Duplicated Code Refactoring Advisor (DCRA): a tool aimed at suggesting the best refactoring techniques of Java code clones

Relatore: Prof.ssa Francesca Arcelli Fontana
Studente: Francesco Zanoni
Correlatore: Dr. Marco Zanoni
Matricola: 072187

Anno accademico 2011/2012
To Barbara,
my sweet nightingale,
whose existence
is my lantern in the night.
If you shut the door to all errors,  
truth will be shut out.

Rabindranath Thakur
# Table of contents

1 Introduction .................................................................................................................. 1

2 Related works ............................................................................................................... 4

3 Duplicated code .......................................................................................................... 6
   3.1 Clone type .................................................................................................................. 6
   3.2 Clone granularity ....................................................................................................... 7
   3.3 Clone grouping ......................................................................................................... 8

4 Duplicated code detection ............................................................................................ 10
   4.1 Detection techniques ............................................................................................... 10
   4.2 Detector review ....................................................................................................... 11

5 DCRA approach foundation .......................................................................................... 14
   5.1 Locations and refactoring techniques ...................................................................... 14
   5.2 Refactoring technique implementation .................................................................... 16
   5.3 Qualitas Corpus analysis ......................................................................................... 18

6 Duplicated Code Refactoring Advisor (DCRA) ............................................................. 22
   6.1 Clone detector .......................................................................................................... 23
   6.2 Clone detailer ........................................................................................................... 25
   6.3 Refactoring advisor ................................................................................................. 27
   6.4 Refactoring advice aggregator ............................................................................... 30

7 Validation ...................................................................................................................... 32

8 Conclusions and future works ..................................................................................... 40

9 Appendices .................................................................................................................... 42
   9.1 Analyzed system pre-processing ............................................................................. 42
   9.2 Refactoring advisor decision tree tables .................................................................. 42
   9.3 Line change computation formulas ......................................................................... 49

References ....................................................................................................................... 52

Acknowledgements ......................................................................................................... 55
List of figures

Figure 3.1: Two clones example .............................................................. 6
Figure 3.2: Type 2 clones example .......................................................... 7
Figure 3.3: Type 3 clones example .......................................................... 7
Figure 5.1: Extract method + flag implementation example ....................... 17
Figure 6.1: Duplicated code data flow throughout DCRA components .......... 22
Figure 6.2: DCRA toy example: clone pair .............................................. 23
Figure 6.3: DCRA toy example: clone detector XML report ....................... 25
Figure 6.4: DCRA toy example: refactoring advisor decision tree visit .......... 28
Figure 6.5: DCRA toy example: refactoring advisor preview of clone pair post-refactoring configuration .......................................................... 29
Figure 7.1: Example of refactoring-worthy clone pair - BEFORE REFACTORING ......................... 35
Figure 7.2: Example of refactoring-worthy clone pair - AFTER REFACTORING ..................... 35
Figure 7.3: Example of FORM TEMPLATE METHOD suggestion substituted by PULL UP METHOD - BEFORE REFACTORING .......................................................... 37
Figure 7.4: Example of FORM TEMPLATE METHOD suggestion substituted by PULL UP METHOD - AFTER REFACTORING .......................................................... 37
Figure 7.5: Example of refactoring-unworthy clone pair - BEFORE REFACTORING .............. 38
Figure 7.6: Example of refactoring-unworthy clone pair - AFTER REFACTORING (NOT APPLIED) ......................................................................................... 39
Figure 9.1: SIBLING CLASS implementation cost whole formula .................. 51

List of charts

Chart 5.1: Clone pair location distribution of the Qualitas Corpus systems ........ 19
Chart 5.2: Clone pair location percentage distribution of Qualitas Corpus systems .......... 20
Chart 5.3: Clone pair location percentage distribution of Qualitas Corpus systems .......... 20
Chart 5.4: Clone pair type distribution of Qualitas Corpus systems .................. 20
Chart 5.5: Clone pair container kind distribution of Qualitas Corpus systems ........... 21
Chart 5.6: Clone pair container kind percentage distribution of Qualitas Corpus systems ..... 21
Chart 7.1: DCRA test system clone pair location distribution percentage .................. 32
Chart 7.2: DCRA test system clone pair type distribution percentage .................... 32
Chart 7.3: DCRA test system clone pair container kind distribution percentage ............ 32
List of tables

Table 2.1: Fowler’s refactoring suggestions...............................................................4
Table 2.2: Golomini’s refactoring suggestions ..........................................................5
Table 2.3: Giesecke’s refactoring suggestions..........................................................5
Table 3.1: Clone granularity criteria evaluations .......................................................8
Table 4.1: Considered clone detectors........................................................................11
Table 4.2: Considered clone detector details .............................................................12
Table 5.1: DCRA locations and refactoring suggestions .............................................15
Table 5.2: DCRA refactoring techniques ....................................................................16
Table 5.3: DCRA implementation steps of refactoring techniques .............................16
Table 5.4: Analyzed systems from Qualitas Corpus collection ....................................19
Table 5.5: Clone pair location and type distributions of Qualitas Corpus systems........19
Table 5.6: Clone pair location and container kind distributions of Qualitas Corpus systems .21
Table 6.1: Finally considered clone detector pros and cons ......................................24
Table 6.2: DCRA toy example: clone detailer output ................................................27
Table 6.3: DCRA toy example: refactoring advisor output ........................................29
Table 6.4: DCRA toy example: refactoring advice aggregator output ......................30
Table 7.1: Qualitas Corpus systems used in DCRA test ...........................................32
Table 7.2: DCRA test system applicability for clone pairs within same class .............34
Table 7.3: DCRA test system applicability for clone pairs between sibling classes .......34
Table 9.1: SAME METHOD location decision tree ....................................................43
Table 9.2: SAME CLASS location decision tree ..........................................................45
Table 9.3: SIBLING CLASS location decision tree .....................................................47
Table 9.4: SAME EXTERNAL SUPERCLASS location decision tree .......................49
Table 9.5: Unhandled location decision tree ..............................................................49
Table 9.6: Implementation step line-based cost formulas ..........................................50
1 Introduction

Defined by Fowler as “the number one of the stink parade” [1], clearly citing Kent Beck’s convention to name “code smells” certain recurring situations within source code, duplicated code involves all non-trivial software systems. The percentage of involved duplicated lines is usually estimated between 5% and 20%, sometimes reaching even 50% [2] [3].

Zibran and Roy clearly summarize pros and cons of duplicated code within the introduction of their survey [2]. Main pros are:

- decreased probability to introduce bugs within new code using flawless code fragments,
- time saving avoiding new algorithm design and implementation,
- component decoupling facilitating independent evolution of entities.

Main cons are:

- redundant code, especially unsuitable within embedded software systems,
- uncontrolled spread of yet-to-know bugs, resulting in heavy correction time cost when discovered,
- heavy update time cost, when modification of an important part of a fragment implies modification of all duplicated fragments.

Even if duplication may not always be avoided, it is considered a serious problem, mainly from a maintenance perspective.

Many works investigated in depth factors causing its insertion, taxonomies according to several criteria and detection techniques so far, but just a few works examined its management procedures [2]. Many sources suggest to fully delegate correction activities (a.k.a. “refactoring”) to developers’ experience and judgement [2] [3] and even Fowler and Golombingi, the main references of this project, assert the importance of the “human in the loop” concept [1] [4]. These assertions follow the awareness that every modification to a software system must consider and respect the founding design choices of that system. Furthermore, design choices are not easily represented within automated procedures. During duplicated code management, two decisional steps involve design aspects:

- **refactoring worthiness evaluation**, meaning the choice of which instances are worth to be refactored and which are not,
- **refactoring techniques selection**, meaning the choice of which technique should be applied to remove a duplication instance, once the instance has been evaluated as refactoring-worthy.

Design aspects are involved within these steps, because duplicated code refactoring means merging code fragments, and merging means moving fragments from their original locations to other locations, such as other functions/methods or even other classes. Code relocation is the real problem, because moving code means moving computational logic belonging to a specific entity of the system: relocation could break the original design coherence, lowering entity cohesion and moving responsibilities into unsuitable entities. Other lower level issues involve code fragment adaptations during merging phase: differences in algorithm steps,
variable declaration and usage, parameter set, return values must all be considered, because of their heavily conditioning correction cost and effort.

There is no unique solution to this problem, since all described aspects may be evaluated in different ways, according to correction purposes: maximum code length optimization differs from maximum system manageability optimization. Even refactoring technique selection is not unique, since each technique has its own pros and cons, involving both design and lower level aspects. The research of a suitable tradeoff between these two approaches summarizes all difficulties of correcting duplicated code, but no algorithm able to automatically compute the suitable tradeoff has been created yet, therefore no automatic solver exists yet. This confirms the need of the “human in the loop”.

Even if the human role may not be replaced by an algorithm, developers may be helped during the two decisional steps described before, partially automating refactoring procedure:

- during refactoring worthiness evaluation of each duplication, since several studies led to development of supporting tools able to automatically filter unworthy instances [5] [6];
- during refactoring technique selection, in two ways:
  - several studies led to development of supporting tools able to suggest refactoring techniques after instance characteristic analysis;
  - development environments usually implement automatic application of some basic refactoring techniques [7] [8], so refactoring techniques may be quickly assessed and unsuitable ones excluded.

Inspired by Golomingi’s diploma thesis [4], this work proposes a corrective approach [9] aiming both at filtering worthy instances from unworthy ones and suggesting the best refactoring techniques for worthy ones. As implementation of this approach, a tool named DCRA (Duplicated Code Refactoring Advisor) was developed in Java programming language. It only accepts Java application source code as input.

At the beginning DCRA was designed as DCRS (Duplicated Code Refactoring Selector), whose output would have been the only possible refactoring technique for each duplication. That purpose was not feasible, since best refactoring techniques cannot be established univocally.

Purpose change led to current approach, which aims to classify refactoring techniques and suggest (not “establish”) most suitable ones. A deep automatic analysis of each duplication detail and the following automatic evaluation of related refactoring techniques provide this classification. Literature does not provide any standardization of criteria needed for this evaluation, since no common agreement has ever been reached. As a consequence, two parameters have been selected as most relevant:

- code line number change,
- compliance with the three OOP\(^1\) paradigm foundation principles: encapsulation, inheritance and polymorphism.

Rationalization of the two parameters through a consistent numeric translation leads to evaluation automation, reducing decisional steps to simple numeric comparisons. DCRA is designed according to a modular pattern, in order to facilitate new feature additions: new

\(^1\) Object Oriented Programming.
suitable evaluation criteria for refactoring techniques may increase classification reliability, for instance.

Finally, since maximization of duplication correction is the main purpose of corrective approach, as described before, duplications must be conveniently filtered in order to exclude all refactoring-unworthy instances, letting developers focus on worthy ones. These filtering activities reduce further human involvement within refactoring procedure. Filtering involves two steps of the procedure:

- duplication detection, where all instances that do not need any evaluation are excluded right away,
- refactoring advising, where all instances whose pre-refactoring evaluation is better than post-refactoring are excluded.

Here is the structure of this document:

1. **Introduction**: duplicated code problem overview, solution conditioning factors and current project classification within all existing approach set;
2. **Related works**: studies providing inspiration and basic ideas for the current project, with common and uncommon aspects pointed out;
3. **Duplicated code**: duplicated code feature overview with all aspects considered during DCRA design and implementation;
4. **Duplicated code detection**: duplicated code detection characteristics, technique and tool overview;
5. **DCRA approach foundation**: general description of the current approach, with some details differentiating it from previous studies and Qualitas Corpus statistical analysis;
6. **Duplicated Code Refactoring Advisor (DCRA)**: design and working principles of DCRA, with a detailed description of data flow managed throughout all its components;
7. **Testing**: descriptive statistics of duplications detected within analyzed systems and related DCRA output, considerations about DCRA output and related issues;
8. **Conclusions and future works**: overall evaluation of DCRA and its approach, with some ideas involving further development;
9. **Appendices**: tables containing all data used by DCRA within decisional steps and system pre-processing procedure description.
2 Related works

Zibran and Roy’s survey [2] reports many duplicated code classifications and taxonomies, each considering different sets of duplication features. This work draws inspiration from duplication classification based on refactoring opportunities, above all the ones suggesting refactoring techniques depending on duplication positions within their class hierarchies. Three works established the main foundations of this work: Fowler’s book about refactoring [1] and Golomini’s [4] and Gieseke’s [10] diploma theses.

Fowler is cited by many other works and provides a short description of a generic approach. The few lines he dedicates to duplicated code are summarized by the Table 2.1:

<table>
<thead>
<tr>
<th>SITUATION</th>
<th>PROPOSED REFACTORING TECHNIQUES</th>
</tr>
</thead>
<tbody>
<tr>
<td>In the Same Class</td>
<td>• Extract Method</td>
</tr>
<tr>
<td>With a Sibling Class</td>
<td>• Extract Method &lt;br&gt; • Pull Up Method &lt;br&gt; • Form Template Method &lt;br&gt; • Substitute Algorithm</td>
</tr>
<tr>
<td>In Unrelated Classes</td>
<td>• Extract Class &lt;br&gt; • <em>Keep one code instance within the mostly related class, replacing all other instances with invocations</em></td>
</tr>
</tbody>
</table>

Table 2.1: Fowler’s refactoring suggestions

This approach is intentionally generic, since its author agrees with the common practice of delegating all solution choices and details to developers. The same approach is summarized and slightly extended by Zibran and Roy [2] by reporting the most commonly suggested techniques in recent years and by adding further highly situation-specific techniques.

Golomini’s approach is much more detailed and methodical [4]: a highly precise and motivated set of locations (“scenarios”) within class hierarchy is used to classify duplications, which are linked to a highly precise and motivated set of refactoring techniques, depending on their scenario. Only general refactoring selection criteria are then provided, committing further considerations to developers (see Table 2.2).

<table>
<thead>
<tr>
<th>SCENARIO</th>
<th>PROPOSED REFACTORING TECHNIQUES</th>
</tr>
</thead>
<tbody>
<tr>
<td>In the Same Method</td>
<td>• Extract Method</td>
</tr>
<tr>
<td>In the Same Class</td>
<td>• Extract Method &lt;br&gt; • Insert Method Call &lt;br&gt; • Form Template Method</td>
</tr>
<tr>
<td>With a Sibling Class²</td>
<td>• Pull Up Method &lt;br&gt; • Extract Method &lt;br&gt; • Substitute Algorithm &lt;br&gt; • Form Template Method &lt;br&gt; • Replace Subclass with Field &lt;br&gt; • Extract Superclass</td>
</tr>
<tr>
<td>With the Superclass</td>
<td>• Insert Super Call</td>
</tr>
</tbody>
</table>

Table 2.2: Golomini’s refactoring suggestions

² Two classes with the same superclass.
With an Ancestor
- Extract Method
- Pull Up Method
- Form Template Method

With a First Cousin
- Pull Up Method
- Form Template Method
- Extract Method
- Extract Superclass

In Common Hierarchy
- Pull Up Method
- Extract Method
- Form Template Method
- Extract Superclass

In Unrelated Classes
- Extract Class

Keep one code instance only within the mostly related class, replacing all other instances with invocations

Table 2.2: Golomini’s refactoring suggestions

As stressed before, Golomini’s strong point is the association of specific refactoring techniques to specific scenarios, whereas Fowler favours a much more generic approach. That is why Golomini’s diploma thesis is to be considered as the starting point of this project. The “human in the loop” concept still holds, since no assessment on actual technique applicability is made, but it is much more guided than in Fowler’s approach.

SUPREMO\(^5\), a tool implemented by Golomini himself, provides further help to developers through graphical representations of all entities involved and their relationships.

At last, Giesecke’s thesis \([10]\) summarizes Golomini’s approach by adapting its features to Java programming language. Giesecke selected function/method as the best duplication granularity \([10] [9]\) (see section 3.2), therefore scenarios and refactoring techniques are reduced. Moreover, he further generalizes several scenarios, only detailing those in Table 2.3:

<table>
<thead>
<tr>
<th>SCENARIO</th>
<th>PROPOSED REFACTORIZING TECHNIQUES</th>
</tr>
</thead>
<tbody>
<tr>
<td>In the Same Class</td>
<td>Extract method</td>
</tr>
<tr>
<td>With the Superclass</td>
<td>Extract method</td>
</tr>
<tr>
<td>In Common Hierarchy</td>
<td>Extract superclass (if needed) + Pull up method</td>
</tr>
</tbody>
</table>

Table 2.3: Giesecke’s refactoring suggestions

To be precise, Table 2.3 does not report Giesecke’s exact terminology, but Fowler’s one \([1]\), much more commonly used. Giesecke suggests a technique called “PseudoExtractMethod”, a customized version of “Extract method”.

As described, DCRA approach mainly draws inspiration from Golomini’s thesis, by extending his duplication classification categories and by adapting his approach to Java programming language features. Furthermore, automated evaluation, classification and suggestion of refactoring techniques is added.

---

\(^3\) Two classes subclassing sibling classes.

\(^4\) Two classes with a common ancestor class.

\(^5\) SUPport for REfactoring Method Objects \([4]\).
3 Duplicated code

Duplicated code research field is very wide, but some concepts are shared by all literature sources.

The first and most important is the concept of “clone”: a code fragment duplicated in several locations within a software system with several similarity degrees. “Cloning” is therefore a synonym of “duplicated code”, both identifying the activity of introducing clones of a code fragment within a software system. Anyway, a shared definition of “similarity” does not exist, resulting in the lack of a rigorous definition of clone [2] [3].

Two Java source code clones in Figure 6.1, as toy example:

```java
public class MyClass {
    public int method1() {
        int a = 0;
        int b = 1;
        a++;
        b++;
        return a + b;
    }

    public void method2() {
        int a = 0;
        int b = 1;
        a++;
        b++;
        System.out.print(a + b);
    }
}
```

Figure 3.1: Two clones example

Two other concepts, strictly related with the clone concept, provide further details about its nature: clone type and clone granularity.

3.1 Clone type

Clone type describes a clone classification involving the amount and the way a code fragment is duplicated. The most commonly accepted classification is (each type has less strict constraints than the previous type) [2]:

- **type 1**: identical code fragments, only white space differences allowed;
- **type 2**: code fragments with identical structure and syntax, with identifier, literal and type renaming allowed;
- **type 3**: code fragments with added, removed or modified statements;
- **type 4**: code fragments implementing the same algorithm in different ways.

Clones in Figure 3.1 are type 1, while examples of type 2 and type 3 clones follow in Figure 3.2 and Figure 3.3.
No example of type 4 is reported, since it may occur in many different ways.

Type 2 clone definition must be carefully considered: since only structural and syntactic matching is required, statements such as `System.out.println()` and `String.class.getName()` are detected as matching, even if semantically totally incompatible. This may lead to a high amount of unmanageable duplication detections.

### 3.2 Clone granularity

Granularity identifies the level of the smallest syntactic structure at which clones are searched during detection: it is one of the most relevant clone detector input parameters (see chapter 4). This concept may vary deeply according to each programming language and its own syntactic structures [2]. The most common ones:
- file,
- class,
- function/method,
- block: set of statements enclosed by specific language elements (braces, in languages derived from C), defining internally declared variable scope; it is a superset of function/method;
- arbitrary set of statements.

Granularity selection is not an easy task, since it consists of a tradeoff between duplication length and features. About this, Giesecke suggests four clone fragment evaluation criteria [9] that allow developers to choose consciously: granularity level satisfying the highest amount of criteria is the best choice, depending on each situation. Zibran and Roy refer to these criteria within their survey, also adding some considerations and further details coming from their own experience [2]:

- coverage: code fragments should cover maximal behavioural aspects of analyzed systems;
- significance: code fragments should implement a significative functionality;
- intelligibility: the purpose of each code fragment should be easily understandable;
- reusability: each code fragment should foster generic reuse, no matter how it is achieved.

Even if Giesecke does not mention file- and block-level granularities [10] [9], Zibran and Roy extended his evaluation criteria to all granularities listed before, as shown in Table 3.1:

<table>
<thead>
<tr>
<th>GRANULARITY</th>
<th>COVERAGE</th>
<th>SIGNIFICANCE</th>
<th>INTELLIGIBILITY</th>
<th>REUSABILITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>file/class</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>function/method/block</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>arbitrary set of statements</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

Table 3.1: Clone granularity criteria evaluations

As a consequence, Zibran and Roy suggest function/method- and block-level granularities as the most suitable ones. Since Giesecke does not consider block-level granularity, he agrees with Zibran and Roy by favouring function/method-level granularity [9] [10], while block-level was chosen by Higo et al. when creating CCShaper, an output post-processor of CCFinder clone detector [5] [6].

### 3.3 Clone grouping

One more aspect is to be considered, more strictly related to clone detection than to duplicated code itself: clone grouping. Clone grouping motivations and implications are easily explained if duplication is considered as a mathematical relationship among clones [2]. Under specific circumstances, properties defining an equivalence relationship hold on clone relationship:

- reflexivity: A is a clone of itself (holding on all clones);
- symmetry: if A is a clone of B, then B is a clone of A (holding on all clones);
• transitivity: if A is a clone of B and B is a clone of C, then A is a clone of C (holding only on type 1 and type 2 clones).

Clone detectors usually group clones together by sets wherein clone relationship holds. Most common clone groupings follow:

• clone pairs,
• clone classes.

Clone pairs are much easier to manage when considered singularly, but they don’t provide a general overview of all entities containing the clones, with all their related issues and characteristics. On the other hand, clone classes are harder to manage, but they provide all information needed to plan a suitable refactoring strategy, since this way all aspects are considered.

Another issue involves clone grouping by pairs: single clone reference amount increases according to binomial coefficient formula, since two clones form a pair, three clones form three pairs, four clones form six pairs, and so on. A heavy information redundancy follows, but it may be avoided:

• before clone detection, conveniently configuring clone detector parameters (if available);
• after clone detection, post-processing detection output by reducing detected clone pair amount to only essential ones; this approach was chosen by Golomingi [4].

Anyway, conversion of clone pairs to clone classes and vice versa is achieved by applying set operations to clone sets.
4 Duplicated code detection

Roy and Cordy’s survey [3] is an important source about duplicated code, it provides a thorough overview of the problem, its causes and its implications. The large amount of related studies provided a large amount of tools aiming at detecting duplications (a.k.a. “clone detectors”), each one featuring its own functionalities and characteristics. Therefore clone detector selection must consider several conditioning factors, first of all the concepts described at chapter 3:

- detected clone types,
- detection clone granularity,
- output clone grouping.

Other unavoidable conditioning factors:

- supported programming languages,
- detection technique.

Supported programming language factor seems trivial, but it is actually strictly related to detection technique, since supported language set depends on detection technique. At section 4.1 the most common detection techniques are described, in order to provide an overview of how clone detection actually works, while at section 4.2 a list of tools implementing these techniques is reported.

4.1 Detection techniques

Detection techniques identify the different ways software systems are analyzed [2]. A first classification may be based on which physical entities are actually considered:

- techniques working on data derived from source code,
- techniques analyzing source code directly.

First group may be further classified into two subcategories:

- techniques comparing metrics (metric-based approach): similar code fragments feature similar metric values;
- techniques comparing application binary code (e.g. bytecode, if considering Java programming language): since similar source code fragments are usually compiled into similar binary entities.

Techniques comparing metric or binary code were not considered within the current project: metric comparison is suitable only for function/method-level granularity, while binary code comparison is not always reliable, since biunivocal correspondence between source code logical entities and binary code logical entities is not guaranteed [2].

On the other hand, source code targeted techniques provide a much richer subclassification based on abstraction level:

- textual comparison, character by character (text-based);
• *lexical* comparison, token by token\(^6\) (token-based);
• *syntactic* comparison, achieved by comparing syntactical trees (AST\(^7\)/tree-based),
• *semantic* comparison, achieved by comparing data elaboration flows (PDG\(^8\)-based).

In Table 4.1 all pros and cons of each detection technique are listed, together with a short description of how source code is represented during detection procedure [3]:

<table>
<thead>
<tr>
<th>DETECTION TECHNIQUE</th>
<th>SOURCE CODE REPRESENTATION</th>
<th>PROs</th>
<th>CONs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text-based</td>
<td>code line sequences</td>
<td>programming language and code correctness independence</td>
<td>equivalent but structurally different code fragments may be not detected as clones</td>
</tr>
<tr>
<td>Token-based</td>
<td>token sequences</td>
<td>programming language and code correctness independence and correct management of lexically equivalent code fragments</td>
<td>equivalent but syntactically different code fragments may be not detected as clones</td>
</tr>
<tr>
<td>AST/tree-based</td>
<td>token trees</td>
<td>correct management of syntactically equivalent code fragments</td>
<td>programming language and code correctness dependence</td>
</tr>
<tr>
<td>PDG-based</td>
<td>data elaboration and control flow graphs</td>
<td>equivalent data elaboration flows are detected, even if contained within syntactically different code fragments</td>
<td>programming language and code correctness dependence, detection of many unmanageable clones since too fragmented</td>
</tr>
</tbody>
</table>

Table 4.1: Clone detection technique details

As said before, the language set supported by a clone detector depends on its detection technique: a clone detector featuring a text-based technique, for instance, is very likely to support a large set of programming languages.

### 4.2 Detector review

All duplicated code and clone detection aspects described so far are necessary in order to consciously choose the most suitable clone detector depending on each situation. This section reports an overview of eight clone detectors among which DCRA clone detector was chosen. The list of tools was compiled striving to satisfy the following criteria:

• Java programming language support,
• command line tools rather than graphical ones or IDE\(^9\), since they are much easier to integrate within other applications,
• easily parsable output format and adequate position details allowing to precisely locate clones within source code,
• the largest amount of detected clone type combinations,

---

\(^6\) The smallest element of a program that is meaningful to the compiler.

\(^7\) Abstract Syntax Tree.

\(^8\) Program Dependence Graph.

\(^9\) Integrated Development Environment.
the largest amount of featured detection techniques,
highly customizable detection,
recent release.

All considered clone detectors are reported in Table 4.2 and Table 4.3, together with some details evaluated to choose the one used within DCRA.

<table>
<thead>
<tr>
<th>CLONE DETECTOR</th>
<th>VERSION</th>
<th>RELEASE DATE</th>
<th>DETECTION TECHNIQUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Checkstyle [13]</td>
<td>5.5</td>
<td>November 2011</td>
<td>Text-based</td>
</tr>
<tr>
<td>DuDe [14]</td>
<td>1.0</td>
<td>2005</td>
<td>Text-based</td>
</tr>
<tr>
<td>NiCad [15]</td>
<td>3.4</td>
<td>September 2012</td>
<td>Hybrid: tree- and text-based</td>
</tr>
<tr>
<td>PMD [16]</td>
<td>5.0.0</td>
<td>May 2012</td>
<td>Token-based</td>
</tr>
<tr>
<td>Scorpio [17]</td>
<td>201103030634</td>
<td>March 2011</td>
<td>PDG-based</td>
</tr>
<tr>
<td>Simian [18]</td>
<td>2.3.33</td>
<td>August 2011</td>
<td>Text-based</td>
</tr>
</tbody>
</table>

Table 4.2: Considered clone detectors

<table>
<thead>
<tr>
<th>CLONE DETECTOR</th>
<th>PROGRAMMING LANGUAGES</th>
<th>CLONE TYPES</th>
<th>OUTPUT FORMAT AND DETAILS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bauhaus Suite</td>
<td>C, C++, C#, Java, Visual Basic, Ada, Cobol</td>
<td>Type-1, Type-2</td>
<td>Text: files with full path, start and end lines, clone type, number of cloned lines, number of cloned tokens</td>
</tr>
<tr>
<td>Checkstyle</td>
<td>Java</td>
<td>Type-1</td>
<td>Text, XML: number of cloned lines, files with full path, clone start line numbers</td>
</tr>
<tr>
<td>DuDe</td>
<td>Language independent, but best results with Java, C, C++</td>
<td>Type-1, Type-2, Type-3</td>
<td>XML, DUP: type, clone structure and length, files with full path, clone start and end lines</td>
</tr>
<tr>
<td>IntelliJ IDEA</td>
<td>Java, Groovy, JavaScript, XML, (X)HTML, CSS, Scala, Flex, AIR, ActionScript, XSL, Ruby, JRuby, SQL, FreeMarker, PHP, Velocity, ColdFusion</td>
<td>Type-1, Type-2, Type-3</td>
<td>HTML site: number of clones, cost, files with full path, clone start and end lines, class and package</td>
</tr>
<tr>
<td>NiCad</td>
<td>C, Java, Python, C#, WSDL</td>
<td>Type-1, Type-2, Type-3</td>
<td>XML, HTML: clone pair length, similarity percentage, file paths, start and end lines, source code</td>
</tr>
<tr>
<td>PMD</td>
<td>Java</td>
<td>Type-1, Type-2</td>
<td>CSV, Text, XML: number of cloned lines, number of cloned tokens, files with full path, clone start line numbers, code</td>
</tr>
<tr>
<td>Scorpio</td>
<td>Java</td>
<td>Type-1, Type-2, Type-3</td>
<td>XML: files with full path, lines of code, number of PDG nodes, duplicated ratio, methods with position (lines), lines and columns of each duplicated expression, number of gaps</td>
</tr>
<tr>
<td>Simian</td>
<td>Java, C#, C++, C, Objective-C, JavaScript, COBOL, ABAP, Ruby, Lisp, SQL, Visual Basic, Groovy, JSP, ASP, HTML, XML</td>
<td>Type-1, Type-2</td>
<td>Text, XML, emacs, vs (Visual Studio), YAML: number of cloned lines, files with full path, clone start and end line numbers, summary (code, if specified as parameter)</td>
</tr>
</tbody>
</table>

Table 4.3: Considered clone detector details
Other well-known clone detectors, not listed before, were not considered at all:

- CodePro Analytix [19], excluded because no clone start or end line was provided within output report;
- iPlasma [20], excluded because its clone detection engine is DuDe;
- inFusion [21], successors of iPlasma, excluded because of their mainly graphical use and FAMIX [22] output format, whereas XML or CSV were preferred.

The whole selection procedure of DCRA clone detector is reported at section 6.1, by describing how all considerations on duplicated code, refactoring and detection techniques led to choose NiCad.
5 DCRA approach foundation

This work proposes an approach aimed at automating correction technique evaluation and selection, starting from duplication classification and resulting in a concrete reduction of human involvement during refactoring procedures. The whole procedure is summarizable as follows:

- Golomini’s [4] scenario set extension with further recurring locations (within this work scenarios are simply named as “locations”),
- exclusion of the most problematic duplications (type 2) during clone detection,
- analysis of characteristics of each duplication, resulting in a specific set of applicable refactoring techniques,
- automatic evaluation of a specific set of refactoring applicability criteria for each duplication, in order to classify applicable techniques by suitability,
- aggregation of information about most suitable refactoring techniques, in order to identify the most problematic parts of the analyzed system.

Automatic applicability evaluation of refactoring techniques is the strong point of this work, since is the aspect that aims at reducing involvement of the “human in the loop”. As introduced at chapter 1, automatic evaluation is achieved by defining a specific set of criteria and by translating them into suitable numeric values. DCRA, the tool implementing this approach, is designed to be easily extendable, since future deeper knowledge may suggest the need of further criteria, further locations or further refactoring techniques.

This first DCRA version evaluates refactoring techniques according to two criteria:

- code line number variation,
- OOP founding principles compliance: encapsulation, inheritance and polymorphism.

These criteria were first introduced at chapter 1, while a detailed description of numeric value assignment procedure is reported at section 6.3.

5.1 Locations and refactoring techniques

Since locations are the starting point of the whole procedure and refactoring techniques are the ending point, Table 5.1 outlines all these elements and the way they are related:

<table>
<thead>
<tr>
<th>LOCATION</th>
<th>PROPOSED REFACTORIZING TECHNIQUES</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAME METHOD</td>
<td>• Extract method</td>
</tr>
<tr>
<td></td>
<td>• Replace method with method object</td>
</tr>
<tr>
<td></td>
<td>• Leave unchanged</td>
</tr>
<tr>
<td>SAME CLASS</td>
<td>• Extract method</td>
</tr>
<tr>
<td></td>
<td>• Replace method with method object</td>
</tr>
<tr>
<td></td>
<td>• Merge method</td>
</tr>
<tr>
<td></td>
<td>• Leave unchanged</td>
</tr>
<tr>
<td>SIBLING CLASS</td>
<td>• Pull up method</td>
</tr>
<tr>
<td></td>
<td>• Pull up method object</td>
</tr>
<tr>
<td></td>
<td>• Form template method</td>
</tr>
</tbody>
</table>

14
SAME EXTERNAL SUPERCLASS is a new location, not considered by Golomingi [4]: it describes all clones located within sibling classes extending a common external class, meaning a class belonging to system libraries or to JRE\textsuperscript{10}. This addition is significant: a dedicated survey on 50 software projects included within Qualitas Corpus collection [23] revealed that over one third of all detected duplications is related to this location (36.02\%, see section 5.3 for details). Golomingi’s approach would classify those instances as UNRELATED CLASS, therefore hardly manageable through an automatic procedure [4]. On the contrary, DCRA validation phase revealed those instances usually belong to semantically related classes, therefore manageable with the same strategies suitable for instances related to SIBLING CLASS location (very small differences, see sections 6.3 and 9.2). Anonymous classes are very recurring examples of those instances, since they usually extend class \texttt{Object}.

Anyway, some of Golomingi’s and Fowler’s refactoring suggestions were not taken into account:

- Substitute algorithm, since type 4 clones are not considered by this approach, as reported at section 6.1;
- Push down method, since this first DCRA version does not manage SUPERCLASS location, as said in following paragraph.

This first DCRA version suggests the most suitable refactoring techniques for only the first four locations, the most recurrent ones. Location frequencies are reported in detail within the survey on Qualitas Corpus at section 5.3, where more than 75\% of all detected instances are reported to be related to the first four locations. Also Fowler’s suggestions are mainly related to the first four locations [1]. All other locations are simply suggested LEAVE UNCHANGED: even if it might seem strange within a corrective approach, refactoring criteria evaluation procedure, described before and detailed at section 6.3, may sometimes lead to consider unmodified configurations as more suitable than technique applications.

All refactoring techniques listed in Table 5.1 are generic suggestions, roughly corresponding to Golomingi and Fowler’s opinions, but the current approach takes into account concrete implementation, too, since refactoring evaluation is heavily based on it. At section 5.2 all refactoring implementative steps are detailed, together with a clear explanation of the need to consider techniques and steps separately.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
\textbf{SAME EXTERNAL SUPERCLASS} & Leave unchanged \\
\hline
Pull up method & Leave unchanged \\
Pull up method object & Leave unchanged \\
Form template method & Leave unchanged \\
\hline
\textbf{SUPERCLASS} & Leave unchanged \\
\hline
\textbf{ANCESTOR} & Leave unchanged \\
\hline
\textbf{FIRST COUSIN} & Leave unchanged \\
\hline
\textbf{COMMON HIERARCHY} & Leave unchanged \\
\hline
\textbf{UNRELATED CLASSES} & Leave unchanged \\
\hline
\end{tabular}
\caption{DCRA locations and refactoring suggestions}
\end{table}

\textsuperscript{10} Java Runtime Environment.
5.2 Refactoring technique implementation

Refactoring techniques at section 5.1 are generic terms indicating the generic strategy to apply in order to correct clone pairs, as used in all cited literature sources [2] [3] [1] [4]. Most technique names follow Fowler’s suggestions [1], as reported in Table 5.2:

<table>
<thead>
<tr>
<th>REFAC TORING TECHNIQUE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extract method</td>
<td></td>
</tr>
<tr>
<td>Replace method with method object</td>
<td>also known as “Extract utility class” [2], considered in its simplest version (private or protected class with public attributes and only one method: the constructor), much shorter than Fowler’s one</td>
</tr>
<tr>
<td>Merge method</td>
<td></td>
</tr>
<tr>
<td>Pull up method object</td>
<td>not cited by any source, but clearly following “Replace method with method object”</td>
</tr>
<tr>
<td>Form template method</td>
<td></td>
</tr>
<tr>
<td>Leave unchanged</td>
<td>sometimes it may be the best refactoring</td>
</tr>
</tbody>
</table>

Table 5.2: DCRA refactoring techniques

Depending on the scenario (within DCRA meaning combination of location, container kind, type, number of variables/class attributes modified within clone code) and on other context details, each technique must be implemented in a precise way. That is where refactoring implementation steps are important: they are the bridge linking theory to practice, meaning they are the exact practical operations to be executed in order to apply a correction generally described by a refactoring technique name. All refactoring implementation steps of this first DCRA version in Table 5.3:

<table>
<thead>
<tr>
<th>IMPLEMENTATION STEP</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extract method</td>
<td></td>
</tr>
<tr>
<td>Return</td>
<td>needed if a variable or attribute is changed and must be returned</td>
</tr>
<tr>
<td>Replace method with method object</td>
<td>needed if a conditional branch must be introduced within refactoring execution of type 3 clone pairs</td>
</tr>
<tr>
<td>Merge method</td>
<td>simply meaning “delete one of the two identical copies of a method”</td>
</tr>
<tr>
<td>Pull up method</td>
<td></td>
</tr>
<tr>
<td>Pull up method object</td>
<td></td>
</tr>
<tr>
<td>Form template method</td>
<td></td>
</tr>
<tr>
<td>Create superclass</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.3: DCRA implementation steps of refactoring techniques

It is worth pointing out that most implementation step names coincide with refactoring technique names. This is likely to be confusing, but the reason to keep these names is the strong semantic relationship holding between the two entities: for instance, if “Pull up method” (refactoring technique) is needed in order to refactor a block clone pair between two sibling classes, that block will be “Extract method”-ed and “Pull up method”-ed (implementation steps). However implementation step names may be easily changed to something more precise, if needed, since they’re implemented as simple enumerations.

“Flag” implementation step needs further explanation, in order to be fully understood. It involves the situation where “Extract method” is applied to type 3 clones: in order to create
a method usable from all original positions, a flag parameter is added to extracted method signature and used by conditional branches within the extracted method body, as shown in Figure 5.1.

```java
public class MyClass {
    public int method1() {
        this.extractedMethod(0);
    }
    public void method2() {
        this.extractedMethod(1);
    }
    private void extractedMethod(int flag) {
        long c = 2;
        long d = 3;
        if (flag == 0) {
            c++;
        }
        if (flag == 1) {
            d++;
        }
        System.out.print(c + d);
    }
}
```

Figure 5.1: Extract method + flag implementation example

This kind of refactoring approach is not described in literature and Roy and Cordy’s survey [3] implicitly reports only type 1 clones may be refactored with “Extract method” technique, since class interface actually changes, if a flag parameter is introduced, and risk of encapsulation level reduction increases. Anyway, during DCRA approach development further considerations led to include this technique:

- very easy to apply, with minimal effort;
- if applied within a single class, class external interface does not change, hence original encapsulation level still holds;
- if applied between a superclass and its subclasses, together with a “Pull up method” application, extracted method is given “protected”\(^\text{11}\) visibility, therefore class hierarchy external interfaces, if considered as a whole, do not change and original encapsulation level still holds;
- as reported by Qualitas Corpus analysis (section 5.3), type 3 clone pairs within the same class are about 28% of total clone pair amount and type 3 clone pairs between sibling classes are about 40%: while the latter are anyway refactorable with other heavier but more OOP-compliant techniques (i.e. “Form template method”), the former must be left unchanged, if no flag parameter is introduced.

The extremely practical nature of implementation steps makes them the base of post-refactoring line number change computation, since each step has its own weight computation, whose result depends on clone pair details such as number of cloned and

\(^{11}\) “The protected modifier specifies that the member can only be accessed within its own package (as with package-private) and, in addition, by a subclass of its class in another package.” [36]
different line, number of different fragments, number of change-conditioning elements (see section 9.3).

5.3 Qualitas Corpus analysis

Design effort of the first version of DCRA focused on the most recurring characteristics featured by duplication instances. In order to determine which are the most recurring characteristics, a statistical assessment was achieved by analyzing 50 software projects included within Qualitas Corpus collection [23]. Qualitas Corpus is a set of more than a hundred open-source software projects written in Java programming language (at least till version 20120401, here used). Its purpose is providing researchers a standardized basis on which reproducible studies may be conducted [24]. Software projects were analyzed by the selected clone detector, NiCad (see section 6.1 for details), and the preliminary version of the clone detailer, the second software component of the whole DCRA procedure.

This assessment analyzed several aspects of duplicated code described so far: clone pair types, clone pair locations and clone pair container kind. “Container kind” corresponds to the concept of granularity (section 3.2). Clones are grouped in pairs, as this grouping has been chosen since this first DCRA design stages.

A rough estimate of analyzed system size and their duplication affection degree is reported by Table 5.4 through code line and clone pair amount of each system. Code lines were computed by the specific tool CLOC 1.56 [25], after all system pre-processing described at section 9.1.

<table>
<thead>
<tr>
<th>PROJECT</th>
<th>LINES OF CODE</th>
<th>CLONE PAIRS</th>
</tr>
</thead>
<tbody>
<tr>
<td>aoi-2.8.1</td>
<td>75488</td>
<td>996</td>
</tr>
<tr>
<td>apache-ant-1.8.2</td>
<td>97059</td>
<td>5383</td>
</tr>
<tr>
<td>batik-1.7</td>
<td>158594</td>
<td>5219</td>
</tr>
<tr>
<td>checkstyle-5.1</td>
<td>17151</td>
<td>182</td>
</tr>
<tr>
<td>c-jdbc-2.0.2</td>
<td>59119</td>
<td>2711</td>
</tr>
<tr>
<td>cobertura-1.9.4.1</td>
<td>50615</td>
<td>55469</td>
</tr>
<tr>
<td>colt-1.2.0</td>
<td>37313</td>
<td>3281</td>
</tr>
<tr>
<td>drawswf-1.2.9</td>
<td>24437</td>
<td>225</td>
</tr>
<tr>
<td>drjava-20100913-r5387</td>
<td>91741</td>
<td>9819</td>
</tr>
<tr>
<td>findbugs-1.3.9</td>
<td>93484</td>
<td>643</td>
</tr>
<tr>
<td>fitjvav-1.1</td>
<td>3643</td>
<td>27</td>
</tr>
<tr>
<td>freecol-0.10.3</td>
<td>87971</td>
<td>1810</td>
</tr>
<tr>
<td>freecs-1.3.20100406</td>
<td>22479</td>
<td>544</td>
</tr>
<tr>
<td>freemind-0.9.0</td>
<td>38856</td>
<td>211</td>
</tr>
<tr>
<td>galreon-2.3.0</td>
<td>53819</td>
<td>15657</td>
</tr>
<tr>
<td>ganttproject-2.0.9</td>
<td>41120</td>
<td>534</td>
</tr>
<tr>
<td>hadoop-1.0.0</td>
<td>48356</td>
<td>612</td>
</tr>
<tr>
<td>heritrix-1.14.4</td>
<td>56626</td>
<td>987</td>
</tr>
<tr>
<td>hsqldb-2.0.0</td>
<td>131946</td>
<td>6498</td>
</tr>
<tr>
<td>htmlunit-2.8</td>
<td>38120</td>
<td>393</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PROJECT</th>
<th>LINES OF CODE</th>
<th>CLONE PAIRS</th>
</tr>
</thead>
<tbody>
<tr>
<td>jfreechart-1.0.13</td>
<td>82305</td>
<td>8076</td>
</tr>
<tr>
<td>jgraph-5.13.0.0</td>
<td>21039</td>
<td>130</td>
</tr>
<tr>
<td>jgraphpad-5.10.0.2</td>
<td>21980</td>
<td>237</td>
</tr>
<tr>
<td>jgraph-0.8.1</td>
<td>9086</td>
<td>53</td>
</tr>
<tr>
<td>jgroups-2.10.0.GA</td>
<td>60501</td>
<td>3107</td>
</tr>
<tr>
<td>jhotdraw-7.5.1</td>
<td>75470</td>
<td>1826</td>
</tr>
<tr>
<td>jmoney-0.4.4</td>
<td>6328</td>
<td>166</td>
</tr>
<tr>
<td>jpf-1.0.2</td>
<td>7644</td>
<td>148</td>
</tr>
<tr>
<td>jruby-1.5.2</td>
<td>153158</td>
<td>4404</td>
</tr>
<tr>
<td>jspwiki-2.8.4</td>
<td>33607</td>
<td>624</td>
</tr>
<tr>
<td>jsXe-0.4.99_0</td>
<td>8560</td>
<td>102</td>
</tr>
<tr>
<td>lucene-3.5.0</td>
<td>60740</td>
<td>1139</td>
</tr>
<tr>
<td>mvnforum-1.2.2-ga</td>
<td>46672</td>
<td>4396</td>
</tr>
<tr>
<td>nekohtmlm-1.9.14</td>
<td>6149</td>
<td>81</td>
</tr>
<tr>
<td>oscache-2.4.1</td>
<td>5786</td>
<td>70</td>
</tr>
<tr>
<td>pmd-4.2.5</td>
<td>50435</td>
<td>9697</td>
</tr>
<tr>
<td>poi-3.6</td>
<td>64635</td>
<td>717</td>
</tr>
<tr>
<td>proguard-4.5.1</td>
<td>37387</td>
<td>788</td>
</tr>
<tr>
<td>quartz-1.8.3</td>
<td>21416</td>
<td>2233</td>
</tr>
<tr>
<td>sablecc-3.2</td>
<td>22766</td>
<td>7052</td>
</tr>
</tbody>
</table>
As reported by Table 5.4, all systems are medium-to-low sized and clone pair distribution is not proportional to system size.

By crossing clone pair location and type distribution data, it is worth noting how location distribution reflects code fragment semantic affinity: locations representing much semantically closer entities (SAME METHOD, SAME CLASS, SIBLING CLASS and SAME EXTERNAL SUPERCLASS) are also the most recurrent clone pair locations (more than 75%) (Table 5.5, Chart 5.1 and Chart 5.2).

<table>
<thead>
<tr>
<th>Clone Pair Location</th>
<th>One</th>
<th>Three</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANCESTOR CLASS</td>
<td>13</td>
<td>281</td>
</tr>
<tr>
<td>COMMON HIERARCHY CLASS</td>
<td>970</td>
<td>3152</td>
</tr>
<tr>
<td>FIRST COUSIN CLASS</td>
<td>416</td>
<td>2980</td>
</tr>
<tr>
<td>SAME CLASS</td>
<td>5645</td>
<td>51308</td>
</tr>
<tr>
<td>SAME EXTERNAL SUPERCLASS</td>
<td>4384</td>
<td>66391</td>
</tr>
<tr>
<td>SAME METHOD</td>
<td>569</td>
<td>4901</td>
</tr>
<tr>
<td>SIBLING CLASS</td>
<td>2721</td>
<td>13868</td>
</tr>
<tr>
<td>SUPER CLASS</td>
<td>91</td>
<td>981</td>
</tr>
<tr>
<td>UNRELATED CLASS</td>
<td>2758</td>
<td>35035</td>
</tr>
</tbody>
</table>

Table 5.5: Clone pair location and type distributions of Qualitas Corpus systems

Chart 5.1: Clone pair location distribution of the Qualitas Corpus systems
Type 3 clone pairs are the most recurrent because of an easily understandable reason: when a code fragment is duplicated, it rarely matches all needed functionalities in both positions, therefore the new clone is to be slightly modified (Chart 5.3 and Chart 5.4).

Container kinds, corresponding to clone granularity concept described at section 3.2, are distributed with a high majority of function/method kind amount. This detail agrees with Giesecke’s opinion [9] [10], which asserts function/method to be most suitable granularity level, according to criteria reported at section 3.2. Further considerations about granularity lead to reckon functions and methods to be much more cohesive and significant semantic units than blocks, therefore featuring higher reusability (Table 5.6, Chart 5.5 and Chart 5.6).

<table>
<thead>
<tr>
<th>CLONE PAIR LOCATION</th>
<th>CLONE PAIR CONTAINER KIND</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BLOCK</td>
</tr>
<tr>
<td>ANCESTOR CLASS</td>
<td>120</td>
</tr>
<tr>
<td>COMMON HIERARCHY CLASS</td>
<td>968</td>
</tr>
<tr>
<td>FIRST COUSIN CLASS</td>
<td>473</td>
</tr>
<tr>
<td>SAME CLASS</td>
<td>17096</td>
</tr>
<tr>
<td>SAME EXTERNAL SUPERCLASS</td>
<td>31534</td>
</tr>
<tr>
<td>SAME METHOD</td>
<td>5449</td>
</tr>
</tbody>
</table>
As a result of this analysis, detection procedure of this first DCRA version was configured to detect:

- type 3 clone pairs, since type 1 clones are therefore included;
- clones contained within blocks, since clones contained within methods are therefore included.

As far as refactoring suggestion is concerned, all design and implementation efforts were focused on the most recurring locations with the highest semantic affinity:

- SAME METHOD,
- SAME CLASS,
- SIBLING CLASS,
- SAME EXTERNAL SUPERCLASS.
6 Duplicated Code Refactoring Advisor (DCRA)

As introduced in chapter 1, this work proposes the approach described at chapter 5 and implements it within a specific tool: DCRA. DCRA consists of four components, each designed with a specific purpose. Duplication data flow is enriched after every step, and the whole elaboration process outputs a suitable list of techniques to be applied on the most problematic duplications.

Here are the four components:

- **clone detector**: external tool detecting clone pairs by analyzing source code according to customizable parameters involving aspects described at chapter 3;
- **clone detailer**: examination component analyzing clone detector output and suitably parsing source and binary code in order to compute all necessary clone pair details;
- **refactoring advisor**: evaluation component basically acting as a decision tree visitor, classifying all possible refactoring techniques related to each clone pair depending on all details computed by clone detailer and according to specific criteria; criteria were introduced at chapter 1;
- **refactoring advice aggregator**: summarizing component acting as spreadsheet or database engine, aggregating all clone details output by clone detailer and all advice output by refactoring advisor, grouping them by class or package and sorting them by refactoring significance or clone pair seriousness.

Data flow is summarized by DFD\(^\text{12}\) in Figure 6.1, according to Gane & Sarson’s notation [26] [27]:

---

12 Data Flow Diagram.
Before DCRA execution, some pre-processing operations on the software system are needed, in order to make source code univocally parsable (see section 9.1 for details):

- standardization of end line characters, needed for code line count and line number detection by clone detector;
- resolution of version-dependent incompatibility issues regarding language syntax.

Clone pair toy example in Figure 6.2 will be used throughout all this chapter to show data set evolution at each procedure step (all code lines are here sequentially arranged for convenience, while in real software systems they would be spread over three different source files):

```java
public class SuperClass {}  
public class SubClass1 extends SuperClass {  
   public void method() {  
      int a = 0;  
      int b = 1;  
      a++;  
      b++;  
      System.out.print(a + b);
   }
}

public class SubClass2 extends SuperClass {  
   public void method() {  
      int a = 0;  
      int b = 1;  
      a++;  
      b++;  
      System.out.print(a + b);
   }
}
```

Figure 6.2: DCRA toy example: clone pair

### 6.1 Clone detector

Clone detector selection started from the set of tool reported by Table 4.2. The first and most effective filter involved clone types: only type 3-capable detectors were accepted, excluding therefore Bauhaus Suite, Checkstyle and PMD from the set of candidates. Bauhaus Suite was excluded even if actually type 3-capable [12], because its type 3-capable component could parse only C/C++ source code, while only fully Java-capable detectors were here considered. Remaining type 3-capable detectors needed much deeper consideration of pros and cons, involving even detection report assessment (Table 6.1).

<table>
<thead>
<tr>
<th>CLONE DETECTOR</th>
<th>PROs</th>
<th>CONs</th>
</tr>
</thead>
</table>
| DuDe           | • detailed clone structure  
                • very clear detection algorithm  
                • XML report | • text-based detection approach with therefore following lack of syntax-based configuration parameters, syntax-incoherent detection and code formatting |
Table 6.1: Finally considered clone detector pros and cons

<table>
<thead>
<tr>
<th>Detector</th>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
</table>
| IntelliJ IDEA | - basic but effective syntax-based configuration parameters  
- fast detection  
- user-friendly GUI report | - very conservative detection approach, following poor type 3 clone detection capability  
- secret detection and clone cost evaluation algorithm  
- HTML-site report, poorly script parsing-oriented |
| NiCad | - extremely fine-grained configurability, providing great detection flexibility  
- standardized clone type compliant detection  
- detection of clones corresponding to syntactic unit (blocks and functions)  
- extremely usable report formats, for both automatic processing and human reading | - missing class and package name details in order to precisely identify clones |
| Scorpio | - highly detailed syntax-based customization  
- high level clone detection, based on statement process dependency  
- extremely detailed report, including clone file, method, start and end line and column number | - clones not fully or correctly reported  
- too long gaps among clone statements within the same elaboration flow, following often impossible considerations about refactoring approach |

At last, NiCad [15] was chosen, mainly because it features a granularity selection parameter allowing developers to choose between function/method- and block level clones. This feature follows from its hybrid detection technique and its importance is reported by Zibran and Roy’s considerations described at section 3.2.

Clone detection is then customized according to following parameter values:

- block-level granularity, since more clones are detected, all satisfying Giesecke’s criteria [9]; this granularity level is also justified by Qualitas Corpus [23] system analysis described in section 5.3, which reports 41.75% of all detected clone pairs to be block-level: an absolutely relevant amount;
- 5 lines as minimum clone length, as suggested by Roy and Cordy [3] when describing most commonly used values in literature;
- 30% as dissimilarity maximum percentage between code fragments; Zibran and Roy [2] report no common agreement exists in literature yet, therefore NiCad authors’ default value was kept [15];
- no renaming allowed, since a large set of unmanageable clone pairs would result, as described at section 3.1; only type 1 and the consequent type 3 subset are then detected;
clones grouped in pairs, since much easier to manage singularly, as no design-related consideration is needed; this assertion is supported by Golomi's approach [4] (see section 3.2), Giesecke's diploma thesis [10] and IntelliJ IDEA's output format [7].

Clone detector output is an XML file containing following information about clone pairs:

- file paths,
- start and end line of each clone,
- clone pair length and maximum dissimilarity percentage (unused, since computed much more precisely by clone detailer component, as reported at section 6.2),
- source code fragment of each clone.

XML report resulting from clone detection in previous example is reported in Figure 6.3, although some details unused by DCRA are not reported, in order to increase readability:

```xml
<clones>
  <systeminfo granularity="blocks" threshold="30%" minlines="7" />
  <clone nlines="7" similarity="100">
    <source file="/pathTo/SubClass1.java" startline="34" endline="40">
      public void method() {
        int a = 0;
        int b = 1;
        a++;
        b++;
        System.out.print(a + b);
      }
    </source>
    <source file="/pathTo/SubClass2.java" startline="56" endline="62">
      public void method() {
        int a = 0;
        int b = 1;
        a++;
        b++;
        System.out.print(a + b);
      }
    </source>
  </clone>
</clones>
```

Figure 6.3: DCRA toy example: clone detector XML report

It is worth noting how NiCad computes code lines: lines containing block starting and ending braces are included, too. Therefore minimum clone length parameter must be set to 7 in order to detect all clones longer than 5 actual code lines, as shown in previous XML report.

### 6.2 Clone detailer

This component classifies all clone pairs by detecting all details needed to select the most suitable refactoring techniques, details not contained within clone detector report. Here are all details:

- **class binary names**, together with full package references;
- simplified although unambiguous **signature of methods** containing clones;

25
variables used within each clone, each one labelled according to following criteria: declaration position (inside clone, outside clone but always within its container method, class attribute, inherited attribute) and usage (used after clone but always within its container method, read within clone, modified within clone); Higo et al.’s article [28] was enlightening about this detail;

real length of each clone, with clear differentiation of cloned and different line amount, whereas clone detector only reports total length of the longest clone;

source code of each clone, organized as string array;

cloned pair location, meaning specific combination of clone positions within their class hierarchy, whose possible values are introduced at section 5.1;

clone pair container kind: method or block (block is selected also when kinds are different, since block features more generic manageability);

clone pair type, defined at section 3.1: 1 or 3 (the latter actually being a subset of real type 3 clones, since here no renaming is allowed, as explained at section 6.1);

cloned line amount within each clone;

different line amount within each clone.

Data set in Table 6.2 is provided by clone detailer execution on example in Figure 6.2:

<table>
<thead>
<tr>
<th>CLONE PAIR</th>
<th>SIBLING_CLASS</th>
</tr>
</thead>
<tbody>
<tr>
<td>location</td>
<td>SIBLING_CLASS</td>
</tr>
<tr>
<td>type</td>
<td>ONE</td>
</tr>
<tr>
<td>container kind</td>
<td>METHOD</td>
</tr>
<tr>
<td>cloned line number</td>
<td>10</td>
</tr>
<tr>
<td>different fragment line number</td>
<td>0</td>
</tr>
</tbody>
</table>

FIRST CLONE
- file path /pathTo/SubClass1.java
- start and end line 34, 40
- length 5
- class binary name SubClass1
- method signature public void method()
- container kind METHOD
- inner variable names a, b
- outer variable names
- class attribute names
- inherited attribute names
- used-in-following-code variable names
- read-inside-clone variable names a, b
- written-inside-clone variable names a, b

SECOND CLONE
- file path /pathTo/SubClass2.java
- start and end line 56, 62
- length 5
- class binary name SubClass2
- method signature public void method()
- container kind METHOD
- inner variable names a, b
- outer variable names
- class attribute names
Some implementation details are worth noting:

- this component accepts clone detector report as input and analyzes application source and binary code in order to provide all remaining details; binary code is needed to find all ancestors of each class at runtime, in order to detect its location; all other details are detected by visiting class file AST through suitable APIs designed according to Visitor design pattern [29]: com.sun.source.tree [30] and com.sun.source.util [31];
- several DCRA classes are designed according to Singleton design pattern [29], in order to cache results from previous operations, therefore optimizing following operation execution time;
- the very first operation executed by this component is transforming clone detector output file into a parsable XML temporary file, since clone detector report contains Java source code as is: Java and XML languages share some reserved characters (such as "<" and ">"), therefore unmodified Java source code cannot be parsed by ordinary XML parsers; as a consequence all XML characters provided with a corresponding XML entity [32], within Java source code inside clone detector report, are replaced by their corresponding XML entities; after report parsing all entities are replaced by their original characters;
- clone pair similarity percentage, cloned and different code line computations are achieved by using DiffUtils library [33].

### 6.3 Refactoring advisor

Refactoring advisor is the DCRA component containing most of the founding ideas of this approach: it classifies all possible refactoring techniques for each clone pair by evaluating specific criteria and by analyzing clone detailer output information related to that specific clone pair. These criteria have already been introduced at chapter 1 while this chapter provides all criteria details.

“Mattering element” concept is here introduced, in order to fully understand DCRA evaluation algorithm: variable and/or object reference whose modification within clone block increases clone coupling level with surrounding code, resulting in inclusion or exclusion of specific refactoring techniques from following evaluation. Actual mattering element entity (variable only or variable and object reference) depends on clone pair location (see foot notes at section 9.2 for details).

Refactoring advisor activity may be divided into two distinct subactivities:

<table>
<thead>
<tr>
<th>Inherited attribute names</th>
<th>Used-in-following-code variable names</th>
<th>Read-inside-clone variable names</th>
<th>Written-inside-clone variable names</th>
</tr>
</thead>
<tbody>
<tr>
<td>a, b</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
1. decision tree visit, in order to select all applicable refactoring techniques for each clone pair, wherein each step path selection depends on a specific clone pair detail: location, container kind, type and mattering element amount;
2. evaluation of all selected techniques for each clone pair, achieved by averaging single numeric evaluations of the following listed criteria (possible values range from -1, positive evaluation, and +1, negative evaluation, while 0 means neutrality):
   • line amount variation expressed by percentage, whose computation is detailed at section 9.3,
   • OOP principle compliance: inheritance, polymorphism and encapsulation, each one singularly evaluated and then summarized by its average values; single OOP principle evaluation values are assigned by hand (see section 9.2 for details).

Diagram in Figure 6.4 shows decision tree path visit of the previous toy example (full decision trees are represented by tables at section 9.2): as described before, tree leaves consist of the set of suitable refactoring techniques for that specific clone pair.

![Decision Tree Diagram](image)

**Figure 6.4: DCRA toy example: refactoring advisor decision tree visit**

The second step of refactoring advisor activity, selected refactoring technique evaluation (techniques provided by the previous step), predicts code configuration after refactoring application. “Pull up method”, the first technique contained within the set corresponding to the reached tree leaf, would modify source code as shown in Figure 6.5:

```java
public class SuperClass {
    public void method() {
        int a = 0;
        int b = 1;
        a++;
        b++;
        System.out.print(a + b);
    }
}
```

```java
public class SubClass1 extends SuperClass {}
```
Line amount variation evaluation is achieved by decomposing refactoring techniques into their implementation steps, as described at section 5.2: each implementing step weight (code line amount) is univocally established, as detailed by weight computation formulas in Figure 9.1, while all composing steps of each technique are reported at section 9.2, depending on clone pair details. In Figure 6.5, clone pair weight before refactoring application (corresponding to “Leave application” technique application, too, of course) is 10, while after “Pull up method” application it would be 5. These two weight values are then suitably combined in order to provide criterion evaluation: $\frac{10}{5} = 2$ normalized to $[-1, +1]$ interval by subtracting 1, resulting 1. 1 is the best possible evaluation, following the consideration that a clone pair refactoring may not lead to a code length less than the original clone pair half length. “Leave unchanged” application evaluation follows: $\frac{10}{10} = 1$, by subtracting 1 result is 0.

As described before, OOP principle compliance evaluation is based on single principle evaluation values contained within tables at section 9.2. Example of Figure 6.2 results in: “Pull up method” with encapsulation 0, inheritance 1 and polymorphism 0, while “Leave unchanged” with encapsulation 0, inheritance -1 and polymorphism -1. By averaging all three values, “Pull up method” OOP compliance is 0.33, while “Leave unchanged” compliance is -0.66.

Both criteria for each technique are then averaged again in order to obtain the real evaluation, resulting in a single value for each refactoring technique. An advice object of Table 6.3 is output by refactoring advisor component, with refactoring techniques sorted by evaluation:

<table>
<thead>
<tr>
<th>ADVICE</th>
<th>OOP Compliance</th>
<th>Line number change</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pull up method</td>
<td>0.33</td>
<td>1</td>
<td>0.66</td>
</tr>
<tr>
<td>Leave unchanged</td>
<td>-0.66</td>
<td>0</td>
<td>-0.33</td>
</tr>
</tbody>
</table>

Table 6.3: DCRA toy example: refactoring advisor output

A real software system would output a list with hundreds or thousands advice objects, one for each clone pair. Advice object list order corresponds to the order of clone pair list output by clone detailer.

As described at previous chapters, DCRA is designed aiming at modularity. Several aspects of refactoring advisor component implement modularity:

- the first step of decision tree (location selection) is implemented according to Abstract Factory design pattern [29], by enclosing all following step implementation within location factory classes, since location is the base of all clone pair classification and all following related choices, including applicable refactoring technique selection, too;
refactoring technique names and implementation step names are implemented as simple enumerations.

6.4 Refactoring advice aggregator

This component summarizes all advice objects and clone pair details received from refactoring advisor and clone detailer components. Information is provided to developers by grouping data in sorted sets of refactoring techniques or clone pairs by their weight: techniques are sorted by effectiveness, clone pairs by harmfulness.

Refactoring technique list for each clone pair is restricted to its first element, whose weight is normalized according to its clone pair length, in order to make refactoring application comparison coherent. For instance, if the most suitable technique for a 5 lines-long duplication is evaluated as 1 and the most suitable technique for a 20 lines-long duplication is evaluated as 0.5, application of the second refactoring will improve system quality much more, since respective weight values would be 5 and 10.

Refactoring advice aggregator output for example in Figure 6.2 is then a pair of lists: one containing the most suitable refactoring technique together with its weight (as shown in Table 6.4), the other containing a reference the clone pair (as shown in Table 6.2).

<table>
<thead>
<tr>
<th>Pull up method</th>
<th>Weight</th>
<th>3,33 (valutazione del refactoring * lunghezza del codice duplicato)</th>
</tr>
</thead>
</table>

Table 6.4: DCRA toy example: refactoring advice aggregator output

In the following list all possible output of current DCRA version, output based on simple processing of all refactoring advice and clone pairs:

- refactoring classification by decreasing weight (evaluation * clone pair length):
  - whole refactoring classification;
  - top N most effective refactorings;
- clone pair classification by decreasing weight (clone pair length):
  - whole clone pair classification;
  - top N most harmful clone pairs;
- package classification by decreasing total refactoring application effectiveness:
  - whole package classification;
  - top N most refactorable packages;
- package classification by decreasing total clone pair harmfulness:
  - whole package classification;
  - top N most affected packages;
- class classification by decreasing total refactoring application effectiveness:
  - whole class classification;
  - top N most refactorable classes;
- class classification by decreasing total clone pair harmfulness:
  - whole class classification;
  - top N most affected classes;
Classification by package was considered, since it should help a system design-aware developer to immediately identify the most problematic parts of the system.
7 Validation

DCRA was tested on four software systems belonging to Qualitas Corpus collection [23] and reported in Table 7.1. Before each execution DCRA had to be configured with some parameters depending on each system: source and binary code paths, clone detector, clone detailer and clone advisor output file paths.

Just like the survey on 50 systems belonging to Qualitas Corpus reported at section 5.3, lines of code were computed by script CLOC 1.56 [25], clone pairs were detected by NiCad, DCRA clone detector (see section 6.1 for details), and clone details were detected by clone detailer component by analyzing clone detector output report (see section 6.2).

<table>
<thead>
<tr>
<th>PROJECT</th>
<th>LINES OF CODE</th>
<th>CLONE PAIRS</th>
<th>CLONED LINES OF CODE</th>
<th>CLONED LINES PERCENTAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>fitjava-1.1</td>
<td>3643</td>
<td>27</td>
<td>218</td>
<td>5,98%</td>
</tr>
<tr>
<td>jgrapht-0.8.1</td>
<td>9086</td>
<td>53</td>
<td>736</td>
<td>8,10%</td>
</tr>
<tr>
<td>nekohtml-1.9.14</td>
<td>6149</td>
<td>81</td>
<td>1130</td>
<td>18,38%</td>
</tr>
<tr>
<td>oscache-2.4.1</td>
<td>5786</td>
<td>70</td>
<td>982</td>
<td>16,97%</td>
</tr>
</tbody>
</table>

Table 7.1: Qualitas Corpus systems used in DCRA test

Clone detailer also detected clone pair characteristics, distributed as reported by Chart 7.1, Chart 7.2 and Chart 7.3, where clone pairs are arranged by location, type and container kind:

Chart 7.1: DCRA test system clone pair location distribution percentage

Chart 7.2: DCRA test system clone pair container kind distribution percentage

Chart 7.3: DCRA test system clone pair type distribution percentage
Clone detailer output was then analyzed by refactoring advisor component, as described at section 6.3. The subset of possible locations fully managed by this component provided refactoring advice for more than 80% of clone pairs (81.68%). All refactoring suggestions were then assessed by hand, in order to verify actual applicability and suitability.

Previous classification by location, type and container kind is here applied in order to clearly report refactoring advice details. Bar charts in Chart 7.4, Chart 7.5, Chart 7.6 and Chart 7.7 are used instead of pie chart, in order to suitably represent actual applicability assessment result, meaning the satisfaction of all conditions needed by each refactoring and its actual application.

Chart 7.4: DCRA test system refactoring applicability by clone pair location

Chart 7.5: DCRA test system refactoring applicability by clone pair type

Chart 7.6: DCRA test system refactoring applicability by clone pair container kind
Detailed results of the manual applicability assessment, represented by ordered pairs of refactoring application/missed application (APPLIED / NOT APPLIED), are reported in Table 7.2 and Table 7.3. Results are divided by related location pairs (SAME METHOD and SAME CLASS, SIBLING CLASS and SAME EXTERNAL SUPERCLASS), since different location pairs means different possible refactoring sets. Classification by container kind is omitted, since it is proportionally distributed, in order to increase readability.

<table>
<thead>
<tr>
<th>LOCATION CONTAINER KIND</th>
<th>EXTRACT METHOD</th>
<th>REPLACE METHOD WITH METHOD OBJECT</th>
<th>MERGE METHOD</th>
<th>LEAVE UNCHANGED</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAME METHOD ONE</td>
<td>2 / 0</td>
<td>1 / 0</td>
<td>0 / 0</td>
<td>0 / 0</td>
</tr>
<tr>
<td>SAME METHOD THREE</td>
<td>8 / 2</td>
<td>0 / 0</td>
<td>0 / 0</td>
<td>33 / 0</td>
</tr>
<tr>
<td>SAME CLASS ONE</td>
<td>6 / 0</td>
<td>0 / 0</td>
<td>0 / 0</td>
<td>0 / 0</td>
</tr>
<tr>
<td>SAME CLASS THREE</td>
<td>81 / 9</td>
<td>0 / 0</td>
<td>0 / 0</td>
<td>10 / 0</td>
</tr>
</tbody>
</table>

Table 7.2: DCRA test system applicability for clone pairs within same class

<table>
<thead>
<tr>
<th>LOCATION CONTAINER KIND</th>
<th>PULL UP METHOD</th>
<th>PULL UP METHOD OBJECT</th>
<th>FORM TEMPLATE METHOD</th>
<th>LEAVE UNCHANGED</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIBLING CLASS ONE</td>
<td>2 / 0</td>
<td>0 / 0</td>
<td>0 / 0</td>
<td>0 / 0</td>
</tr>
<tr>
<td>SIBLING CLASS THREE</td>
<td>0 / 2</td>
<td>0 / 0</td>
<td>1 / 1</td>
<td>0 / 0</td>
</tr>
<tr>
<td>SAME EXTERNAL SUPERCLASS ONE</td>
<td>8 / 1</td>
<td>0 / 0</td>
<td>0 / 0</td>
<td>0 / 0</td>
</tr>
<tr>
<td>SAME EXTERNAL SUPERCLASS THREE</td>
<td>0 / 0</td>
<td>0 / 3</td>
<td>8 / 17</td>
<td>0 / 0</td>
</tr>
</tbody>
</table>

Table 7.3: DCRA test system applicability for clone pairs between sibling classes

In Figure 7.1 and Figure 7.2 an example of clone pair refactoring-worthy, whose suggested technique was EXTRACT METHOD. In this specific situation, no flag parameter was needed, even if this clone pair is type 3, since all domain values of v variable are correctly managed by Java Virtual Machine, without the need any conditional branch. The final configuration essentially consists of a private method aliased by two public methods.
public class CycleDetector<V, E> {

   [...]

   public boolean detectCycles() {
      try {
         execute(null, null);
      } catch (CycleDetectedException ex) {
         return true;
      }
      return false;
   }

   public boolean detectCyclesContainingVertex(V v) {
      try {
         execute(null, v);
      } catch (CycleDetectedException ex) {
         return true;
      }
      return false;
   }

   [...]
}

Figure 7.1: Example of refactoring-worthy clone pair - BEFORE REFACTORING

public class CycleDetector<V, E> {

   [...]

   private boolean extractedMethod(V v) {
      try {
         execute(null, v);
      } catch (CycleDetectedException ex) {
         return true;
      }
      return false;
   }

   public boolean detectCycles() {
      return extractedMethod(null);
   }

   public boolean detectCyclesContainingVertex(V v) {
      return extractedMethod(v);
   }

   [...]
}

Figure 7.2: Example of refactoring-worthy clone pair - AFTER REFACTORING

Scenario-related considerations about clone pairs:

- **type 3 between sibling classes (both SIBLING CLASS and SAME EXTERNAL SUPERCLASS):** 72% of refactoring techniques suggested as the most suitable were actually not worth of application, above all FORM TEMPLATE METHOD, because of the actual cloned line small amount; PULL UP METHOD was usually preferred to FORM TEMPLATE METHOD (Figure 7.3 and Figure 7.4); in order to prevent this issue, a
higher lower bound to clone line amount is needed: under this bound, a clone pair should be automatically assigned LEAVE UNCHANGED as the best possible refactoring, especially if the clone pair is subset of no larger clone class;

- **type 3 within the same class (both SAME METHOD and SAME CLASS)**: in refactoring classification from the most effective to the least effective, the last part of the list reports mainly application-unworthy refactorings, because of the actual cloned line small amount (see Figure 7.5 and Figure 7.6); as in previous consideration, a higher lower bound to clone line amount is needed for this scenario;

- **type 1**: infrequent, but refactoring suggestions are mostly applicable;

- **more than one mattering element**: infrequent, therefore REPLACE METHOD WITH METHOD OBJECT and PULL UP METHOD OBJECT techniques, very line-expensive, were suggested just a few times.

In Figure 7.3 and Figure 7.4 an example of duplication where FORM TEMPLATE METHOD technique was suggested, but actually PULL UP METHOD was applied: these two small clones differed only by the continue statement, therefore a type 1 clone pair results by excluding that single statement, easily refactorable with PULL UP METHOD. In this specific situation, PULL UP METHOD application was favoured by enclosing class attributes and methods seen as "global" within the two classes containing the two clones.

```java
public class HTMLScanner implements XMLDocumentScanner, XMLLocator, HTMLComponent {
    protected CurrentEntity fCurrentEntity;
    protected int skipNewlines() throws IOException {
        return 0;
    }
    public class ContentScanner implements Scanner {
        protected boolean scanMarkupContent(XMLStringBuffer buffer, char cend) throws IOException {
            if (c == 'n' || c == 'r') {
                fCurrentEntity.rewind();
                int newlines = skipNewlines();
                for (int i = 0; i < newlines; i++) {
                    buffer.append('n');
                }
                continue;
            }
        }
    }
    public class SpecialScanner implements Scanner {
        protected void scanCharacters(XMLStringBuffer buffer, int delimiter) throws IOException {
            if (c == 'r' || c == 'n') {
                fCurrentEntity.rewind();
            }
        }
    }
}
```
int newlines = skipNewlines();
for (int i = 0; i < newlines; i++) {
    buffer.append('
');
}

Figure 7.3: Example of FORM TEMPLATE METHOD suggestion substituted by PULL UP METHOD - BEFORE REFACTORING

public class HTMLScanner implements XMLDocumentScanner, XMLLocator, HTMLComponent {
    protected CurrentEntity fCurrentEntity;
    protected int skipNewlines() throws IOException {
        return 0;
    }
    private class CreatedSuperClass {
        protected void pulledUpMethod(XMLStringBuffer buffer) {
            fCurrentEntity.rewind();
            int newlines = skipNewlines();
            for (int i = 0; i < newlines; i++) {
                buffer.append('
');
            }
        }
    }
    public class ContentScanner extends CreatedSuperClass implements Scanner {
        protected boolean scanMarkupContent(XMLStringBuffer buffer, char cend) throws IOException {
            if (c == '\n' || c == '') {
                super.pulledUpMethod(buffer);
                continue;
            }
        }
    }
    public class SpecialScanner extends CreatedSuperClass implements Scanner {
        protected void scanCharacters(XMLStringBuffer buffer, int delimiter) throws IOException {
            if (c == '\r' || c == '\n') {
                super.pulledUpMethod(buffer);
            }
        }
    }
}

Figure 7.4: Example of FORM TEMPLATE METHOD suggestion substituted by PULL UP METHOD - AFTER REFACTORING

In Figure 7.5 and Figure 7.6 an example of clone pair whose refactoring suggestion, EXTRACT METHOD implemented with flag parameter, was not application-worthy, since resulting
code would be longer than the original one and mostly contained within conditional branches.

```java
public class DefaultListenableGraph<V, E>
    extends GraphDelegator<V, E>
    implements ListenableGraph<V, E>, Cloneable {
    ...

    public E addEdge(V sourceVertex, V targetVertex) {
        E e = super.addEdge(sourceVertex, targetVertex);
        if (e != null) {
            fireEdgeAdded(e);
        }
        return e;
    }
    ...

    public E removeEdge(V sourceVertex, V targetVertex) {
        E e = super.removeEdge(sourceVertex, targetVertex);
        if (e != null) {
            fireEdgeRemoved(e);
        }
        return e;
    }
    ...

    private E extractedMethod(V sourceVertex, V targetVertex, int flag) {
        E e = null;
        if (flag == 0) {
            e = super.addEdge(sourceVertex, targetVertex);
        }
        if (flag == 1) {
            e = super.removeEdge(sourceVertex, targetVertex);
        }
        if (e != null) {
            if (flag == 0) {
                fireEdgeAdded(e);
            }
            if (flag == 1) {
                fireEdgeRemoved(e);
            }
        }
        return e;
    }

    public E addEdge(V sourceVertex, V targetVertex) {
        return extractedMethod(sourceVertex, targetVertex, 0);
    }
    ...
```

Figure 7.5: Example of refactoring-unworthy clone pair - BEFORE REFACTORING
public E removeEdge(V sourceVertex, V targetVertex) {
    return extractedMethod(sourceVertex, targetVertex, 1);
}

[...]

Figure 7.6: Example of refactoring-unworthy clone pair - AFTER REFACTORING (NOT APPLIED)

Application-related considerations about refactoring techniques:

- **PULL UP METHOD OBJECT**: suggested but never applied, since mattering elements mainly consisted of class attributes, therefore PULL UP METHOD was applied together with attribute repositioning into the superclass; the choice of pulling up attributes is hardly representable by a decisional algorithm, since it involves both practical and design issues: practical issues are, for instance, testing whether an attribute is used within cloned and/or different code fragments or whether an attribute belongs to both sibling classes, while design issues are, for instance, testing whether the system responsibility policy allows attribute generalization towards a superclass;

- **continue, return and break**: clones containing these statements are usually avoided, since their management is very expensive [34]; nevertheless, if one of them coincides with the last statement of a clone, clone management may simply exclude that statement from considered code lines (an example in Figure 7.4).

Refactoring technique application highlighted a problem only partially considered during first design steps: clone grouping by pairs and not by classes (see section 3.3) caused an unoptimized management of correlated clones. This issue involves especially groups of clones consisting of both clones belonging to the same class and clones belonging to sibling classes: by not considering all group clone characteristics all together, incompatible strategies might be applied simultaneously to clone subgroups, forcing a final adapting merging procedure of resulting entities very difficult to execute. For instance: if a clone subgroup is refactored with PULL UP METHOD technique and the other subgroup with FORM TEMPLATE METHOD, all components of involved entities will be at last spread within different classes, since PULL UP METHOD concentrates everything within the superclass, while FORM TEMPLATE METHOD delegates all different procedure steps to original classes [1]. As a consequence, final configuration will be a set of interdependent classes without a shared design logic.
Conclusions and future works

Duplicated code is a widespread code smell [1], meaning a symptom of bad developing practices or potential design issues.

Even if several sources assert leaving affected code unchanged or delegating full choice to developers are the only ways to deal with it [2] [3], this work suggests a corrective approach, since suitable duplicated code removal improves software system design quality, resulting in increased manageability.

Proposed approach may be summarized as an automated correction technique evaluation and selection based on duplication classification, resulting in a concrete reduction of human involvement during duplicated code refactoring procedures. This approach is mainly inspired by Golomini’s diploma thesis [4], whose scenario-based approach is based on clone pair classification by clone location (clone positions within class hierarchy); a set of possible refactoring techniques is suggested for each clone pair category, leaving all selection details to developers.

Proposed approach extends Golomini’s classification categories, filters duplications depending on their refactoring-worthiness and automatically evaluates technique suitability for each duplication.

Classification category extension proved to be very suitable, since it allowed to automatically suggest suitable refactoring techniques for a large amount of duplications considered hardly manageable by Golomini’s approach.

All applied filters proved to be mostly effective, since mainly refactoring-worthy duplications were proposed to developer. Only 8% of duplications were actually refactoring-unworthy.

Refactoring technique advice proved to be effective when related to clone pairs within the same class (locations SAME METHOD and SAME CLASS), with 92% of applicable suggestions. This led to suitably refactor 66% of all duplications of analyzed software systems.

Clone pairs within sibling classes (locations SIBLING CLASS and SAME EXTERNAL CLASS) were much more problematic because applicability evaluation was not suitably fine-tuned: FORM TEMPLATE METHOD technique, for instance, is usually evaluated to be the best solution because of its high OOP principle compliance, even if its code line-based cost is very high. Therefore 40% of all FORM TEMPLATE METHOD suggestions resulted to be refactoring-unworthy and some other instances were refactored with PULL UP METHOD technique.

Some refactoring application practical aspects were not considered during approach design, but they came to light during testing step:

- during EXTRACT METHOD application, main difficulties involve argument and return value type and number, usually solved by introducing suitable casting operations and optional parameters (meaning they’re not used within all extracted method execution paths);
- during refactoring of clone pairs between sibling classes, refactoring procedure was highly simplified by pulling up shared attributes.

Future DCRA versions will include and consider:

- refactoring technique suggestions for all unmanaged locations;
• how clone detector configuration parameters condition detected duplication set
  manageability, especially minimum clone length (now 5 lines) and minimum allowed
dissimilarity percentage (now 30%);
• evaluation weight balancing for each refactoring technique for clone pairs within
  sibling classes;
• suitability assessment about the same weight assigned to the two refactoring
evaluation criteria;
• more specific evaluation criteria about LEAVE UNCHANGED refactoring technique;
• a further detail within clone detailer component: whether a mattering element is
  read or modified within cloned or different fragments of code clone, in order to get
  more suitably evaluated clone pairs and related refactoring techniques.
9 Appendices

9.1 Analyzed system pre-processing

In order to get maximal information accuracy, the whole DCRA procedure should be preceded by some standardization operations on analyzed system source code, as described by following list:

- end line character standardization, since not all end line characters are properly managed by all systems; before Qualitas Corpus system analysis (see section 5.3), this operation was necessary because the script CLOC 1.56 [25], used here to count actual source code lines, could not recognize character “\r” as an end line character: standardization was achieved by executing shell commands dos2unix and mac2unix [35];
- manual adaptations of code fragments incompatible with the current Java Virtual Machine (version 7) because of obsolete characteristics:
  - “enum” keyword used as variable name (cause of most incompatibilities),
  - generic parameters of some methods declared without generic types, resulting in unimplemented method errors (especially for subclasses of abstract classes),
  - ambiguous method calls,
  - unimplemented methods within classes implementing specific interfaces,
  - unresolvable dependencies,
  - method calls with a wrong parameter number.

Code formatting standardization through eclipse command “Format...” is useless, since NiCad standardizes code internally [15], therefore two clone detections, before and after “Format...” command execution, provide equal sets of clone pairs.

Deeper code standardization through eclipse command “Clean up...” was not executed, since many problems occurred: error introduction (especially within loops) and unresolved dependency involving package org.apache.commons.lang.enum (which is hidden by current Java Virtual Machine even if present, since it is deprecated), for instance.

9.2 Refactoring advisor decision tree tables

All the following tables represent the refactoring advisor decision tree. See section 6.3 to understand how they are used.
### Location SAME METHOD:

<table>
<thead>
<tr>
<th>CONTAINER KIND</th>
<th>TYPE</th>
<th>NUMBER OF MATTERING ELEMENTS&lt;sup&gt;15&lt;/sup&gt;</th>
<th>REFACTORING TECHNIQUES</th>
<th>IMPLEMENTATION STEPS</th>
<th>OOP COMPLIANCE EVALUATION&lt;sup&gt;16&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block</td>
<td>1</td>
<td>0</td>
<td>Extract method</td>
<td>Extract method</td>
<td>Encapsulation: 0  Inheritance: 0  Polymorphism: 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Leave unchanged</td>
<td>- none -</td>
<td>Encapsulation: 0  Inheritance: 0  Polymorphism: 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Extract method</td>
<td>Extract method + return</td>
<td>Encapsulation: 0  Inheritance: 0  Polymorphism: 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Leave unchanged</td>
<td>- none -</td>
<td>Encapsulation: 0  Inheritance: 0  Polymorphism: 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Replace method with method object</td>
<td>Replace method with method object</td>
<td>Encapsulation: 0  Inheritance: 0  Polymorphism: 0</td>
</tr>
<tr>
<td>Block</td>
<td>1</td>
<td>&gt;1</td>
<td>Extract method</td>
<td>Extract method + flag</td>
<td>Encapsulation: 0  Inheritance: 0  Polymorphism: 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Leave unchanged</td>
<td>- none -</td>
<td>Encapsulation: 0  Inheritance: 0  Polymorphism: 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Extract method</td>
<td>Extract method + return + flag</td>
<td>Encapsulation: 0  Inheritance: 0  Polymorphism: 0</td>
</tr>
<tr>
<td>Block</td>
<td>3</td>
<td>0</td>
<td>Replace method with method object</td>
<td>Replace method with method object + flag</td>
<td>Encapsulation: 0  Inheritance: 0  Polymorphism: 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Leave unchanged</td>
<td>- none -</td>
<td>Encapsulation: 0  Inheritance: 0  Polymorphism: 0</td>
</tr>
<tr>
<td>Block</td>
<td>3</td>
<td>&gt;1</td>
<td>Replace method with method object</td>
<td>Replace method with method object + flag</td>
<td>Encapsulation: 0  Inheritance: 0  Polymorphism: 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Leave unchanged</td>
<td>- none -</td>
<td>Encapsulation: 0  Inheritance: 0  Polymorphism: 0</td>
</tr>
<tr>
<td>Method</td>
<td>any</td>
<td>any</td>
<td>IMPOSSIBLE SCENARIO</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<sup>15</sup> Number of outer variables (declared before clone), written within and used after clone.

<sup>16</sup> All OOP evaluations equal 0 in this location, because:
- If clones are detected within the same method and these clones are then refactored, class outer interface does not change, so encapsulation level does not change;
- If clones are detected within the same method, there is no class hierarchy between the two clone container objects, therefore no change to implement inheritance and polymorphism.
## Location SAME CLASS:

<table>
<thead>
<tr>
<th>CONTAINER KIND</th>
<th>TYPE</th>
<th>NUMBER OF MATTERING ELEMENTS</th>
<th>REFACTORIZING TECHNIQUES</th>
<th>IMPLEMENTATION STEPS</th>
<th>OOP COMPLIANCE EVALUATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block</td>
<td>1</td>
<td>0</td>
<td>Extract method</td>
<td>Extract method</td>
<td>Encapsulation: 0 Inheritance: 0 Polymorphism: 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Leave unchanged</td>
<td>- none -</td>
<td>Encapsulation: 0 Inheritance: 0 Polymorphism: 0</td>
</tr>
<tr>
<td>Block</td>
<td>1</td>
<td>1</td>
<td>Extract method + return</td>
<td></td>
<td>Encapsulation: 0 Inheritance: 0 Polymorphism: 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Leave unchanged</td>
<td>- none -</td>
<td>Encapsulation: 0 Inheritance: 0 Polymorphism: 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Replace method with method object</td>
<td>Replace method with method object</td>
<td>Encapsulation: 0 Inheritance: 0 Polymorphism: 0</td>
</tr>
<tr>
<td>Block</td>
<td>1</td>
<td>&gt;1</td>
<td>Extract method + flag</td>
<td></td>
<td>Encapsulation: 0 Inheritance: 0 Polymorphism: 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Leave unchanged</td>
<td>- none -</td>
<td>Encapsulation: 0 Inheritance: 0 Polymorphism: 0</td>
</tr>
<tr>
<td>Block</td>
<td>3</td>
<td>0</td>
<td>Extract method + return + flag</td>
<td></td>
<td>Encapsulation: 0 Inheritance: 0 Polymorphism: 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Leave unchanged</td>
<td>- none -</td>
<td>Encapsulation: 0 Inheritance: 0 Polymorphism: 0</td>
</tr>
<tr>
<td>Block</td>
<td>3</td>
<td>&gt;1</td>
<td>Replace method with method object</td>
<td>Replace method with method object + flag</td>
<td>Encapsulation: 0 Inheritance: 0 Polymorphism: 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Leave unchanged</td>
<td>- none -</td>
<td>Encapsulation: 0 Inheritance: 0 Polymorphism: 0</td>
</tr>
<tr>
<td>Method</td>
<td>1</td>
<td>0</td>
<td>Merge method</td>
<td>Merge method</td>
<td>Encapsulation: 0 Inheritance: 0 Polymorphism: 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Leave unchanged</td>
<td>- none -</td>
<td>Encapsulation: 0 Inheritance: 0 Polymorphism: 0</td>
</tr>
</tbody>
</table>

---

17 Number of outer variables (declared before clone), written within and used after clone.

18 All OOP evaluations equal 0 in this location, because:

- If clones are detected within the same class and these clones are then refactored, class outer interface does not change, so encapsulation level does not change;
- If clones are detected within the same class, there is no class hierarchy between the two clone container objects, therefore no change to implement inheritance and polymorphism.
<table>
<thead>
<tr>
<th>Method</th>
<th>3</th>
<th>0</th>
<th>Extract method</th>
<th>Extract method + flag</th>
<th>Encapsulation: 0</th>
<th>Inheritance: 0</th>
<th>Polymorphism: 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method</td>
<td>any</td>
<td>&gt;0</td>
<td>Leave unchanged</td>
<td>- none -</td>
<td>Encapsulation: 0</td>
<td>Inheritance: 0</td>
<td>Polymorphism: 0</td>
</tr>
</tbody>
</table>

Table 9.2: SAME CLASS location decision tree

### Location SIBLING CLASS:

<table>
<thead>
<tr>
<th>CONTAINER KIND</th>
<th>TYPE</th>
<th>NUMBER OF MATTERING ELEMENTS</th>
<th>REFACTORING TECHNIQUES</th>
<th>IMPLEMENTATION STEPS</th>
<th>OOP COMPLIANCE EVALUATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block</td>
<td>1</td>
<td>0</td>
<td>Pull up method</td>
<td>Extract method + Pull up method</td>
<td>Encapsulation: 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Leave unchanged</td>
<td>- none -</td>
<td>Encapsulation: 0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>Pull up method</td>
<td>Extract method + return + Pull up method</td>
<td>Encapsulation: 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Leave unchanged</td>
<td>- none -</td>
<td>Encapsulation: 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&gt;1</td>
<td>Pull up method object</td>
<td>Replace method with method object + Pull up method object</td>
<td>Encapsulation: 0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>&gt;1</td>
<td>Leave unchanged</td>
<td>- none -</td>
<td>Encapsulation: 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Form template method</td>
<td>Extract method + Form template method</td>
<td>Encapsulation: 0</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0</td>
<td>Pull up method</td>
<td>Extract method + flag + Pull up method</td>
<td>Encapsulation: 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Leave unchanged</td>
<td>- none -</td>
<td>Encapsulation: 0</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1</td>
<td>Form template method</td>
<td>Extract method + return + Form template method</td>
<td>Encapsulation: 0</td>
</tr>
</tbody>
</table>

19 Number of outer variables (declared before clone), written within and used after clone and number of class attributes written within clone.
<table>
<thead>
<tr>
<th>Block</th>
<th>3</th>
<th>&gt;1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Pull up method</td>
<td>Extract method + flag + return + Pull up method</td>
<td>Encapsulation: 0 Inheritance: 1 Polymorphism: 0</td>
</tr>
<tr>
<td>Leave unchanged</td>
<td>- none -</td>
<td>Encapsulation: 0 Inheritance: -1 Polymorphism: -1</td>
</tr>
<tr>
<td>Method</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Pull up method object</td>
<td>Replace method with method object + flag + Pull up method object</td>
<td>Encapsulation: 0 Inheritance: 1 Polymorphism: 0</td>
</tr>
<tr>
<td>Leave unchanged</td>
<td>- none -</td>
<td>Encapsulation: 0 Inheritance: -1 Polymorphism: -1</td>
</tr>
<tr>
<td>Method</td>
<td>1</td>
<td>&gt;1</td>
</tr>
<tr>
<td>Pull up method object</td>
<td>Replace method with method object + Pull up method object</td>
<td>Encapsulation: 0 Inheritance: -1 Polymorphism: 0</td>
</tr>
<tr>
<td>Leave unchanged</td>
<td>- none -</td>
<td>Encapsulation: 0 Inheritance: -1 Polymorphism: -1</td>
</tr>
<tr>
<td>Method</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Pull up method</td>
<td>Extract method + flag + Pull up method</td>
<td>Encapsulation: 0 Inheritance: 1 Polymorphism: 0</td>
</tr>
<tr>
<td>Leave unchanged</td>
<td>- none -</td>
<td>Encapsulation: 0 Inheritance: -1 Polymorphism: -1</td>
</tr>
<tr>
<td>Method</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Pull up method</td>
<td>Extract method + flag + return + Pull up method</td>
<td>Encapsulation: 0 Inheritance: 1 Polymorphism: 0</td>
</tr>
<tr>
<td>Leave unchanged</td>
<td>- none -</td>
<td>Encapsulation: 0 Inheritance: -1 Polymorphism: -1</td>
</tr>
<tr>
<td>Method</td>
<td>3</td>
<td>&gt;1</td>
</tr>
<tr>
<td>Pull up method object</td>
<td>Replace method with method object + flag + Pull up method</td>
<td>Encapsulation: 0 Inheritance: 1 Polymorphism: 0</td>
</tr>
<tr>
<td>CONTAINER KIND</td>
<td>TYPE</td>
<td>NUMBER OF MATTERING ELEMENTS</td>
</tr>
<tr>
<td>----------------</td>
<td>------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>Block</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Block</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Block</td>
<td>1</td>
<td>&gt;1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Block</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Block</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 9.3: SIBLING CLASS location decision tree

Location SAME EXTERNAL CLASS:

---

20 Number of outer variables (declared before clone), written within and used after clone and number of class attributes written within clone.
<table>
<thead>
<tr>
<th>Block</th>
<th>Method</th>
<th>Encapsulation</th>
<th>Inheritance</th>
<th>Polymorphism</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
**Table 9.4: SAME EXTERNAL SUPERCLASS location decision tree**

<table>
<thead>
<tr>
<th>Method</th>
<th>3</th>
<th>&gt;1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pull up method object</td>
<td>Create superclass + Replace method object + flag + Pull up method object</td>
<td>Encapsulation: 0 Inheritance: 1 Polymorphism: 0</td>
</tr>
<tr>
<td>Leave unchanged</td>
<td>- none -</td>
<td>Encapsulation: 0 Inheritance: -1 Polymorphism: -1</td>
</tr>
</tbody>
</table>

**9.3 Line change computation formulas**

As said at section 5.2, refactoring implementation steps are used to compute post-refactoring line change. Each step has its own formula, which combines one or more clone pair detail values shown in the following list:

- number of mattering elements,
- total number of cloned lines (of both clones),
- total number of different lines (of both clones),
- number of different fragments.

All implementation step formulas in Table 9.6:

<table>
<thead>
<tr>
<th>IMPLEMENTATION STEP</th>
<th>LINE-BASED COST FORMULA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extract method</td>
<td>1 * 2 + totalClonedLineNumber / 2 + 1 // extracted method invocations</td>
</tr>
<tr>
<td>return</td>
<td>1 // &quot;return&quot; statement</td>
</tr>
<tr>
<td>Replace method with method object</td>
<td>1 * 2 + totalClonedLineNumber / 2 + 1 + 1 + matteringElementNumber + matteringElementNumber // method object instantiations</td>
</tr>
<tr>
<td>flag</td>
<td>totalDifferentLineNumber + 1 * differentFragmentNumber // uncommon line number (total)</td>
</tr>
<tr>
<td>Merge method</td>
<td>totalClonedLineNumber / 2 // common line number (not total)</td>
</tr>
<tr>
<td>Pull up method</td>
<td>0</td>
</tr>
<tr>
<td>Pull up method object</td>
<td>0</td>
</tr>
</tbody>
</table>
All these formulas are used within each clone refactoring advice factory, combined within longer formulas which generalize all possible situations. In Figure 9.1 the generic formula for SIBLING CLASS location (long uppercase names with underscore characters are enumeration values, here used just to represent the situation where that specific implementation step is contained within the set of the refactoring technique whose line number change is being evaluated).

```latex
// common line number (not total)
totalClonedLineNumber / 2 +
// extracted method invocations / method object instantiations
(EXTRACT_METHOD ||
  REPLACE_METHOD_WITH_METHOD_OBJECT ||
  FORM_TEMPLATE_METHOD ? 1 * 2 : 0) +
// extracted method signature / method object constructor signature
(EXTRACT_METHOD ||
  REPLACE_METHOD_WITH_METHOD_OBJECT ||
  FORM_TEMPLATE_METHOD ? 1 : 0) +
// return statement
(RETURN ? 1 : 0) +
// method object class first line
(REPLACE_METHOD_WITH_METHOD_OBJECT ? 1 : 0) +
// method object class attribute declarations
(REPLACE_METHOD_WITH_METHOD_OBJECT ? matteringElementNumber : 0) +
// method object class attribute assignments
(REPLACE_METHOD_WITH_METHOD_OBJECT ? matteringElementNumber : 0) +
// variable value fetch from method object attributes
(REPLACE_METHOD_WITH_METHOD_OBJECT ? matteringElementNumber : 0) +
// uncommon line number (total) / uncommon line number (total) within concrete submethods
(FLAG || FORM_TEMPLATE_METHOD ? totalDifferentLineNumber : 0) +
// "if" block first lines / abstract submethod signatures
(FLAG || FORM_TEMPLATE_METHOD ? 1 * differentFragmentNumber : 0) +
// submethod invocations within extracted method
(FORM_TEMPLATE_METHOD ? 1 * differentFragmentNumber : 0) +
```

Table 9.6: Implementation step line-based cost formulas
// concrete submethod signatures
(FORM_TEMPLATE_METHOD ? 1 * differentFragmentNumber * 2 : 0);

Figure 9.1: SIBLING CLASS implementation cost whole formula
References


Acknowledgements

I would like to thank all the people who helped me during this work and during these years. I cannot remember all their names and all their faces, since they’re really a lot of people. Therefore, before beginning the list of acknowledgements, I apologize with all unmentioned people who deserve my gratitude.

I thank:

- Barbara, my life’s companion, for supporting me every single day, for her endless patience with my endless mood swings, for discovering and valuing my inner strength, for making every single day a day to remember;
- Erminio and Margherita, my parents, who taught me thinking; I can remember when they allowed me to disassemble and reassemble our only (at that time) computer, in hopes that would broaden my horizons: I think it was worth it;
- Gina, my grandmother, from which I inherited all the necessary stubbornness to succeed in finishing this hard but wonderful journey;
- Cichin, my grandfather, from which I inherited a bit crafty and a strong love for preciseness;
- Angela, Barbara’s mother, who provided an extraordinarily efficient logistical support apparatus, while waiting for me to teach her surfing the web;
- Prof. Francesca Arcelli Fontana, my supervisor, for her endless availability;
- Dr. Marco Zanoni, my assistant supervisor, whose patience have already made him earn a place in heaven: I’m sure he’ll never forget all the hundreds of “Olà!”s at the beginning of each Skype conversation;
- Dr. James R. Cordy, from Queen’s University at Kingston, Ontario, Canada, for all the precious hints about NiCad use and customization;
- Dr. Simon Giesecke, from Carl von Ossietzky University of Oldenburg, Germany, who provided me his diploma thesis, on which part of this work is based, as soon as I asked him;
- all tutors at Nursing School in Novara, who accepted my not-so-suitable presence during all these months, by supporting me with their contagious hilarity;
- the whole Università degli Studi di Milano-Bicocca, for the wonderful educational experience.