files during pattern analysis.
4.4. CHANGES TO THE PATTERN REPORT

Figure 4.6: An example of the file directory hyperlink

```java
private boolean useParentFirst(String resourceName) {
    // default to the global setting and then see
    // if this class belongs to a package which has been
    // designated to use a specific loader first
    // (this one or the parent one)
    // XXX - shouldn’t this always return false in isolated mode?

    boolean useParentFirst = parentFirst;
    for (Enumeration e = systemPackages.elements();
            e.hasMoreElements();)
        String packageName = (String) e.nextElement();
        if (resourceName.startsWith(packageName))
            useParentFirst = true;
    if (useParentFirst)
        break;
    for (Enumeration e = loaderPackages.elements();
            e.hasMoreElements();)
        String packageName = (String) e.nextElement();
        if (resourceName.startsWith(packageName))
            useParentFirst = false;
    return useParentFirst;
}
```
Figure 4.7: The “AntClassLoader.java” file appears after click the hyperlink

After clicking the file directory right above the clone classes, the page will jump to the file in the web browser as seen in Figure 4.7. In this way, researchers are able to access to the source files much more easily instead of going through sources files manually.
4.5 Summary

In this chapter, we introduce how we have extended NiPAT to handle Java and Python programs. We added class extractors to detect the code clones at class granularity. We discussed how we fix the broken files for Java and Python projects. In the end, we introduce all changes to the pattern detection and pattern reports to make NiPAT work better for our study.
Chapter 5

Experimental Setup

The setup and settings used in the experiment are important factors to consider in order to get appropriate experimental results in the pattern analyses in our study. In this chapter, we describe how we choose appropriate open source projects to study code patterns in Section 5.1. In Section 5.2, all the settings of NiCad will be covered including the granularity, threshold, size, normalization and report of code clones. Finally, all the settings of NiPAT (file extractions and number of files in set) are discussed in Section 5.3.

5.1 Choosing Appropriate Open Source Projects

As we mentioned in Section 2.3, NiPAT was originally developed for detecting code patterns from TXL projects for the sake of improving its programming features. In this study, we attempt to find some code patterns in some programming languages other than TXL. Besides finding some emerging coding patterns, we also want to find some design patterns from the code pattern report, such as factory method patterns and adapter patterns.

The theory of design patterns is mostly based on object-oriented programming
languages. Therefore, we prefer to choose some projects from which are implemented in object-oriented programming languages, such as Java, python, C++, C#, etc. After some consideration, we chose to conduct the experiment on Java and Python projects from open source archives. We explain the reasons as follows in Section 5.1.1 and Section 5.1.2.

5.1.1 Java Projects in Our Study

First, Java is one of the most widely used programming languages used by nine million developers and running on seven billion devices worldwide [21]. Java has been fully developed for twenty years and there are a lot of open source Java projects to access online.

Second, Java is a completely object-oriented programming language and design patterns are widely used in Java projects. For instance, most concrete implementations of the abstract Class “Java.io.inputstream” such as “BufferedInputStream”, “GzipInputStream”, “ObjectInputStream” which have constructors taking an instance of the same abstract class are considered as decorator patterns [18]. The frequently used package “Java.io” is from Java core libraries and provides for system input and output through data streams, serialization and the file system. Besides, design patterns are also applied to some other Java APIs, such as “java.langRunnable”, “java.util.pattern”, “java.util.Iterator”, etc [18]. Therefore, it is reasonable to choose Java projects for our study according to the two reasons above.

In our study, we used an open source Java project collection called Qualitas Corpus [9], which is a curated collection of software systems intended to be used for empirical studies of code artifacts. The corpus contains 79 popular Java projects in total.
including JHotDraw, Apache Ant and many others.

5.1.2 Python Projects in Our Study

Similarly to Java, Python is the second most popular programming language and it is a high-level, interpreted, interactive and object-oriented scripting language which is designed to be highly readable [8]. There are many available Python projects on Github to be studied.

Moreover, Python is not an exclusively object-oriented programming language, which means objects may not be used in some projects. Therefore, we can not only find some object-based patterns similar to Java projects, but also some emergent patterns exclusive to the Python programming language. Therefore, we also include Python projects as a comparison to Java projects.

5.2 Settings in the NiCad Clone Detector

There are several experimental settings in NiCad we need to discuss in the following five sections, including granularity, threshold, size, normalization and report of code clones. We also explain and compare the differences applying different settings in NiCad.

5.2.1 Clone Granularities

As we mentioned in Section 2.2, only code clone detection at function and block granularities are supported in NiCad. However, we add class extractors for Java and Python projects, which could be used for detecting code clones at the class granularity. Thus we are able to detect code clones in three different granularities:
5.2. SETTINGS IN THE NICAD CLONE DETECTOR

blocks, functions and classes.

We initially chose to detect clones at the function and class granularities because we hoped to find some emergent patterns and design patterns at the class level. The function granularity works very well when we doing the clone detection. However, the clone detection does not work very well when it comes to class granularity.

After code analysis, we found that the clone detection at the class granularity did not work very well in our study. NiCad can find clones between files, so it would find class clones without a problem. However, the real issue is that NiPAT patterns are within a file, which means each code pattern consists of several functions from one file. We can detect class-level clone pairs, but we cannot detect class-level patterns because each file has only one class and hence at most one class clone. And we always get the whole Java file when we extract a class from Java projects. Thus, it is practically impossible to detect any clone pairs because it is quite difficult to find any code clones between two integrated classes from Java projects. Accordingly, we are also unable to detect any code patterns because pattern detections are based on the code clone reports. Hence, we finally decided to perform the clone detection only in function granularity instead.

5.2.2 Maximum difference threshold

As we mentioned in Section 2.1.1, there are many different ways to define code clones. In our study, we consider the maximum difference threshold less or equal to 0.3 as code clones using NiCad. Thus, during the clone detection, we set the threshold from 0.1 to 0.3 to get different clone reports and compare the results. When we put up the threshold from 0.1 to 0.3, we always get more code clones just as seen in Table 5.1
and Table 5.2 without applying any code normalizations.

Table 5.1: The influence of Threshold for Java projects

<table>
<thead>
<tr>
<th>Threshold</th>
<th>Blind Renaming</th>
<th>No. of Clone Classes</th>
<th>No. of Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>None</td>
<td>1273</td>
<td>199</td>
</tr>
<tr>
<td>0.2</td>
<td>None</td>
<td>2062</td>
<td>338</td>
</tr>
<tr>
<td>0.3</td>
<td>None</td>
<td>2693</td>
<td>422</td>
</tr>
</tbody>
</table>

Table 5.2: The influence of threshold for Python projects

<table>
<thead>
<tr>
<th>Threshold</th>
<th>Blind Renaming</th>
<th>No. of Clone Classes</th>
<th>No. of Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>None</td>
<td>325</td>
<td>33</td>
</tr>
<tr>
<td>0.2</td>
<td>None</td>
<td>770</td>
<td>44</td>
</tr>
<tr>
<td>0.3</td>
<td>None</td>
<td>1230</td>
<td>75</td>
</tr>
</tbody>
</table>

5.2.3 The Size of Clones

The size of clones literally means the lines of code in clone detections. In NiCad configuration files, we are able to set the minimum and the maximum number of lines of code. In the default configuration files, the size of clones is set to 10 to 2500. In our study, we firstly keep the default value of the size of clones. However, we find that there are some patterns that are less worthy to study as seen in Figure 5.1. In our study, we get a lot of similar patterns that are generated by parsers such as JavaCC and most of these patterns are relatively shorter than the other patterns. Thus, we set the size of clones between 20 to 800 to efficiently filter some patterns which are too short or too long.
5.2. SETTINGS IN THE NICAD CLONE DETECTOR

5.2.4 Code Normalization

About the code normalization, we compare the patterns before and after applying blind renaming as seen in Table 5.3 and Table 5.4. Apparently, more patterns are detected after we apply the blind renaming. And we also find some new patterns after applying blind renaming. More details about the patterns we found will be covered in Chapter 6.

Figure 5.1: An example of Java compiler generated patterns

```java
private boolean jj_3_10() {
    if (jj_3R_73()) return true;
    if (jj_3R_72()) return true;
    Token xsp;
    while (true) {
        xsp = jj_scanpos;
        if (jj_3R_74()) {
            jj_scanpos = xsp; break;
        }
    }
    xsp = jj_scanpos;
    if (jj_scan_token(92)) {
        jj_scanpos = xsp;
    }
    if (jj_scan_token(94)) {
        jj_scanpos = xsp;
    }
    if (jj_scan_token(91)) return true;
}
return false;
```
Table 5.3: The influence of normalization (Blind Renaming) for Java projects

<table>
<thead>
<tr>
<th>Threshold</th>
<th>Blind Renaming</th>
<th>No. of Clone Classes</th>
<th>No. of Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>None</td>
<td>1273</td>
<td>199</td>
</tr>
<tr>
<td>0.1</td>
<td>Yes</td>
<td>2026</td>
<td>327</td>
</tr>
<tr>
<td>0.2</td>
<td>None</td>
<td>2062</td>
<td>338</td>
</tr>
<tr>
<td>0.2</td>
<td>Yes</td>
<td>2637</td>
<td>482</td>
</tr>
<tr>
<td>0.3</td>
<td>None</td>
<td>2693</td>
<td>422</td>
</tr>
<tr>
<td>0.3</td>
<td>Yes</td>
<td>3168</td>
<td>596</td>
</tr>
</tbody>
</table>

Table 5.4: The influence of normalization (Blind Renaming) for Python projects

<table>
<thead>
<tr>
<th>Threshold</th>
<th>Blind Renaming</th>
<th>No. of Clone Classes</th>
<th>No. of Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>None</td>
<td>325</td>
<td>33</td>
</tr>
<tr>
<td>0.1</td>
<td>Yes</td>
<td>633</td>
<td>39</td>
</tr>
<tr>
<td>0.2</td>
<td>None</td>
<td>770</td>
<td>44</td>
</tr>
<tr>
<td>0.2</td>
<td>Yes</td>
<td>1081</td>
<td>61</td>
</tr>
<tr>
<td>0.3</td>
<td>None</td>
<td>1230</td>
<td>75</td>
</tr>
<tr>
<td>0.3</td>
<td>Yes</td>
<td>1466</td>
<td>109</td>
</tr>
</tbody>
</table>

5.2.5 Code clone report

In NiCad, users are also able to choose to export the clone report with or without source code. In our study, in order to identify code patterns from the pattern report, we export the clone report with source code in it.
5.3. Settings in NiPAT Pattern Detector

Apart from all settings in NiCad code clone detector, there are also two settings in NiPAT we want to introduce in the following sections.

5.3.1 File Extractions

All the source files are extracted with the given extensions in NiPAT. In our study, we want to extract all the source files from Java (as well as Python) projects. Therefore, we extract all the files with the extension names “.java” and “.py” for code clones and pattern detections.

5.3.2 Number of Files in Set

After doing pattern detections, we got a lot of different code patterns from the report. However, not all the patterns are worthy to be studied and we want to study the patterns that are commonly used. As we mentioned in Section 2.3, the definition of code patterns means there are more than two clone pairs co-appearing in at least two files. Thus, we consider that the more file sets the pattern appears in, the more common the pattern is. For the sake of finding some more frequently used java patterns among hundreds of them, we only keep the patterns which have at least three files in set to study. And after filtering some patterns, all the numbers of patterns are seen as in the following Table 5.5.
In our study, we only apply this filtering to Java projects because we got a lot of similar patterns from a large collection of Java projects. However, when it comes to Python projects, we don’t really have to filter any patterns because only 30 Python projects are involved in this study and we got 109 code patterns the most when we set the threshold to 0.3 with blind renaming.

### 5.4 Summary

In this chapter, we outline all the experimental settings of our study, including the choice of object-oriented languages, NiCad and NiPAT settings, and explain the reasons for these choices. In the next Chapter, we will analyze the emergent and design patterns uncovered in our experiment and compare these patterns between Java and Python projects.
Chapter 6

Pattern Analysis

In this chapter, we analyze all the patterns discovered by NiPAT in our experiment, including both Java and Python patterns. Since we have hundreds of patterns in total and it is impossible to present each pattern individually, we introduce each category of pattern by analyzing one pattern example. Then we provide more details about all patterns in each category and compare the difference between Java and Python patterns. The whole chapter is organized as follows.

In Section 6.1, we analyze code patterns from Java projects and classify them into seven categories, including four different types of design patterns and other three categories of new emergent Java code patterns. Then, in Section 6.2, we introduce all the patterns from Python projects and classify them into seven different categories of code patterns, including two categories of design patterns and five new emergent code patterns. After analyzing all patterns from the Java and Python projects, we also find that there is a lot of difference between them. For instance, we find more design patterns in Java projects than Python projects. By contrast, more emergent patterns were found in Python projects. Therefore, we also analyze and compare the difference between the Java and Python patterns in the last Section 6.3.
6.1 Patterns in Java Projects

All the patterns from Java projects will be covered in this section including four categories of design patterns and three new categories of patterns that emerged from the pattern reports. The whole process of Java pattern analysis can be divided into three stages. First, we go through all the patterns and try to find some code patterns, including both emergent patterns and some simple design patterns that can be easily recognized, such as factory patterns. Second, we refer to the implementations of design patterns in Java API and try to find some other design patterns. In the last stage, we summarize all the different patterns under different experimental settings and analyze these patterns under different experimental settings.

6.1.1 Design Patterns in Java Projects

Design patterns are widely used in Java projects, and the concept of design pattern is also applied to Java core libraries and some Java APIs [12]. In software engineering, design patterns can be divided into three subcategories. Creational design patterns are the patterns used for creating different objects according to diverse situations. Structural design patterns are design patterns that providing a simple way to implement relationships between entities. Behavioral design patterns are patterns which are easing the communications between objects [4].

In fact, design patterns are mostly identified by the relations between classes. For instance, researchers always need to analyze the factory class, all concrete subclasses and the inheritance relationships between them before deciding whether a pattern is a factory pattern or not. Therefore, we proposed to detect code clones at class granularity in the beginning and tried to find some design patterns. However, after we
extracted all classes from Java and performed code clone detection at class granularity, we could barely find any clone pairs because we actually extracted the whole Java file when we tried to extract classes from one Java file in most cases. However, NiPAT can only be used to detect code patterns within one file instead of several files. Therefore, we abandoned the pattern detection at class granularity. As we mentioned in Section 2.2, NiCad only supports clone detection in block and function granularities. We finally decide to detect code patterns at function granularity because too many clone pairs are detected at block granularity and each code pair is too small that we are not able to study them properly.

Since the code clone detection is based on function granularities, we identify the Java design patterns by the initiation of specific classes. For instance, we firstly try to find all the factory classes by referring to Java APIs and classes named by the keyword “factory”. Once we find an object is created by a factory class from a code pattern, we go back to the source code to check the class which has all these functions, find its parent class and subclasses, and check whether it is a factory pattern or not. And all the other design patterns are also identified in a similar way. Finally, we find the following four categories of design patterns. In the following four sections, we analyze each design pattern and provide examples of each pattern.

**Factory Method Pattern**

The factory method pattern is one of the creational design patterns. In a factory method pattern, the factory class defines an interface for creating an object, but lets subclasses decide which class to instantiate [26]. In other words, subclasses are responsible for creating the instance of the class when the factory class is called.
From the pattern reports we got from NiPAT, we detected some factory method patterns and we provide a pattern example in Figure 6.1. In fact, there are three files which consist of the whole pattern, namely “JRLongIncrementerFactory”, “JRIntegerIncrementerFactory” and “JRDoublleIncrementerFactory” and all these three factory classes are inherited by some subclasses which are used for creating different objects under different situations. For instance, different objects “incrementer” are created under different cases through several concrete subclasses, such as “JRLongCountIncrementer” and “JRLongSumIncrementer” in the first clone class. And the second function of the pattern is one of methods from subclass “JRLongCountIncrementer”.
public JRExtendedIncrementer getExtendedIncrementer(CalculationEnum calculation) {
    JRExtendedIncrementer increneter = null;

    switch (calculation) {
    case COUNT :
    {
        increneter = JRLongCountIncrementer.getInstance();
        break;
    }
    case SUM :
    {
        increneter = JRLongSumIncrementer.getInstance();
        break;
    }
    case AVERAGE :
    {
        increneter = JRLongAverageIncrementer.getInstance();
        break;
    }
    case LOWEST :
    case HIGHEST :
    {
        increneter = JRCmparableIncrementerFactory.getInstance().getExtendedIncrementer(calculation);
        break;
    }
    case STANDARD_DEVIATION :
    {
        increneter = JRLongStandardDeviationIncrementer.getInstance();
        break;
    }
    }
}

(a) Factory Method Pattern (ID:289) - Clone Class 1 of 2 (Part I)
6.1. PATTERNS IN JAVA PROJECTS

(b) Factory Method Pattern (ID:289) - Clone Class 1 of 2 (Part II)

```java
    case VARIANCE :
        
    case DISTINCT_COUNT :
        
    case SYSTEM :
    case NOTHING :
    case FIRST :
        
    default :
        
    return incrementer;
```
6.1. PATTERNS IN JAVA PROJECTS

Class: 2 of 2 ... Code in NiCAD Clone Class: 1301 ... Similarity in NiCAD origin Class: 90

Figure 6.1: An example of factory method pattern from pattern report

```java
public Object combine(JRCalculable calculable, JRCalculable calculableValue, AbstractValueProvider valueProvider) {
    Number value = (Number)calculable.getIncrementedValue();
    if (calculableValue.getValue() == null) {
        if (calculable.isInitialized()) {
            return null;
        }
    }
    return value;
} else if (value == null || calculable.isInitialized()) {
    return new Long(((Number)calculableValue.getIncrementedValue()).longValue());
}

double v1 = value.doubleValue();
double c1 = ((Number)valueProvider.getValue(calculable.getHelperVariable(JRCalculable.HELPER_COUNT))).doubleValue();
double s1 = ((Number)valueProvider.getValue(calculable.getHelperVariable(JRCalculable.HELPER_SUM))).doubleValue();

double v2 = ((Number)calculableValue.getIncrementedValue()).doubleValue();
double c2 = ((Number)valueProvider.getValue(calculableValue.getHelperVariable(JRCalculable.HELPER_COUNT))).doubleValue();
double s2 = ((Number)valueProvider.getValue(calculableValue.getHelperVariable(JRCalculable.HELPER_SUM))).doubleValue();

c1 = c2;
s1 = s2;

double c = c1 + c2;
return new Long(
    (long) {
    c1 / c * v1 +
    c2 / c * v2 +
    c2 / c1 * s1 / c + s1 / c +
    c1 / c2 * s2 / c + s2 / c -
    2 * s1 / c + s2 / c
    });
}
```

(c) Factory Method Pattern (ID:289) - Clone Class 2 of 2
Table 6.1: All factory method patterns detected from Java projects

<table>
<thead>
<tr>
<th>No.</th>
<th>Name of Factory Class</th>
<th>Source</th>
<th>Name of Project</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>JRBaseObjectFactory</td>
<td>net.sf.jasperreports.engine.base</td>
<td>JasperReports</td>
</tr>
<tr>
<td>2</td>
<td>KeyedPoolable-ObjectFactory</td>
<td>org.apache.commons.pool</td>
<td>Apache Tomcat</td>
</tr>
<tr>
<td>3</td>
<td>ResourceDownloader-Factory</td>
<td>org.gudy.azureus2.plugins.utils</td>
<td>Azureus</td>
</tr>
</tbody>
</table>

In the Table 6.1, we provide all the other different factory method patterns found in the NiPAT pattern report. All the classes above are factory classes which create objects differently according to different situations.

**Abstract Factory Pattern**

The abstract factory pattern is a creational design pattern used to create factory objects. It is also called a factory of factories. Abstract factory patterns are relatively easy to identify in Java projects because most of the abstract factory classes are named with the keyword “factory”. In the abstract factory pattern, an interface is responsible for creating a factory of related objects without explicitly specifying their classes and each generated factory can give the objects as per the factory pattern [15].

The abstract factory pattern is widely used in Java projects, including the Java API. For instance, the class “DocumentBuilderFactory”, an implementation of an abstract factory for getting factory instance, is widely used for building DOM documents [13]. And there are many implementations of DocumentBuilderFactory depending on
JDK version and installed libraries [1]. The pattern example in Figure 6.2 from Java projects is a pattern example using “DocumentBuilderFactory”.

In this pattern, there are two different functions “toXMLElement” and “readFromXMLElement”. In the first function, “DocumentBuilderFactory” enables applications to obtain a parser that produces DOM object trees from XML documents [1]. In this function, a new instance of “DocumentBuilderFactory” is created in order to further create a “DocumentBuilder”.

In the second function, “TransformerFactory” is being used for an abstract implementation of a factory for creating a specific kind of “Transformer”. And the method “newInstance()” uses a ordered lookup procedure to determine the TransformerFactory implementation class to load. In this case, an object “xmlTransformer” is created after the instantiation of the abstract factory class.
6.1. PATTERNS IN JAVA PROJECTS

(a) Abstract Factory Pattern (ID: 232) - Clone Class 1

```java
public Element toXMLElement(Player player, Document document, boolean showAll,
  boolean toSavedGame, String[] fields) {
  try {
    StringWriter sw = new StringWriter();
    XMLOutputFactory xof = XMLOutputFactory.newInstance();
    XMLStreamWriter xsw = xof.createXMLStreamWriter(sw);
    if (fields == null) {
      toXML(xsw, player, showAll, toSavedGame);
    } else {
      toXMLPartialImpl(xsw, fields);
    }
    xsw.close();

    DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
    Document tempDocument = null;
    try {
      DocumentBuilder builder = factory.newDocumentBuilder();
      tempDocument = builder.parse(new InputSource(new
        StringReader(sw.toString())));
      return (Element) document.importNode(tempDocument.getDocumentElement(),
        true);
    } catch (ParserConfigurationException pce) {
      // Parser with specified options can't be built
      StringWriter swe = new StringWriter();
      pce.printStackTrace(new PrintWriter(swe));
      logger.warning(swe.toString());
      throw new IllegalStateException("ParserConfigurationException");
    } catch (SAXException se) {
      StringWriter swe = new StringWriter();
      se.printStackTrace(new PrintWriter(swe));
      logger.warning(swe.toString());
      throw new IllegalStateException("SAXException");
    } catch (IOException ie) {
      StringWriter swe = new StringWriter();
      ie.printStackTrace(new PrintWriter(swe));
      logger.warning(swe.toString());
      throw new IllegalStateException("IOException");
    } catch (XMLStreamException e) {
      logger.warning(e.toString());
      throw new IllegalStateException("XMLStreamException");
    }
  } catch (IOException e) {
    logger.warning(e.toString());
    throw new IllegalStateException("IOException");
  }
}
```
6.1. PATTERNS IN JAVA PROJECTS

Since abstract factory classes are being used in both functions, we categorize the pattern as a abstract factory pattern. Apart from the pattern above, we also find some other abstract factory patterns from the pattern reports and some information of them is provided in the following Table 6.2.
6.1. PATTERNS IN JAVA PROJECTS

Table 6.2: All abstract factories detected from Java projects

<table>
<thead>
<tr>
<th>No.</th>
<th>Name of Factory Class</th>
<th>Source</th>
<th>Name of Project</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>JspFactory</td>
<td>javax.servlet.jsp</td>
<td>Heritrix</td>
</tr>
<tr>
<td>2</td>
<td>JRAbstractObjectFactory</td>
<td>net.sf.jasperreports.engine</td>
<td>JasperReports</td>
</tr>
<tr>
<td>3</td>
<td>ManagedConnectionFactory</td>
<td>javax.resource.spi</td>
<td>JBoss</td>
</tr>
<tr>
<td>4</td>
<td>EJBJProxyFactory</td>
<td>org.jboss.ejb</td>
<td>JBoss</td>
</tr>
<tr>
<td>5</td>
<td>SAXParserFactory</td>
<td>javax.xml.parsers</td>
<td>JChemPaint</td>
</tr>
<tr>
<td>6</td>
<td>CertificateFactory</td>
<td>java.security.cert</td>
<td>Apache Tomcat</td>
</tr>
<tr>
<td>7</td>
<td>DTDDVFactory</td>
<td>org.apache.xerces.impl.dv</td>
<td>Apache Xerces</td>
</tr>
<tr>
<td>8</td>
<td>TransformerFactory</td>
<td>javax.xml.transform</td>
<td>FreeCol</td>
</tr>
<tr>
<td>9</td>
<td>DocumentBuilderFactory</td>
<td>javax.xml.parsers</td>
<td>FreeCol</td>
</tr>
<tr>
<td>10</td>
<td>EscherRecordFactory</td>
<td>org.apache.poi.ddf</td>
<td>Apache POI</td>
</tr>
</tbody>
</table>

Builder Pattern

The builder pattern is aimed to separate the construction of a complex object from its representation so that the same construction process can create different representations [26]. After referring to Java libraries, we learned that all implementations of “java.lang.Appendable” are considered as the using of builder patterns. For example, the class “StringBuffer” and “StringBuilder” are both implementations of builder patterns. However, “StringBuffer” could be used for synchronized and that is also the reason why it is more widely used for Java projects.

As shown in Figure 6.3, “StringBuffer” is used for creating two objects “expected” in the first function and ‘retval’ with different parameters. The function “append()”
is the builder method from the class “StringBuffer” and it could be used to append different specified character sequences to an appendable sequence. Especially in the second function, the object “retval” is created under several different situations. Therefore, we categorize all patterns like this as builder patterns.

Unlike factory patterns, the “StringBuffer” class is the only builder class we find in the NiPAT pattern report but it appears several times in different Java projects.
6.1. PATTERNS IN JAVA PROJECTS

(a) Builder Pattern (ID: 152) - Clone Class 1

```java
public String getMessage() {
    if (!specialConstructor) {
        return super.getMessage();
    }
    StringBuffer expected = new StringBuffer();
    int maxSize = 0;
    for (int i = 0; i < expectedTokenSequences.length; i++) {
        if (maxSize < expectedTokenSequences[i].length) {
            maxSize = expectedTokenSequences[i].length;
        }
    }
    for (int j = 0; j < expectedTokenSequences[0].length; j++) {
        expected.append(tokenImage[expectedTokenSequences[0][j]]).append(' ');}
    }
    if (expectedTokenSequences[0][expectedTokenSequences[0].length - 1] != 0) {
        expected.append("...");
    }
    expected.append((col).append("  ");
    }
    String retval = "Encountered 
";
    Token tok = currentToken.next;
    for (int i = 0; i < maxSize; i++) {
        if (i != 0) retval += 
" ";
        if (tok.kind == 0) {
            retval += tokenImage[tok.kind];
            retval += 
" ";
            retval += addEscapeChars(tok.image);
            retval += 
" ";
            tok = tok.next;
        }
    retval += ) at line " + currentToken.next.beginLine + ", column " +
currentToken.next.beginColumn;
    retval += 
"." + col;
    if (expectedTokenSequences.length == 1) { 
        retval += "Was expecting:" + col + 
" ";
    } else {
        retval += "Was expecting one of:" + col + 
" ";
    }
    retval += expected.toStr();
    return retval;
}
```
6.1. PATTERNS IN JAVA PROJECTS

(b) Builder Pattern (ID: 257) - Clone Class 2

Figure 6.3: An example of builder pattern from pattern report
Adapter Pattern

The adapter pattern, also known as a wrapper pattern, works as a bridge between two incompatible interfaces. This type of design pattern comes under the structural pattern category as it combines the capability of two independent interfaces. This pattern involves a single class which is responsible to join functionalities of independent or incompatible interfaces [16].

After referring to Java core library, we found the most widely used adapter classes are “InputStreamReader” and “OutputStreamWriter”. An “InputStreamReader” is a bridge from byte streams to character streams, which means it reads bytes and decodes them into characters using a specified charset [2]. Similarly, an “OutputStreamWriter” is a bridge from character streams to byte streams, which means characters written to it are encoded into bytes using a specified charset [3].

In the pattern report, we found some patterns using the two adapter classes above and we provide an example of the pattern we found in Figure 6.4. In this pattern, the two functions are used to load “Rfunctions”. And the class “InputStreamReader” in both two functions is instantiated to wrapped the object and so it could be used by the “BufferedReader” after being adapted.
private void loadRFunctions(Engine engine) {
    // File.separator is used to be system independent
    // Fix me: After creating a jar file it don't work on a windows OS
    // but within eclipse it won't work on while working with '/' on windows OS
    // No idea how to solve this

    String scriptLocator = "org" + File.separator + "opensecience" +
                          File.separator + "cdk" + File.separator + "org" + File.separator +
                          "model" + File.separator + "data" + File.separator + "helper.R";
    String outputLocator = "org/opensecience/cdk/qsar/model/data/helper.R";
    try {
        File scriptFile = File.createTempFile("XXXX", ".R");
        scriptFile.deleteOnExit();

        InputStreamReader reader = new InputStreamReader(
            this.getClass().getClassLoader().getResourceAsStream(scriptLocator));
        BufferedReader inFile = new BufferedReader(reader);

        FileWriter outFile = new FileWriter(scriptFile);
        BufferedWriter outBuffer = new BufferedWriter(outFile);
        String inputLine;
        while ((inputLine = inFile.readLine()) != null) {
            StringBuffer outBuffer = new StringBuffer(inputLine);
            outBuffer.newLine();
        }
        outBuffer.close();
    }
    catch (IOException exception) {
        log.error("Could not load helper R script for JRI: ", scriptLocator);
        log.debug(exception);
    }
}

(a) Adapter Pattern (ID: 318) - Clone Class 1 of 2
6.1. PATTERNS IN JAVA PROJECTS

(b) Adapter Pattern (ID: 318) - Clone Class 2 of 2

Figure 6.4: An example of adapter pattern from pattern report

6.1.2 Emergent Java Patterns

Apart from the four design patterns above, we also find some other code patterns emerging from the pattern report and we categorize them into the following three categories. We also analyze each of them by providing some pattern examples in the
next three sections.

**Generated Parser Patterns**

From the Java pattern reports, we find some patterns which are generated by different compilers in Java projects, such as JSON and JavaCC. Among all the Java pattern reports, the most commonly generated parser pattern is from JavaCC and they are always identified with similar function names such as “final private boolean jj_3R_129()”.

In the following Figure 6.5, we provide an example of the generated parser pattern. In the first clone class, there are two functions listed and separated by the dotted line. In fact, there are four different methods in this clone class, however, we only keep two of them to introduce the pattern because they are all similar to each other. In this clone class, we are able to find the functions “jj_3R_129()” and “jj_3R_57()”, which don’t have a lot of meaning but part of the parser generated by JavaCC compiler, which is an open source parser generator and lexical analyzer generator written in Java programming language [6].

In the second and third clone classes, we can still see some parameters start with “jjte”, “jjtn”, “jjtc” and “label” which also means the functions are all generated by JavaCC. For the second clone class, there are actually 21 functions in it and 3 functions consists of the third clone class. However, we only listed two functions for each clone class, which is already enough to understand the whole pattern.

In fact, there is not too much to study about this category of pattern. However, since it’s appears quite frequently in the pattern report, we decide to categorize all these patterns into “Generated Parser Patterns”.

Class: 1 of 3...Code in NiCAD Clone Class: 332...Similarity in NiCAD origin Class: 80

examples/avappjedit/src/org/gjt/sp/edt/bsh/Parserjava
CloneClass= 332 Similarity= 80

```java
final private boolean jj_3R_129() {
    Token xsp;
    xsp = jj_scanpos;
    if (jj_3R_130()) {
        jj_scanpos = xsp;
        if (jj_3R_139()) {
            jj_scanpos = xsp;
            if (jj_3R_140()) {
                jj_scanpos = xsp;
                if (jj_3R_141()) {
                    jj_scanpos = xsp;
                    if (jj_3R_142()) {
                        jj_scanpos = xsp;
                        if (jj_3R_143()) {
                            jj_scanpos = xsp;
                            if (jj_3R_144()) return true;
                        }
                    }
                }
            }
        }
    }
    return false;
}
```

```
final private boolean jj_3R_87() {
    Token xsp;
    xsp = jj_scanpos;
    if (jj_3R_88()) {
        jj_scanpos = xsp;
        if (jj_3R_89()) {
            jj_scanpos = xsp;
            if (jj_3R_90()) {
                jj_scanpos = xsp;
                if (jj_3R_91()) {
                    jj_scanpos = xsp;
                    if (jj_3R_101()) {
                        jj_scanpos = xsp;
                        if (jj_3R_102()) {
                            jj_scanpos = xsp;
                            if (jj_3R_103()) return true;
                        }
                    }
                }
            }
        }
    }
    return false;
}
```

(a) Parser Generated Pattern (ID: 58) - Clone Class 1 of 3
Class: 2 of 3  Code in NiCad Clone Class: 553  Similarity in NiCad origin Class: 80

examples/avajprettisrc/org/itsp/edittbsh/Parserjava
CloneClass = 553  Similarity = 80

```
final public void StatementExpressionList() throws ParseException {
    /*$bn(jjtree) StatementExpressionList */
    BSHStatementExpressionList jjtn000 = new
    BSHStatementExpressionList(JJSTATEMENTEXPRESSIONLIST);
    boolean jjtc000 = true;
    jjtree.openNodeScope(jjtn000);
    jjtreeOpenNodeScope(jjtn000);
    try {
        StatementExpression();
        label_26:
        while (true) {
            switch (((jj_ntx==1)||jj_ntx()==jj_ntk)) {
            case COMMA:
                ;
                break label_26;
            default:
                ;
                break label_26;
            }
            jj_consume_token(COMMA);
            StatementExpression();
        }
        catch (Throwable jjte000) {
            if (jjtc000) {
                jjtree.closeNodeScope(jjtn000);
                jjtc000 = false;
            }
            else {
                jjtree.popNode();
            }
            if (jjte000 instanceof RuntimeException) {
                if (true) throw (RuntimeException)jjte000;
            }
            if (jjte000 instanceof ParseException) {
                if (true) throw (ParseException)jjte000;
            }
            if (true) throw (Error)jjte000;
        }
        finally {
            if (jjtc000) {
                jjtree.closeNodeScope(jjtn000, true);
                jjtreeCloseNodeScope(jjtn000);
            }
        }
    }
}
```

(b) Parser Generated Pattern (ID: 58) - Clone Class 2 of 3 (Part I)
6.1. PATTERNS IN JAVA PROJECTS

...............

final public void Block() throws ParseException {
  /*@Generated({jjtree}) Block */
  BSHElack jjtn000 = new BSHElack(JJTEBLOCK);
  boolean jjtc000 = true;
  jjtree.addNodeScope(jjtn000);
  jjtree.addNodeScope(jjtn000);
  try {
    jj_consume_token(LBRACE);
    label_22:
    while (true) {
      if (jj_2_23(1)) {
      } else {
        BREAK label_22;
      }
      BlockStatement();
    }
    jj_consume_token(RBRACE);
  } catch (Throwable jjte000) {
    if (jjtc000) {
      jjtree.addNodeScope(jjtn000);
      jjtc000 = false;
    } else {
      jjtree.popNode();
    }
    if (jjte000 instanceof RuntimeException) {
      if (true) throw (RuntimeException)jjte000;
    }
    if (jjte000 instanceof ParseException) {
      if (true) throw (ParseException)jjte000;
    }
    if (true) throw (Error)jjte000;
  } finally {
    if (jjtc000) {
      jjtree.addNodeScope(jjtn000, true);
      jjtreeCloseNodeScope(jjtn000);
    }
  }
}

(c) Parser Generated Pattern (ID: 58) - Clone Class 2 of 3 (Part II)
6.1. PATTERNS IN JAVA PROJECTS

Class: 3 of 3 ...Code in NICAD Clone Class: 560 ...Similarity in NICAD origin Class: 80

CloneClass= 560 Similarity= 80

eamples/javajpl/ed/srjg/ptjedj/bsh/Parser.java

```java
final public void forInit() throws ParseException {
  Token t = null;
  if (jj_2_31(2147493647)) {
    TypeVariableDeclaration();
  } else {
    switch (((jj_ntk===-1)?jj_ntk():jj_ntk)) {
      case BOOLEAN:
      case BYTE:
      case CHAR:
      case DOUBLE:
      case FALSE:
      case FLOAT:
      case INT:
      case LONG:
      case NEW:
      case NULL:
      case SHORT:
      case TRUE:
      case VOID:
      case INTEGER_LITERAL:
      case FLOATING_POINT_LITERAL:
      case CHARACTER_LITERAL:
      case STRING_LITERAL:
      case IDENTIFIER:
      case LPAREN:
      case BANG:
      case TILDE:
      case INCR:
      case DECR:
      case PLUS:
      case MINUS:
      {
        StatementExpressionList();
        break;
      } default:
      {
        jj_consume_token(-1);
        throw new ParseException();
      }
    }
  }
}
```

(d) Parser Generated Pattern (ID: 58) - Clone Class 3 of 3 (Part I)
6.1. PATTERNS IN JAVA PROJECTS

```java
final public void Expression() throws ParseException {
    if (jj_2_8((2147483647))) {
        Assignment();
    } else {
        switch (((jj_ntk == -1) ? jj_ntk() : jj_ntk)) {
            case BOOLEAN:
            case BYTE:
            case CHAR:
            case DOUBLE:
            case FALSE:
            case FLOAT:
            case INT:
            case LONG:
            case REAL:
            case NULL:
            case SHORT:
            case TRUE:
            case VOID:
            case INTEGER_LITERAL:
            case FLOATING_POINT_LITERAL:
            case CHARACTER_LITERAL:
            case STRING_LITERAL:
            case IDENTIFIER:
            case LPAREN:
            case EQ:
            case TILDE:
            case INCR:
            case DECR:
            case PLUS:
            case MINUS:
                ConditionalExpression();
                break;
            default:
                jj_consume_token(-1);
                throw new ParseException();
        }
    }
}
```

(e) Parser Generated Pattern (ID: 58) - Clone Class 3 of 3 (Part II)

Figure 6.5: An example of parser generated pattern from pattern report
6.1. PATTERNS IN JAVA PROJECTS

Class Polymorphism Pattern

Apart from the parser generated pattern we mentioned above, we also find another pattern which appears a lot in the pattern report and we define it as “class polymorphism pattern”. “Polymorphism” is commonly used in Java projects for reusing purpose. However, when it comes to the “class polymorphism pattern”, we actually mean the classes which have similar functions but they are all final classes and don’t inherit from any parent classes.

In the following Figure 6.6, we show an example of the class polymorphism we detect from the Java projects. In fact, there are five different files consist of the pattern, “OpenDoubleIntHashMap.java”, “OpenIntObjectHashMap.java”, “OpenLongObjectHashMap.java”, “OpenIntDoubleHashMap.java” and “OpenIntIntHashMap.java”. Considering this pattern is a relationship between different classes, we list two file sets out of five to illustrate the relationship between different classes.
6.1. PATTERNS IN JAVA PROJECTS

protected int indexOfInsertion(double key) {
    final double tab[] = table;
    final byte stat[] = state;
    final int length = tab.length;

    final int hash = HashFunctions.hash(key) & 0x7FFFFFFF;
    int i = hash % length;
    int decrement = hash % (length-2); // double hashing, see
    http://www.ecs.umn.edu/~faculty/haileman/hasht/node4.html
    //int decrement = (hash / length) % length;
    if (decrement == 0) decrement = 1;

    // stop if we find a removed or free slot, or if we find the key itself
    // do NOT skip over removed slots (yes, open addressing is like that...)
    while (stat[i] == FULL & tab[i] != key) {
        i -= decrement;
        //hashCollisions++;
        if (i<0) i+=length;
        }

    if (stat[i] == REMOVED) {
        // stop if we find a free slot, or if we find the key itself.
        // do skip over removed slots (yes, open addressing is like that...)
        // assertion: there is at least one FREE slot.
        int j = 1;
        while (stat[j] != FREE & (stat[j] == REMOVED || tab[j] != key)) {
            j -= decrement;
            //hashCollisions++;
            if (j<0) j+=length;
        }
        if (stat[j] == FREE) i = j;
    }

    if (stat[i] == FULL) {
        // key already contained at slot i.
        // return a negative number identifying the slot.
        return -i-1;
    }

    // not already contained, should be inserted at slot i.
    // return a number = 0 identifying the slot.
    return i;
}
protected int indexOfInsertion(int key) {
    final int tab[] = table;
    final byte stat[] = state;
    final int length = tab.length;

    final int hash = HashFunctions.hash(key) & 0x7FFFFFFF;
    int i = hash % length;
    int decrement = hash % (length-1); // double hashing, see
    http://www.ece.ucr.edu/faculty/helleman/hasb/node4.html
    //int decrement = (hash / length) % length;
    if (decrement == 0) decrement = 1;

    // stop if we find a removed or free slot, or if we find the key itself
    // do NOT skip over removed slots (yes, open addressing is like that...)
    while (stat[i] == FULL || tab[i] != key) {
        i -= decrement;
        //hashCollisions++;
        if (i<0) i+=length;
    }

    if (stat[i] == REMOVED) {
        // stop if we find a free slot, or if we find the key itself.
        // do skip over removed slots (yes, open addressing is like that...)
        // assertion: there is at least one FREE slot.
        int j = i;
        while (stat[i] != FREE && (stat[i] == REMOVED || tab[i] != key)) {
            i -= decrement;
            //hashCollisions++;
            if (i<0) i+=length;
        }
        if (stat[i] == FREE) i = j;
    }

    if (stat[i] == FULL) {
        // key already contained at slot i.
        // return a negative number identifying the slot.
        return -i-1;
    }

    // not already contained, should be inserted at slot i.
    // return a number = 0 identifying the slot.
    return i;
}
public boolean put(double key, int value) {
    int i = index0fInsertion(key);
    if (i<0) { //already contained
        i = -i -1;
        this.values[i]=value;
        return false;
    }

    if (this.distinct this.highWaterMark) {
        int newCapacity = chooseGrowCapacity(this.distinct=1, this.minLoadFactor, this.maxLoadFactor);
        /*
        System.out.print("grow rehashing ");
        System.out.println("at distinct="+distinct+, capacity="+table.length+"
        to newCapacity="+newCapacity+" ..."));
        */
        rehash(newCapacity);
        return put(key, value);
    }

    this.table[i]=key;
    this.values[i]=value;
    if (this.state[i]==FREE) this.freeEntries--;
    this.state[i]=FULL;
    this.distinct++;

    if (this.freeEntries < 1) { //delta
        int newCapacity = chooseGrowCapacity(this.distinct+1, this.minLoadFactor, this.maxLoadFactor);
        rehash(newCapacity);
    }

    return true;
}
public boolean put(int key, Object value) {
    int i = indexOfInsertion(key);
    if (i<0) { // already contained
        i = -i -1;
        this.values[i]=value;
        return false;
    }

    if (this.distinct && this.highWaterMark) {
        int newCapacity = chooseGrowCapacity(this.distinct+1, this.minLoadFactor,
                this.maxLoadFactor);
        rehash(newCapacity);
        return put(key, value);
    }

    this.table[i]=key;
    this.values[i]=value;
    if (this.state[i]==FREE) this.freeEntries--;
    this.state[i]=FULL;
    this.distinct++;

    if (this.freeEntries < 1) { // delta
        int newCapacity = chooseGrowCapacity(this.distinct+1, this.minLoadFactor,
                this.maxLoadFactor);
        rehash(newCapacity);
    }

    return true;
}
6.1. PATTERNS IN JAVA PROJECTS

(e) Class Polymorphism Pattern (ID:195) - Clone Class 3 of 3 (Part I)
(f) Class Polymorphism Pattern (ID:195) - Clone Class 3 of 3 (Part II)

Figure 6.6: An example of class polymorphism pattern from pattern report

From the pattern above, the three functions from “OpenDoubleIntHashMap.java” is quite similar to the three functions from “OpenIntObjectHashMap.java”. The big difference between them is the type of the data. After referring to the source code and , we learned that class “OpenDoubleIntHashMap” implements a hash map holding (key,value) associations of type (double–>int) and it automatically grows and shrinks as needed. The class “OpenIntObjectHashMap” defines a hash map holding
(key, value) associations of type (int -> object) and it automatically grows and shrinks as needed. Similarly, the other three files in this pattern but not listed are hash maps holding (key, value) associations of type (long -> object), (int -> double) and (int -> int).

Therefore, we consider the pattern above and all the other similar patterns as “class polymorphism” patterns because the functions and the name of classes are similar. However, there is no inheritance relationships between them.

**Database Operation Pattern**

This pattern is another pattern we found from the Java pattern report which is used to operate the database in Java projects. For instance, some functions are designed to add, delete or update the value of the database. In our project, we find some patterns which are being used for operating database in Figure 6.7.

From the pattern below, we found that all the functions are being used to implement the modifications or search for the SQL database. For example, the method “delete()” is being used for removing a specific row from the database. Also, the function “findByPrimaryKey()” is being used to search for the specific row by the value of primary key from the database. Similarly, the function “findByAlternativeKey()” is designed to search for the specific row through the value of alternative key from the database. Therefore, considering all the functions above are mostly being used to operate the data from the database, we categorize these patterns as database operation patterns.
public void delete(int zoneID) {
    throws DatabaseException, ObjectNotFoundException {
        Connection connection = null;
        PreparedStatement statement = null;
        StringBuffer sql = new StringBuffer(512);
        sql.append("DELETE FROM " + TABLE_NAME);
        sql.append(" WHERE ZoneID = ?");
        try {
            connection = DBUtils.getConnection();
            statement = connection.prepareStatement(sql.toString());
            statement.setInt(1, zoneID);
            if (statement.executeUpdate() != 1) {
                throw new ObjectNotFoundException("Cannot delete a row in table mvnadZone where primary key = (" + zoneID + ")");
            }
            catch (SQLException sqle) {
                log.error("Sql Execution Error!", sqle);
                throw new DatabaseException("Error executing SQL in ZoneDAOImplJDBC.delete.");
            }
        } finally {
            DBUtils.closeStatement(statement);
            DBUtils.closeConnection(connection);
        }
    }
}
6.1. PATTERNS IN JAVA PROJECTS

(b) Data Operation Pattern (ID:396) - Clone Class 2 of 6

```java
public int getNumberOfBeans() throws DatabaseException {
    Connection connection = null;
    PreparedStatement statement = null;
    ResultSet resultSet = null;
    StringBuffer sql = new StringBuffer(512);
    sql.append("SELECT Count(*)");
    sql.append(" FROM " + TABLE_NAME);
    try {
        connection = DBUtils.getConnection();
        statement = connection.prepareStatement(sql.toString());
        resultSet = statement.executeQuery();
        AssertUtil.doAssert(resultSet.next(), "Assertion in ZoneDAOImplJDBC.getNumberOfBeans.");
        return resultSet.getInt(1);
    } catch (SQLException sqlE) {
        log.error("Sql Execution Error!", sqlE);
        throw new DatabaseException("Error executing SQL in ZoneDAOImplJDBC.getNumberOfBeans.");
    } finally {
        DBUtils.closeResultSet(resultSet);
        DBUtils.closeStatement(statement);
        DBUtils.closeConnection(connection);
    }
}
```
public void updateReceivedImpression(int zoneID, int zoneReceivedImpression)
    throws ObjectNotFoundException, DatabaseException {
    Connection connection = null;
    PreparedStatement statement = null;
    StringBuffer sql = new StringBuffer(512);
    sql.append("UPDATE " + TABLE_NAME + " SET ZoneReceivedImpression = ?");
    sql.append(" WHERE ZoneID = ?");
    try {
        connection = DBUtils.getConnection();
        statement = connection.prepareStatement(sql.toString());

        // // column(s) to update
        statement.setInt(1, zoneReceivedImpression);

        // primary key column(s)
        statement.setInt(2, zoneID);

        if (statement.executeUpdate() != 1) {
            throw new ObjectNotFoundException("Cannot update table Zone where
            primary key = (" + zoneID + ").");
        }
    } catch (SQLException sqlException) {
        log.error("Sql Execution Error!", sqlException);
        throw new DatabaseException("Error executing SQL in
        ZoneDAOImplJDBC.updateReceivedImpression.");
    } finally {
        DBUtils.closeStatement(statement);
        DBUtils.closeConnection(connection);
    }
}
(d) Data Operation Pattern (ID:396) - Clone Class 4 of 6
6.1. PATTERNS IN JAVA PROJECTS

(e) Data Operation Pattern (ID:396) - Clone Class 5 of 6
6.1. PATTERNS IN JAVA PROJECTS

(f) Data Operation Pattern (ID:396) - Clone Class 6 of 6

Figure 6.7: An example of data operation pattern from pattern report

6.1.3 Summary Of all Java Patterns

As we mentioned above, there are four categories of design pattern and three emerging patterns detected from NiPAT. And also, there are still some patterns that we are not able to identify because there are no apparent features in it. In Section 5.1.1, we mentioned all the number of patterns we study in this project, and in the following
Table 6.3, a summary of all the design patterns we identified are presented. Besides, we also present Table 6.4 to show the number of emerging Java patterns from different pattern reports. In these two tables, the “No. of Patterns (-)” stands for the number of patterns before applying blind renaming and “No. of Patterns (+)” means the number of patterns after applying blind renaming during code clone detections.

Table 6.3: A summary of all Java design patterns under different thresholds

<table>
<thead>
<tr>
<th>Pattern Category</th>
<th>Threshold</th>
<th>No. of Patterns (-)</th>
<th>No. of Patterns (+)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract Factory</td>
<td>0.1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>0.2</td>
<td>9</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>0.3</td>
<td>11</td>
<td>15</td>
</tr>
<tr>
<td>Factory Method</td>
<td>0.1</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>0.2</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>0.3</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>Builder</td>
<td>0.1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>0.2</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>0.3</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>Adapter</td>
<td>0.1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>0.2</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>0.3</td>
<td>7</td>
<td>16</td>
</tr>
</tbody>
</table>
Table 6.4: A summary of all emergent Java patterns under different thresholds

<table>
<thead>
<tr>
<th>Pattern Category</th>
<th>Threshold</th>
<th>No. of Patterns (-)</th>
<th>No. of Patterns (+)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parser Generated</td>
<td>0.1</td>
<td>28</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>0.2</td>
<td>33</td>
<td>52</td>
</tr>
<tr>
<td></td>
<td>0.3</td>
<td>44</td>
<td>61</td>
</tr>
<tr>
<td>Class Polymorphism</td>
<td>0.1</td>
<td>9</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>0.2</td>
<td>13</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>0.3</td>
<td>17</td>
<td>28</td>
</tr>
<tr>
<td>Database Operation</td>
<td>0.1</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>0.2</td>
<td>5</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>0.3</td>
<td>9</td>
<td>17</td>
</tr>
<tr>
<td>Unknown Patterns</td>
<td>0.1</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>0.2</td>
<td>3</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>0.3</td>
<td>6</td>
<td>24</td>
</tr>
</tbody>
</table>

From the two tables above, we find several points as follows. First as the increment of threshold during code clone detection, the more code patterns we get from NiPAT. Applying “blind renaming” is also an important factor for the number of patterns we get. Second, emerging patterns are more than Java design patterns in general and parser generated patterns appear most frequent in the pattern report. Finally, we find that when we set the threshold to 0.1 without blind renaming, no builder patterns are detected. However, when we set the threshold to 0.2 with using blind renaming when we do the code clone detection, all categories of these patterns can be detected and that is the reason why we picked all our pattern examples from the
latter experiment settings to analyze.

6.2 Patterns in Python Projects

After analyzing all the Java patterns above, we also analyzed the pattern report from Python projects. It is easier to identify and categorize Python patterns compared to Java patterns mainly because the size of patterns from Python projects is smaller compared to Java projects. The relationships between classes in Python projects are also less complicated than Java projects.

In Section 6.2.1, we show two design patterns we find from the code pattern report and analyze them by giving examples for each of them. Since Python is not an exclusively object-oriented programming language, we are also able to find some emerging code patterns from the Python pattern report and all these patterns will be covered in Section 6.2.2. In the last Section 6.2.3, we summarize all the patterns from the Python projects and conclude some findings from different pattern reports under different experimental settings.

6.2.1 Python Design Patterns

After going through all the Python report, we find two categories of design patterns and we analyze these two patterns in the following two sections.

Singleton Pattern

Compared to Java projects, the singleton pattern is not often used in Python projects. Sometimes, singleton patterns are even considered code antipatterns in Python projects, which means a bad practice of coding [31]. However, from the pattern report, we still
find a singleton pattern that creates only one instance for tcp socket connections.

```python

def __new__(cls, opts, **kwargs):
    
    Only create one instance of channel per __key() 
    
    # do we have any mapping for this io_loop 
    io_loop = kwargs.get('io_loop') or tornado.ioloop.IOLoop.current()
    if io_loop not in cls.instance_map:
        cls.instance_map[io_loop] = weakref.WeakValueDictionary()
        loop_instance_map = cls.instance_map[io_loop]

    key = cls.__key(opts, **kwargs)
    if key not in loop_instance_map:
        log.debug('Initializing new AsyncTCPReqChannel for [{}].format(key))
        # we need to make a local variable for this, as we are going to store
        # it in a WeakValueDictionary-- which will remove the item if no one
        # references it-- this forces a reference while we return to the caller
        new_obj = object.__new__(cls)
        new_obj.__singleton_init__(opts, **kwargs)
        loop_instance_map[key] = new_obj
    else:
        log.debug('Re-Using AsyncTCPReqChannel for [{}].format(key))
        return loop_instance_map[key]
```

(a) Singleton Pattern (ID:34) - Clone Class 1 of 2
In Figure 6.8, there are two functions in this pattern. And both of these two functions are from the class called “AsyncTCPReqChannel”. Routines sending to tcp are all encapsulated in this class. The first function in the pattern is being used to create a new instance of the class and the second function is a singleton constructor which is being used to make sure only initialize one and only instance per key. Although there are still some other methods or functions in this class, these two functions are good enough for us to identify a singleton design pattern.
In fact, this singleton pattern is slightly different from the general definition of “Singleton”. In software engineering, the singleton pattern is a design pattern that restricts the instantiation of a class to only one object [10]. However, in this pattern, one instance of channel can be created per key, which means more than one object can be created with different key values. Since the constructor was named as singleton and it has some features of singleton pattern, we still consider it as a singleton pattern.

**Decorator Pattern**

Because of incompatibilities between Python 2 and Python 3, decorators are being used a lot in Python projects. For instance, the Python library “Six” provides simple utilities for wrapping over differences between Python 2 and Python 3. It is intended to support codebases that work on both Python 2 and 3 without modification.

```python
def __init__(self, name, old_mod, new_mod, old_attr=None, new_attr=None):
    super(MovedAttribute, self).__init__(name)
    if PY3:
        if new_mod is None:
            new_mod = name
        self.mod = new_mod
        if new_attr is None:
            if old_attr is None:
                new_attr = name
            else:
                new_attr = old_attr
        self.attr = new_attr
    else:
        self.mod = old_mod
        if old_attr is None:
            old_attr = name
        self.attr = old_attr
```
def load_module(self, fullname):
    try:
        # in case of a reload
        return sys.modules[fullname]
    except KeyError:
        pass
    mod = self._get_module(fullname)
    if isinstance(mod, MovedModule):
        mod = mod._Resolve()
    else:
        mod._loader__ = self
    sys.modules[fullname] = mod
    return mod
6.2. PATTERNS IN PYTHON PROJECTS

```python
def print_(*args, **kwargs):
    """The new-style print function for Python 2.4 and 2.5.""
    fp = kwargs.pop("file", sys.stdout)
    if fp is None:
        return
    def write(data):
        if not isinstance(data, basestring):
            data = str(data)
        # If the file has an encoding, encode unicode with it.
        if (isinstance(fp, file) and
            isinstance(data, unicode) and
            fp.encoding is not None):
            errors = getattr(fp, "errors", None)
            if errors is None:
                errors = "strict"
            data = data.encode(fp.encoding, errors)
        fp.write(data)
    want_unicode = False
    sep = kwargs.pop("sep", None)
    if sep is not None:
        if isinstance(sep, unicode):
            want_unicode = True
        elif not isinstance(sep, str):
            raise TypeError("sep must be None or a string")
    end = kwargs.pop("end", None)
    if end is not None:
        if isinstance(end, unicode):
            want_unicode = True
        elif not isinstance(end, str):
            raise TypeError("end must be None or a string")
    if kwargs:
        raise TypeError("invalid keyword arguments to print()")
    if not want_unicode:
        for arg in args:
            if isinstance(arg, unicode):
                want_unicode = True
                break
    if want_unicode:
        newline = unicode("\n")
        space = unicode(" ")
    else:
        newline = "\n"
        space = " "
    if sep is None:
        sep = space
    if end is None:
        end = newline
    for i, arg in enumerate(args):
        if i:
            write(sep)
        write(arg)
    write(end)
```

(c) Decorator Pattern (ID:2) - Clone Class 3 of 4
6.2. PATTERNS IN PYTHON PROJECTS

Figure 6.9: An example of decorator pattern from pattern report

We provide a pattern example from the pattern report in Figure 6.9. There are four functions in this pattern, the first of which is a constructor for the whole class. From this constructor, we can see some hints about the decorator from the two parameters “old_mod” and “new_mod”. The second function is a normal function to load a module. The Third function defines the different printing styles for Python 2 and Python 3. Then, from the fourth function, we can clearly see a “wrapper” is defined in the function “add_metaclass” in order to decorate a class to be applied in both Python 2 and 3.

6.2.2 Emergent Python Patterns

In fact, we find more emergent patterns than design patterns in Python projects. We group these patterns into five categories as follows.
File Handling Pattern

After analyzing all pattern reports, we categorize some patterns into file handling patterns according to their functionalities. Python is always being used for processing text files, such as xml and html. For instance, reading and writing to files are the most basic operations in file handling. Apart from reading and writing to files, there are also some other file operations. We provide a file handling pattern in Figure 6.10 to show some different file operations.

```python
def find(path, saltenv='base', env=None):
    ...'return a dict of the files located with the given path and environment...
    ...'
    if env is not None:
        salt.utils.warn_until('Boron',
            'Passing a salt environment should be done using \'saltenv\'
            'not \'env\'. This functionality will be removed in Salt Boron.'
        )
    # Backwards compatibility
    saltenv = env

    # Return a list of paths + text or bin
    ret = []
    if saltenv not in __opts__['file_roots']:
        return ret
    for root in __opts__['file_roots'][saltenv]:
        full = os.path.join(root, path)
        if os.path.isfile(full):
            # Add it to the dict
            with salt.utils.fopen(full, 'rb') as fp_:
                if salt.utils.is_textfile(fp_):
                    ret.append({'full': 'txt'})
                else:
                    ret.append({'full': 'bin'})
    return ret
```

(a) File Handling Pattern (ID:93) - Clone Class 1 of 4
def list_env(saltenv='base', env=None):
    """
    Return all of the file paths found in an environment
    """
    if env is not None:
        salt.utils.warn_until(
            'Boron',
            'Passing a salt environment should be done using \"saltenv\" ' +
            'not \"env\". This functionality will be removed in Salt Boron.'
        )
        # Backwards compatibility
        saltenv = env
    ret = ()
    if saltenv not in __opts__['file_roots']:
        return ret
    for f_root in __opts__['file_roots'][saltenv]:
        ret[f_root] = ()
        for root, dirs, files in os.walk(f_root):
            sub = ret[f_root]
            if root != f_root:
                # grab subroot ref
                sroot = root
                above = []
                # Populate the above dict
                while not os.path.samefile(sroot, f_root):
                    base = os.path.basename(sroot)
                    if base:
                        above.insert(0, base)
                        sroot = os.path.dirname(sroot)
                for aroot in above:
                    sub = sub[aroot]
                for dir_ in dirs:
                    sub[dir_] = ()
                for fn_ in files:
                    sub[fn_] = 'f'
    return ret

(b) File Handling Pattern (ID:93) - Clone Class 2 of 4
def read(path, saltenv='base', env=None):
    '''
    Read the contents of a text file, if the file is binary then
    '''
    if env is not None:
        salt.utils.warn_until('Boron',
            'Passing a salt environment should be done using \"saltenv\" '
            'not \"env\". This functionality will be removed in Salt Boron.'
        )
    # Backwards compatibility
    saltenv = env
    # Return a dict of paths + content
    ret = []
    files = find(path, saltenv)
    for fn in files:
        full = next(six.iterkeys(fn_))
        form = fn_[full]
        if form == 'txt':
            with salt.utils.fopen(full, 'rb') as fp_:
                ret.append((full, fp_.read()))
    return ret
In this pattern, we are able to find four functions which are being used for reading files, writing files, returning the paths and the dictionaries of files. Besides, there are also some other programming patterns which are applied to fetch, remove or update files. And we categorize all these patterns as “File Handling Pattern” according to their similar functionalities.
Data Operation Pattern

Apart from the file handling patterns we mentioned above, there are also some patterns which are contributed for changing data types and content. We categorize these as the “Data Operation Pattern”. In order to better introduce this pattern, we provide a pattern example in Figure 6.11.
def recursive_unicode(obj):
    """Walks a simple data structure, converting byte strings to unicode.
    Supports lists, tuples, and dictionaries."
    if isinstance(obj, dict):
        return dict((recursive_unicode(k), recursive_unicode(v)) for (k, v) in obj.items())
    elif isinstance(obj, list):
        return list(recursive_unicode(i) for i in obj)
    elif isinstance(obj, tuple):
        return tuple(recursive_unicode(i) for i in obj)
    elif isinstance(obj, bytes):
        return to_unicode(obj)
    else:
        return obj

def _convert_entity(m):
    if m.group(1) == "#":
        try:
            if m.group(2)[1].lower() == 'k':
                return unichr(int(m.group(2)[1:], 16))
            else:
                return unichr(int(m.group(2)))
        except ValueError:
            return "%%s" % m.group(2)
        try:
            return _HTML_UNICODE_MAP[m.group(2)]
        except KeyError:
            return "%%s" % m.group(2)
        except ValueError:
            return "%%s" % m.group(2)
        try:
            return _HTML_UNICODE_MAP[m.group(2)]
        except KeyError:
            return "%%s" % m.group(2)

(a) Data Operation Pattern (ID:12) - Clone Class 1 & 2 of 3
def linkify(text, shorten=False, extra_params="", require_protocol=False, permitted_protocols=["http", "https"])::
    """Converts plain text into HTML with links."
    For example: `linkify("Hello http://tornadoweb.org!")` would return
    `"Hello <a href="http://tornadoweb.org" http://tornadoweb.org/a !"`

Parameters:
* `\`shorten\``: Long urls will be shortened for display.
* `\`extra_params\``: Extra text to include in the link tag, or a callable
    taking the link as an argument and returning the extra text
    e.g. `\`linkify(text, extra_params='rel="nofollow" class="external"')``
    or:

    def extra_params_cb(url):
        if url.startswith("http://example.com"):
            return 'class="internal"
        else:
            return 'class="external" rel="nofollow"
    linkify(text, extra_params=extra_params_cb)
* `\`require_protocol\``: Only linkify urls which include a protocol. If
    this is false, urls such as www.facebook.com will also be linkified.
* `\`permitted_protocols\``: List (or set) of protocols which should be
    linkified, e.g. `\`linkify(text, permitted_protocols=["http", "ftp",
    "mailto"])`
    It is very unsafe to include protocols such as
    `\`javascript\``.
    """
    if extra_params and not callable(extra_params):
        extra_params = " " + extra_params.strip()

def make_link(m):
    url = m.group(1)
    proto = m.group(2)
    if require_protocol and not proto:
        return url # not protocol, no linkify
    if proto and proto not in permitted_protocols:
        return url # bad protocol, no linkify
    href = m.group(1)
    if not proto:
        href = "http:" + href # no proto specified, use http

(b) Data Operation Pattern (ID:12) - Clone Class 3 of 3 (Part I)
All these three functions are converting some strings. In the first function, the method is used to converting byte strings to Unicode. The second function is...
contributed to change the value of a group of strings. The third one is a function
called linkify, which is actually being used for converting plain text into a HTML with
links. Because all the functions in this pattern are coded for changing or converting
data of the program, we categorize these patterns as data operation pattern.

Package Management Pattern

Users are allowed to install any extended packages on Python according to their own
needs. Therefore, there should be some functions which are being used to manage all
those packages. From the NiPAT pattern report, we also find some patterns which
are being used for adding, updating or removing the packages from Python projects.
We provide an example in Figure 6.12 to illustrate package management patterns.
6.2. PATTERNS IN PYTHON PROJECTS

Class: 1 of 3 ...Code in NiCAD Clone Class: 857 ...Similarity in NiCAD origin Class: 70

```
def list_pkgs(versions_as_list=False, **kwargs):
    """
    List the packages currently installed in a dict:
    ('<package_name', '<version')
    
    CLI Example:
    .. code-block:: bash

        salt '*' pkg.list_pkgs

    versions_as_list = salt.utils.is_true(versions_as_list)
    # not yet implemented or not applicable
    if any([salt.utils.is_true(kwargs.get(x))
            for x in ('removed', 'purge_desired')]):
        return []
    if 'pkg.list_pkgs' in __context__:
        if versions_as_list:
            return __context__['pkg.list_pkgs']
        else:
            ret = copy.deepcopy(__context__['pkg.list_pkgs'])
            _salt__['pkg_resource.stringify'](ret)
            return ret

    ret = []
    cmd = ['rpm', '-qa', '--queryformat',
           salt.utils.pkg.rpm.QUERYFORMAT.replace('%(REPOID)', '', '(none)\n')]
    output = __salt__['cmd.run'](cmd,
                                  python_shell=False,
                                  output_loglevel='trace')
    for line in output.splitlines():
        pkginfo = salt.utils.pkg.rpm.parse_pkginfo(
            line,
            osarch=__grains__['osarch'])
        if pkginfo is not None:
            _salt__['pkg_resource.add_pkg'](ret,
                                            pkginfo.name,
                                            pkginfo.version)

    __salt__['pkg_resource.sort_pkglist'](ret)
    __context__['pkg.list_pkgs'] = copy.deepcopy(ret)
    if not versions_as_list:
        __salt__['pkg_resource.stringify'](ret)
    return ret
```

(a) Package Management Pattern (ID:11) - Clone Class 1 of 3
def remove(name=None, pkgs=None, **kwargs):  # pylint: disable=W0613
    """Remove packages

    name
    The name of the package to be removed

    Multiple Package Options:

    pkgs
    A list of packages to delete. Must be passed as a python list. The
    'name' parameter will be ignored if this option is passed.
    ...
    try:
        pkg_params = __salt__['pkg_resource.parse_targets'](name, pkgs)[0]
    except MinionError as exc:
        raise CommandExecutionError(exc)

    old = list(pkgs)
    targets = [x for x in pkg_params if x in old]
    if not targets:
        return ()

    out = __salt__['cmd.run_all']({
        _yum(): ['-y', 'remove'] + targets,
        output_loglevel='trace',
        python_shell=False
    })

    if out['retcode'] != 0 and out['stderr']:
        errors = [out['stderr']]  # errors = []
        __context__.pop('pkg.list_pkgs', None)
        new = list(pkgs())
        ret = salt.utils.compare_dicts(old, new)

        if errors:
            raise CommandExecutionError(
                'Error occurred removing package(s)',
                info={'errors': errors, 'change': ret}
            )
    return ret
There are three functions consisting of the pattern above. The first function is being used for displaying all the packages, which will list all the packages installed with their names and version numbers when it is called. The second function is being
6.2. PATTERNS IN PYTHON PROJECTS

used to remove the packages with the given names. And the last function is devoted
to upgrade the package from an old version to a new one.

Apart from the functions we mentioned above, there are also some other dif-
ferent functions which are being used to manage packages in Python projects. For
instance, listing the specific package information including the specific file and adding
a version lock to a package to make sure it will not be removed or updated directly.
Since all these functions are designed to manage the packages in Python projects, we
categorized all patterns implementing similar functionalities as package management
patterns.

Testing and Checking Pattern

Apart from the patterns above, we also find some patterns which implement testing
and checking functionalities in Python programs and we categorize them as the “Test-
ing and Checking Pattern”. For instance, some patterns we find are being used for
verifying whether Python environment is installed or not. Some patterns are aimed
to checking the parameters or other settings of the project. Some patterns are used
to checking whether required packages are installed or not. Others are being used to
check whether the id is as same as the previous one, and so on.
def _check_and_install_python(ret, python, default=False, user=None):
    """
    Verify that python is installed, install if unavailable
    """
    ret = __salt__['pyenv.install_python'](python, runas=user)
    if not ret['result']:
        if __salt__['pyenv.install_python'](python, runas=user):
            ret['result'] = True
            ret['changes'][python] = 'Installed'
            ret['comment'] = 'Successfully installed python'
            ret['default'] = default
        else:
            ret['result'] = False
            ret['comment'] = 'Could not install python.'
            return ret
    if default:
        __salt__['pyenv.default'](python, runas=user)
    return ret
def _check_and_uninstall_python(ret, python, user=None):
    """
    Verify that python is uninstalled
    """
    ret = _python_installed(ret, python, user=user)
    if ret['result']:
        _salt__['pyenv.default']['system', runas=user]
        if __salt__['pyenv.uninstall_python'](python, runas=user):
            ret['result'] = True
            ret['changes'][python] = 'Uninstalled'
            ret['comment'] = 'Successfully removed python'
            return ret
        else:
            ret['result'] = False
            ret['comment'] = 'Failed to uninstall python'
            return ret
    else:
        ret['result'] = True
        ret['comment'] = 'python {} is already absent'.format(python)

    return ret
We provide a pattern example in Figure 6.13, and there are three functions that are being used to check the Python environment. The function “check_and_install_python” is implemented to check whether Python environment is installed and install Python if it’s not in the system. The second function is used to check whether the Python
environment is successfully removed from the system. And the last function is being used for verifying whether the specified Python environment is removed successfully. According to the descriptions above, we categorized all the similar patterns as “Testing and Checking” patterns.

API Pattern

The API patterns are patterns which implement specific functions and we categorize them as API patterns. There are a lot of external python libraries which are being widely used in open source Python projects. For example, SAX, which is short for “Simple API for XML” is one of the popular API libraries and we are able to find the functions which are specific belong to SAX API in our pattern report. In the following Figure 6.14, we provide an example of SAX API pattern.
### 6.2. PATTERNS IN PYTHON PROJECTS

<table>
<thead>
<tr>
<th>Class: 1 of 2</th>
<th>Code in NiCAD Clone Class: 72</th>
<th>Similarity in NiCAD origin Class: 70</th>
</tr>
</thead>
</table>

```python
def startElement(self, name, attrs, connection):
    if name == "Parameters":
        self.parameters = ResultSet([('member', Parameter)])
        return self.parameters
    elif name == "Outputs":
        self.outputs = ResultSet([('member', Output)])
        return self.outputs
    elif name == "Capabilities":
        self.capabilities = ResultSet([('member', Capability)])
        return self.capabilities
    elif name == "Tags":
        self.tags = Tag()
        return self.tags
    elif name == 'NotificationARNs':
        self.notification_arns = ResultSet([('member', NotificationARN)])
        return self.notification_arns
    else:
        return None
```

(a) API Pattern (ID:23) - Clone Class 1 of 2
In Figure 6.14, there are two functions “StartElement” and “EndElement” in this pattern and they are methods defined in class “sax.handler.ContentHandler”. The first function is being used to signal the start of an element in non-namespace mode in XML files and the second function signals the end of an element in non-namespace mode. Therefore, we categorized this pattern as one of the Python “API patterns” because these two functions are specific to Python SAX API. Apart from that, we also find some other API patterns, such as pattern with methods “Serialize” and
“Deserialize”, which belongs to “PyYAML API”. And these two methods are being used to serialize data into a Python data structure.

6.2.3 Summary of all Python Patterns

From the two tables below, we find several points as follows. Similarly to Java Patterns, as we increase the threshold during code clone detection, the more code patterns we get from NiPAT. Applying “blind renaming” is always an important factor for the number of patterns. Secondly, emergent code patterns are much more common than known design patterns under all different experimental settings. Thirdly, when we set the code clone detection threshold to 0.1, no singleton patterns are detected. However, after we increase the threshold to 0.2 and 0.3, there is always just one singleton pattern detected. All of the Python patterns we analyzed above were generated with threshold 0.3 and blind renaming.

Table 6.5: A summary of all Python design patterns under different thresholds

<table>
<thead>
<tr>
<th>Pattern Category</th>
<th>Threshold</th>
<th>No. of Patterns (-)</th>
<th>No. of Patterns (+)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Singleton</td>
<td>0.1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0.2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0.3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Decorator</td>
<td>0.1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0.2</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>0.3</td>
<td>5</td>
<td>8</td>
</tr>
</tbody>
</table>
### 6.2. PATTERNS IN PYTHON PROJECTS

Table 6.6: A summary of all emergent Python patterns under different thresholds

<table>
<thead>
<tr>
<th>Pattern Category</th>
<th>Threshold</th>
<th>No. of Patterns (-)</th>
<th>No. of Patterns (+)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>File Handling</strong></td>
<td>0.1</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>0.2</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>0.3</td>
<td>12</td>
<td>16</td>
</tr>
<tr>
<td><strong>Data Operation</strong></td>
<td>0.1</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>0.2</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>0.3</td>
<td>15</td>
<td>20</td>
</tr>
<tr>
<td><strong>Package Management</strong></td>
<td>0.1</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>0.2</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>0.3</td>
<td>12</td>
<td>20</td>
</tr>
<tr>
<td><strong>Testing &amp; Checking</strong></td>
<td>0.1</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>0.2</td>
<td>11</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>0.3</td>
<td>16</td>
<td>24</td>
</tr>
<tr>
<td><strong>Python API</strong></td>
<td>0.1</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>0.2</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>0.3</td>
<td>10</td>
<td>14</td>
</tr>
<tr>
<td><strong>Unknown Patterns</strong></td>
<td>0.1</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>0.2</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>0.3</td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>
6.3 Pattern Comparisons

After categorizing and analyzing all the Java and Python patterns, we find two main differences between them. One difference is the categories of patterns and the other one is the size of patterns.

First, it is more likely to find instances of design patterns in Java projects than Python projects. We suppose the reason is because Python is not an exclusively object-oriented language and “class” is not really necessary for every Python project. However, Java programs are fully object-oriented and “class” is always required for Java programs. Thus, object-oriented design patterns are probably more being used for Java projects. Besides, after referring to some Python APIs, some design patterns are provided by Python core library. For example, singletons can be replaced with modules in Python and modules are naturally singletons because the Python interpreter creates the object once and once only [31]. That may be another reason why design patterns are more difficult to find in Python projects compared to Java projects.

Second, the size of Java code patterns is greater than the size of Python patterns. The size of patterns can be illustrated into two dimensions: total files in the set and the number of clone classes in the pattern. After comparing all patterns, we found that Java patterns are bigger than Python patterns. For example, the Java pattern which has the most Java files in a set has 23 files, but the biggest number of Python files in a set is 12. Similarly, compared to the biggest number of clone classes 23 from Java projects, the number is 11 when it comes to Python projects. The reason may be because Java is an exclusively object-oriented programming language. Each Java file is a class encapsulated with a set of functions. However, classes are not necessary
for all Python projects. Thus, programmers may not write too many functions in one file in order to better manage the whole project.

6.4 Summary

In this chapter, we first categorized and analyzed all the patterns we detected in Java projects and provided an example of each pattern in Section 6.1. Then, all the Python patterns were analyzed in Section 6.2. In the last Section 6.3, we compared the difference between Java and Python patterns and analyzes the possible causes of these differences.
Chapter 7

Conclusions and Future Work

In the final Chapter, we briefly summarize our study in Section 7.1. We point out the limitations of our study in Section 7.2 and the future work we may need to continue in Section 7.3.

7.1 Summary

In this study, we build upon previous work in detection of code patterns, and extended the NiPAT pattern detector to Python and Java projects. Rather than just detecting emergent code patterns, we also showed that we can uncover instances of known design patterns which are exclusively for object-oriented programming languages using the NiPAT pattern detector.

We presented an empirical study of code pattern analysis based on the NiPAT methodology. We first extended the code pattern detector NiPAT to make it suitable for Java and Python projects and the unique characteristics of these two programming languages. We varied different experimental settings in NiCad and NiPAT for our experiment to uncover different sets of patterns. Finally, we undertook an analysis of the reported code patterns for both Java and Python projects. We analyzed all the
pattern reports under different experimental settings and categorized them into two main categories, design patterns and emergent code patterns. As part of the analysis, we demonstrated each of the categories of patterns by providing pattern examples. To conclude, we compared the difference between patterns in Java and Python projects and discussed the possible causes of these differences.

7.2 Limitations

Although we tried a lot of different experimental settings and extended NIPAT for better code pattern detection, there are still some limitations that could be improved to make the study more authentic in the following three aspects.

7.2.1 The Number of Projects

As mentioned in Chapter 5, 79 Java projects and 30 Python projects were included in our empirical study. Although we got hundreds of patterns from these projects, we still want to have more open source projects involved in the study to get more different patterns. However, when more projects are brought into the study, the workload will be too much if we still analyze and categorize all these patterns manually. Thus, before bringing more projects to our study, more improvement of NiPAT is required, such as automation of pattern categorization.

7.2.2 Programming Languages

We studied code patterns in Java and Python projects in our empirical study. Although these two programming languages are the most two popular object-oriented programming languages, we still want to include more projects which are programmed
7.3. FUTURE WORK

in other programming languages such as C, C++ and C#. The more different projects
we study, the more different patterns we will uncover.

7.2.3 The Efficiency of Pattern analysis

Analyzing Patterns Manually: For now, all the patterns from pattern reports are
categorized and analyzed manually. It takes a lot of time to categorize all the patterns
because we need to first go through each pattern and then refer to the source file to
find the relationships between functions and classes to decide which pattern it is.
These patterns are difficult to analyze for people without any background knowledge
of code patterns.

Run time of Pattern Detection: It didn’t take a lot of time for the code clone
detection stage in our study. However, the run time of NiPAT during the pattern
detection stage was much longer than we expected. For instance, when we set the
threshold to 0.2 with blind renaming for Java projects, it took us about half an hour
to detect all the code clones but more than 18 hours to generate code patterns. Since
our study is based on 79 Java projects, the run time of NiPAT is much longer than
our anticipation and it could also be a limitation of our study.

7.3 Future Work

Since there are still some limitations in our study, we propose the following three
aspects that can be improved in future work.
7.3.1 Involving More Different Programming Languages

As we mentioned before, we only detect and analyze code patterns from Java and Python projects. However, if we have more time to improve the study, we would like to further explore some other projects in different programming languages, such as C++ and C#. And after detecting patterns from more different programming languages, we will find more different emerging code patterns and design patterns from all these projects.

7.3.2 Automatic Pattern Classification

Since patterns are classified manually and it takes a lot of time to do it, we may try to make the whole process automatic in the future. However, there are still a lot of things to consider about how to recognize the patterns according to some specific features of each pattern.

7.3.3 Detecting Antipatterns

Apart from detecting good practice, namely design patterns, we may also able to find some antipatterns from the pattern report which are considered as bad practice during coding. Thus, in the future, we may also extend NiPAT and use it to detect code antipatterns.
Bibliography


