

SUPREMO

A Scenario Based Approach for Refactoring Duplicated Code in Object Oriented Systems

Diploma Thesis
of the Faculty of Sciences
University of Bern

by

Georges Golomingi Koni-N'Sapu

2001

Supervisors:
Prof. Dr. Oscar Nierstrasz
Dr. Stéphane Ducasse
Matthias Rieger

Institute of Computer Science

Abstract

Code duplication is one of the factors that severely complicates the maintenance and evolution of large software systems. Tools exist that allow detection of duplicated code. Technics to change, correct and improve code exist also. But it is difficult to find programs that work between both domains.

In this work, we discuss a scenario based approach to analyze, categorize and remove duplicated code in an object oriented context. The scenario is defined as the relationship between classes containing methods where the duplications were found.

A prototype framework, SUPREMO, has been developed to validate our approach. It is characterized by the following aspect:

- Visualization of the scenario in a graphical global context that gives the developer the possibility to see the impact of the duplication.
- Visualization of the source code in a textual viewer where a pop-up menu gives the user the opportunity to refactor.

Nine case studies (seven written in Smalltalk, one in C++ and one in Java) are analyzed. A presentation of statistical results and a discussion about the qualitative aspect of three applications developed in the SCG group are presented. The qualitative validation is illustrated with a list of examples that simulate the functioning of SUPREMO.

Acknowledgements

I would like to thank people who helped me during this work:

- Antho, Davina and Murielle, my little family, for enduring my absences, my nervousity, and for encouraging me to continue.
- Matthias Rieger, Stéphane Ducasse and Oscar Nierstrasz, my supervisors, for their support.
- Thomas Hofmann and Michele Lanza for reading and pushing me during the last sprint.

Georges Golomingi Koni-N'Sapu
June, 2001

Contents

1	Introduction	1
1.1	Our Contribution	1
1.2	Structure of the Thesis	2
2	Duplicated Code and Refactoring	3
2.1	The Problem of Duplicated Code	3
2.1.1	Software Reuse and Origin of Clones	3
2.1.2	The Problem	4
2.1.3	Detection of Duplicated Code	5
2.2	Refactoring	5
2.2.1	A Survey of Refactoring	5
2.2.2	How Refactoring Can Remedy Code Duplication	6
2.2.3	Techniques of Refactoring	6
3	Classifying Duplication in Refactorable Scenarios	8
3.1	Definition and Properties of Duplication	9
3.2	Constraints	9
3.3	Duplication in the Same Method	10
3.4	Duplication in the Same Class	13
3.5	Duplication between Sibling Classes	18
3.6	Duplication with Superclass	21
3.7	Duplication with Ancestor	22
3.8	Duplication with First Cousin	23
3.9	Duplication in Common Hierarchy	24
3.10	Duplication in Unrelated Classes	25
3.11	Summary	26
4	SUPREMO: Tool Support for Duplication Elimination	27
4.1	Introduction	27
4.2	Requirements and functionality	27
4.3	Required Applications	29
4.3.1	HotDraw	29
4.3.2	Duploc	29
4.3.3	FAMIX	29
4.4	Tool Architecture	30
4.4.1	Implementation	31

	Creation and Management of the Object DuplicationUnit . . .	31
	User Interfaces	32
4.4.2	Noise Filtering	33
4.4.3	Duplications with Low Density	34
4.4.4	Duplications with Low Number of Matches	34
4.4.5	Redundancies	34
4.5	The Textual Viewer	36
4.5.1	The Selection Part	36
4.5.2	The Qualification Part	37
4.5.3	The Information Part	37
4.5.4	The Source Code Part	38
4.6	The Graphical Viewer	39
5	Validation: Statistical Analysis	40
5.1	Applications in Smalltalk	40
5.1.1	Analyzed Applications	40
5.1.2	Noise Filtration	42
	Distribution of Density	42
	Distribution of Number of Matches	49
5.1.3	Metrics	54
	Scenario Distribution	54
5.1.4	Distribution of Impact on Classes	55
5.1.5	Distribution of Number of Matches	56
5.1.6	Distribution of Length of Duplication	57
5.1.7	Conclusion	57
5.2	Applications in C++ and Java	58
6	Validation: Qualitative Aspects	60
6.1	Case Studies Results	60
6.1.1	Analysis of DUPLOC	61
6.1.2	Analysis of CODECRAWLER	62
6.1.3	Analysis of MOOSE	63
6.2	Scenario based Examples	64
6.2.1	Ancestor Scenario	65
6.2.2	Common Hierarchy Scenario	67
6.2.3	First Cousin Scenario	69
6.2.4	Same Method Scenario	72
6.2.5	Sibling Classes Scenario	74
6.2.6	Same Class Scenario	77
6.2.7	Superclass Scenario	81
6.2.8	Unrelated Classes Scenario	82
6.2.9	Summary	83
7	Conclusions and Perspectives	84
7.1	Conclusions	84
7.2	Perspectives and Future Work	85
7.2.1	Limits of the Approach	85

7.2.2	Future Work	85
A	Appendix	86
A.1	Refactorings	86
A.1.1	Extract Method	86
A.1.2	Pull Up Method	87
A.1.3	Push Down Method	88
A.1.4	Form Template Method	88
A.1.5	Parameterize Method	88
A.1.6	Collapse Hierarchy	89
A.1.7	Extract Superclass	90
A.1.8	Rename Method	90
A.1.9	Replace Subclass with Field	91
A.1.10	Substitute Algorithm	91
A.1.11	Pull Up Field	92
A.1.12	Replace Constructor with Factory Method	92
A.1.13	Pull Up Constructor Body	92
A.1.14	Inline Method	93
A.1.15	Self Encapsulate Field	93
A.2	Structure of the Smalltalk Applications	94
A.2.1	SCG Group	94
	MOOSE	94
	CODECRAWLER	95
	DUPLOC	96
A.2.2	Reference Group	97
	REFACTORINGBROWSER	97
	VISUALWORKS	98
A.2.3	Industrial Group	99
	PDP	99
	MAF	100
A.3	Validation: Statistical Analysis	101
A.3.1	Distribution of Scenarios	101
A.3.2	Impact on Classes	102
A.3.3	Number of Matches	103
A.3.4	Length	104
A.3.5	Number of Matches before Filtering	105

Chapter 1

Introduction

Duplicated code is a phenomenon that occurs frequently in large systems for several reasons (see [31, 18]). Although code duplication can have its justifications, it is considered bad practice. During maintenance, which is estimated at 70% of the overall effort for producing a software system [37], duplicated codes give the following problems [36]:

1. Hindrance to comprehension of the program.
2. Independent evolution of the clones.
3. Bad design.

In this thesis we investigate how duplication can be cured in software systems developed in object oriented programming languages.

1.1 Our Contribution

We present an approach characterized by its simplicity and a language independent tool for analyzing and refactoring duplicated code in an object oriented context.

- We use the relationships between the software entities constrained by the object oriented context to define different scenarios. Each of them determines a set of applicable refactorings.
- To validate the approach, we implemented a program called SUPREMO that:
 1. characterizes the detected duplication,
 2. presents a textual view of the duplications,
 3. draws the graphical overview of the situation,
 4. proposes a set of applicable refactorings for each duplication.

1.2 Structure of the Thesis

In Chapter 2, we discuss the origins and detection of duplications and make a survey of the applicable refactorings. Chapter 3 deals with our approach and in Chapter 4, we present SUPREMO, the implemented tool.

The quantitative validation of the approach is presented in the Chapter 5. The qualitative validation is discussed in Chapter 6. In Chapter 7 we present some conclusions and discuss the perspectives of this work.

Chapter 2

Duplicated Code and Refactoring

In this chapter, we discuss the issues related to code duplication. First the origins, problems and detection of duplicated code is investigated. Then we present refactoring techniques to change, correct, and improve existing code.

At the end of the chapter, we present a specific list of refactorings we can use to eliminate the clones from an object oriented system.

2.1 The Problem of Duplicated Code

Code duplication is one of the factors that severely complicates the maintenance and evolution of large software systems. Techniques for detecting duplicated code exist but rely mostly on parsers, a technology that has proven to be brittle in the face of different languages and dialects.

2.1.1 Software Reuse and Origin of Clones

Software reuse is the process of creating software systems from existing software rather than building everything from scratch. The kinds of artifacts that can be reused are not limited to source code fragments. They may include design structures, module-level implementation structures, specifications, documentation, transformations, and more. Forms of code source reuse include loops, functions, procedures, subprograms, subroutines, software component libraries, inheritance, application generators, generic software templates [39]. The mechanisms of software reuse are well integrated in the software development process.

Many programmers rather adopt an apparently simpler approach to reusing software system designs and source code. They collect fragments from existing software systems and use them as part of new software by simply applying the well known practice that we call *copy-and-paste programming*. This occurs frequently during the development phase when they reuse tried and tested code in a new context. Every developer copies pieces of software. When encountering a familiar problem that has been solved before, it's a normal reflex to reuse the existing code. One does not have to reinvent the wheel. This copy-and-paste programming style leads to duplicated code.

Duplication occurs also during the maintenance phase when the program must be

adapted to the new requirements of the users: a program that is used in a real-world environment must be changed to add new functionality or to adapt to changes in the environment [29]. Since the existing system already treats many problems of the domain, an obvious way to integrate the changes is to copy fragments with only small modifications.

Duplicated code is therefore a phenomenon that occurs frequently in large systems. The reasons why programmers duplicate code are widely discussed in [31, 18]. Some of them are listed below:

- The following conditions in the development environment can increase the trend to code duplication:
 1. There is no time to design, implement, and test a newly developed component. If a programmer cannot finish on time, it's wiser to copy a piece of code that runs properly than to persist in doing a very good design that will not run.
 2. Software systems have become more and more complex. One of the result of a study says that: "*as a program evolves, it becomes more complex, and extra resources are needed to preserve and simplify its structure*" [29]. It becomes very difficult to keep the overview of the design and this leads to plenty of opportunities for code duplication.
 3. Efficiency considerations may make the cost of a procedure call or method invocation seem too high a price.
 4. The productivity of developers is sometimes measured in terms of number of lines of code written. This rewards copy-and-paste rather than writing new code.
- The programmer personality: we all have a natural laziness.
 1. Making a copy of a code fragment is simpler and faster than writing the code from scratch. In addition, the fragment may already have been tested so the introduction of a bug seems less likely.
 2. Making code reusable takes extra efforts.
 3. Rethinking an implementation if you see similarities between different parts of the implementation requires energy.

2.1.2 The Problem

Although copy-and-paste programming helps to meet short term goals (the code is already designed, implemented and debugged), it involves a lot of problems in software maintenance, which is estimated to cost 70% of the overall effort for producing software system in average [37]:

1. It complicates the comprehension of the program.
2. Code duplication increases the size of the code, extending compile time and expanding the size of the executable.

3. It uses more memory and complicates the error detection. Defects found in a code segment that has possibly been copied involves searching the clones of the segment and assessing the impact of the correction in each new context. If one repairs a bug in a system with duplicated code, all possible duplications of that bug must be checked.
4. Code duplication often indicates design problems like missing inheritance or missing procedural abstraction. In turn, such a lack of abstraction hampers the addition of functionality.

2.1.3 Detection of Duplicated Code

The analysis of code in order to identify duplication is a wide domain. Different techniques are used: structural comparison using pattern matching, metrics or statistical analysis of the code, code fingerprints.

In [32], clones are detected by identifying programming patterns. Statistical comparisons are used in [35, 34]. Jankowitz in [33] uses the static execution tree (the call graph) of a program to determine a fingerprint of the program.

Visualization of duplicated code is used by *dotplot* [10] and by DUPLOC [11]. Johnson [12] and Baker [1] do not have graphical support, but provide a report that presents an overall similarity percentage between two files.

The tool of [18] transforms source code into abstract syntax trees and Kontogianis [13] evaluates the use of five data and control flow related metrics for identifying similar code fragments.

However, most of the approaches [31, 32, 13, 18] are based on parsing techniques and thus rely on having the *right* parser for the right dialect for *every* language that is used within an organization. The need for parsing hinders the application of these techniques in an industrial context.

2.2 Refactoring

This section presents an overview of refactoring, and how it helps to remedy code duplication. We mention also at the end a list of refactorings that could be useful to remove duplicated code.

2.2.1 A Survey of Refactoring

Refactoring consists of changing a software system in such a way that it does not alter the external behavior of the program. It is a disciplined way to clean up code [3]. Three of the most important advantages of refactoring are listed below:

1. It improves the design of software.
During its evolution, the design of the program deteriorates as people change code. As programmers change code, for instance to realize short-term goals, sometimes without a full comprehension of the design, the code loses its structure.

Refactoring is a way to restructure the code. In essence when you refactor you are improving the design of the code.

2. It makes software easier to understand.
Someone (maybe yourself) will read your code later in order to make changes. When you write code you don't think of the other programmer who will modify your code, you are preoccupied with the short-term goals that the program must meet.
When you refactor, you describe the code better because you have to understand first the code you are refactoring. Usually you write comments or/and rename the methods with a more communicative name.
3. It helps you to avoid errors.
It is a way to clean up code that minimizes the chances of introducing defects. If you better understand the code you are being refactoring, then you can also find errors in the program more easily.
4. It helps you to maintain and modify a program with more accuracy and speed.
The three points mentioned before (Improving design, understandability and reducing bugs) lead to the conclusion that refactoring improves the quality of design. Good design is essential for rapid software development. You don't spend time finding and fixing bugs instead of adding new function.

Refactoring helps you to develop software more rapidly, because it stops the design of the system from decaying and can even improve a design. Refactorings are however also risky because they require changes to working code.

2.2.2 How Refactoring Can Remedy Code Duplication

Several times during its life cycle, because of its evolution, software must be refactored. We must take advantage of this process to reduce or better eliminate all clones of the system.

There are two cases to distinguish:

1. A complete function is duplicated exactly.
For example functions $F^1, F^2, F^3, \dots, F^n$: in this case the solution is to change the calls to the functions F^2, F^3, \dots, F^n into calls to the function F^1 , and then remove all duplicates but F^1 .
2. Only a piece of code is found as part of a number of functions.
The solution is to extract the piece of code and create a new function N from it. The duplicated code is then replaced everywhere with calls to N.

By eliminating the duplicates, you ensure that the code *says everything once and only once*, which is a rule of good design [27].

2.2.3 Techniques of Refactoring

Refactoring must be done systematically to avoid or reduce the risk of introducing bugs on the working code. From Fowlers catalog [3] of 72 refactorings, we list those

that are important for eliminating duplication. The details of the mechanics of each refactoring are explained in appendix [A.1](#).

1. Extract Method (see Section [A.1.1](#)).

If a code fragment can be grouped together, turn it into a method whose name explains the purpose of the method and replace the fragment with a call to the new method.

2. Pull Up Method (see Section [A.1.2](#)).

Methods with duplicated code in two subclasses of a common ancestor X can be refactored as follows: extract a method (see 1.) in both classes and put it into the superclass X.

Often *Pull Up Method* comes after other steps. You see two methods in different classes that can be parameterized in such a way that they end up as essentially being the same method. A special case of the need for *Pull Up Method* occurs when you have a subclass that overrides a method from the superclass yet does the same thing.

The most awkward element of *Pull Up Method* is that the body of the methods may refer to features that are in the subclass but not in the superclass. If the feature is a method, you can create an abstract method in the superclass.

3. Push Down Method (see Section [A.1.3](#)).

Behavior on a superclass is relevant only for the subclass. Push Down Method is the opposite of *Pull Up Method*.

4. Form Template Method (see Section [A.1.4](#)).

Two methods in subclasses that seem to carry out broadly similar steps in the same sequence, but the steps are not the same. Move the sequence to the superclass and allow polymorphism to play its role, ensure that the different steps do their things differently. This kind of method is called a template method [21].

5. Parameterization (see Section [A.1.5](#)).

Several methods do similar things but with different values contained in the method body, one method that uses a parameter for the different values can be created.

6. Collapse Hierarchy (see Section [A.1.6](#)).

Refactoring the hierarchy often involves pushing methods and fields up and down the hierarchy. After you have done this, you can find you have a subclass that is not adding any value, so you need to merge the classes together.

7. Extract Superclass (see Section [A.1.7](#)).

You have two classes with similar features.

Create a superclass and move the common features to the superclass.

Chapter 3

Classifying Duplication in Refactorable Scenarios

In the previous chapter we mentioned the difficulty to find tools that work between the detection and the refactoring of duplicated code. In this chapter we expose our approach for the removal of duplication and define the outlines of its application domain.

Refactoring in an object oriented context is constrained by the relationships between the software entities, principally between classes. We cannot apply any refactoring in any situation. Only a specific set of refactorings can be used in a given context.

Our idea is to define such situations or “scenarios”, as we call them and to find out the corresponding set of refactorings. Depending on the relationship between classes containing the methods where the duplicated code was found, we have defined different scenarios that characterize the situation and allow the definition of possible cures. The list of defined scenarios is presented below:

1. In the Same Method (see Section [3.3](#)).
2. In the Same Class (see Section [3.4](#)).
3. With a Sibling Class (see Section [3.5](#)).
4. With the Superclass (see Section [3.6](#)).
5. With an Ancestor (see Section [3.7](#)).
6. With a First Cousin (see Section [3.8](#)).
7. In Common Hierarchy (see Section [3.9](#)).
8. In Unrelated Classes (see Section [3.10](#)).

The next section presents some definitions that we need in this work.

3.1 Definition and Properties of Duplication

Duplication or Clones. In this work we define the notion of “duplication” or “clone” as describing an association between two pieces of source code that are considered as copied by the detection tool.

Impact on Classes. The definition of scenario infers that there are only two concerned classes at maximum. The duplication however, could be repeated more than one time and with different scenarios. It is difficult to anticipate all possible combinations of situations. We define the notion of “impact on classes” as the number of different classes participating on the duplication.

3.2 Constraints

The concept of scenario is the basis of our approach: . A set of applicable refactorings is associated to a given scenario. The principle of our approach is to find for each duplication the corresponding scenario and to propose a list of possible refactorings. This is performed in two steps:

1. Duplication detection.

Given the difficulty to find a parser for each dialect (as mentioned in Chapter 2), we opt for the principle of language independence and decide, for the detection of duplication, upon a tool named DUPLOC [11]. Its detection algorithm is based on line comparison of source code using string matching.

2. Scenario definition.

For each detected duplication, we must find the information about the origin of copied code (the method and the class) to establish the scenario. Here also, we opt for language independence and decide, upon a framework named MOOSE [38].

In the following sections, we describe the scenarios we defined. Each section contains a figure to illustrate the relation between entities in the system, a paragraph that describes the scenario and a list of proposed refactorings. This group of refactoring might possibly be used by the developer. He decides if a duplication must be refactored or not, and which refactoring will be used.

3.3 Duplication in the Same Method

Description. Two pieces of code (see Figure 3.1) are duplicated in the same method.

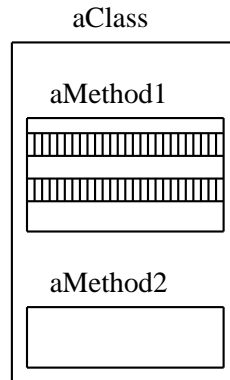


Figure 3.1: Duplication in the same method.

Proposed Refactorings:

- *Extract Method* (see Section A.1.1).
- *Parameterization* (see Section A.1.5).

Discussion. This case represents the simplest scenario. We don't have to take care of side effects between classes. If an extract method is applied, the piece of code is replaced by a call to the newly created method. The signature of the original methods are not changed and a possible client does not see the difference.

If we don't have local variable in the duplicated piece of code, we propose at first the *Extract Method* refactoring for this scenario. In some circumstance (see below), the *Parameterization* could be used or a combination of both refactorings.

The biggest problem with our proposed refactoring, *Extract Method*, is dealing with local and temporary variables. In the simplest case, there is no local variable and the refactoring is trivially easy. Take the example of Figure 3.2 which shows a duplication in the same class found in CODECRAWLER.

Upper Class: **CCGraphSubcanvas**
Upper Method: **changedEdges**

```
changedEdges
self areEdgesChecked
  ifFalse:
    [self disable: #chkEdgesEnabled.
     chkEdgesEnabled value: false.
     self disable: #chkWeightedEdgesEnabled.
     chkWeightedEdgesEnabled value: false]
  ifTrue:
    [self enable: #chkEdgesEnabled.
     self enable: #chkWeightedEdgesEnabled]
```

```
postBuildWith: aBuilder
self changedCheckClasses;
changedCheckMethods;
changedCheckAttributes;
changedCheckFunctions.
self areEdgesChecked
  ifFalse:
    [self disable: #chkEdgesEnabled.
     chkEdgesEnabled value: false.
     self disable: #chkWeightedEdgesEnabled.
     chkWeightedEdgesEnabled value: false]
```

Lower Class: **CCGraphSubcanvas**
Lower Method: **postBuildWith:**

Figure 3.2: Application of Extract Method.

It is easy to extract the code contained in the `ifFalse:` branch of the conditional. It is just a cut, paste, name and put in a call:

```
CCGraphSubcanvas>>disableEdges
  self disable: #chkEdgesEnabled.
  chkEdgesEnabled value: false.
  self disable: #chkWeightedEdgesEnabled.
  chkWeightedEdgesEnabled value: false
```

Both methods are changed and the duplication is removed. A third method are created.

Upper Class: **CCGraphSubcanvas**
Upper Method: **changedEdges**

```
changedEdges
  self areEdgesChecked
  ifFalse:
    [self disableEdges]
  ifTrue:
    [self enable: #chkEdgesEnabled.
     self enable: #chkWeightedEdgesEnabled]
```

```
postBuildWith: aBuilder
  self changedCheckClasses;
  changedCheckMethods;
  changedCheckAttributes;
  changedCheckFunctions.
  self areEdgesChecked
  ifFalse:
    [self disableEdges]
```

Lower Class: **CCGraphSubcanvas**
Lower Method: **postBuildWith:**

Figure 3.3: Duplication refactored with Extract Method.

With Local Variable. The problem is that local variables are only in scope in that method. The easiest case with local variable is when the variables are read but not changed. In this case we can just pass them in as a parameter (see discussion in Section 3.5).

If the local variable is assigned, the simplest case is that in which the variable is a temporary variable used only within the extracted code. In this case you can move the variable into the extracted code.

If the assigned local variable is used outside the extracted code, it is difficult to give a general solution. The situation must be deeply checked.

3.4 Duplication in the Same Class

Description. Two different methods of the same class contain the same piece of code (see Figure 3.4).

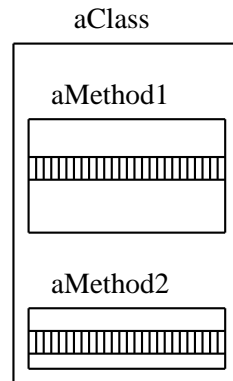


Figure 3.4: Duplication in the same class.

Proposed Refactorings:

- *Extract Method* (see Section A.1.1).
- *Insert Method Call*.
- *Parameterization* (see Section A.1.5).
- *Form Template Method* (see Section A.1.4).

Discussion. We propose four refactorings which could be applied each alone or in combination. The *Extract Method* was discussed in the previous section, we present below typical examples where we could apply *Insert Method Call* and *Parameterization*. *Form Template Method* is discussed in Section 3.5.

Insert Method Call could be applied when one method is entirely copied in the other method or when an other method could be called with a special value (see Figure 3.5).

Upper Class: **LineByLineReader**
Upper Method: **getRecordUntilEmptyLine**

```

getRecordUntilEmptyLine
  "returns a collection of all lines up to the occurrence
  of an empty line (not returning this line if removeRecordDelimiter
  is set to true) or it returns all the lines until the end of the input
  if no empty line is found"

  | lineColl found line |
  lineColl := OrderedCollection new.
  readStream isNil ifTrue: [^lineColl].
  line := ''.
  found := false.
  [line isNil | found]
    whileFalse: [(line := self getNextLineFromBuffer) isNil
      iffFalse:
        [line = '' ifTrue: [found := true].
          found not | removeRecordDelimiter not
            ifTrue: [lineColl addLast: line]].
        ^lineColl
  ]

```

```

getRecordDelimitedBy: aStringOrNil
  "returns a collection of all lines up to the occurrence
  of a line prefixed by aString (including this line)
  or it returns all the lines until the end of the input
  if the
  is nil"

  | lineColl found line |
  lineColl := OrderedCollection new.
  readStream isNil ifTrue: [^lineColl].
  line := ''.
  found := false.
  [line isNil | found]
    whileFalse: [(line := self getNextLineFromBuffer) isNil
      iffFalse:
        [aStringOrNil notNil
          ifTrue: [found :=
            (line indexOfSubCollection: aStringOrNil
              startingAt: 1) == 1].
          aStringOrNil isNil | (found not | removeRecordDelimiter not)
            ifTrue: [lineColl addLast: line]].
        ^lineColl
  ]

```

Lower Class: **LineByLineReader**
Lower Method: **getRecordDelimitedBy:**

Figure 3.5: Application of Insert Method Call.

The method `#getRecordUntilEmptyLine` could be replaced by the following code:

```
LineByLineReader>>#getRecordUntilEmptyLine
  "...comment..."
  self getRecordDelimitedBy: ' '
```

The duplication is then eliminated and the code becomes:

Upper Class: **LineByLineReader**

Upper Method: **getRecordUntilEmptyLine**

```
getRecordUntilEmptyLine
"returns a collection of all lines up to the occurrence
of an empty line (not returning this line if removeRecordDelimiter
is set to true) or it returns all the lines until the end of the input
if no empty line is found"

^self getRecordDelimitedBy: ' '
```

```
getRecordDelimitedBy: aStringOrNil
"returns a collection of all lines up to the occurrence
of a line prefixed by aString (including this line)
or it returns all the lines until the end of the input
if the
is nil"

| lineColl found line |
lineColl := OrderedCollection new.
readStream isNil ifTrue: [^lineColl].
line := ''.
found := false.
[line isNil | found]
  whileFalse: [(line := self getNextLineFromBuffer) isNil
ifFalse:
  [aStringOrNil notNil
  ifTrue: [found :=
    (line indexOfSubCollection: aStringOrNil
    startingAt: 1) == 1].
  aStringOrNil isNil | (found not | removeRecordDelimiter not)
  ifTrue: [lineColl addLast: line]].
^lineColl
```

Lower Class: **LineByLineReader**

Lower Method: **getRecordDelimitedBy:**

Figure 3.6: Duplication refactored with Insert Method Call.

Parameterization This refactoring is applied if methods that do similar things but vary depending on few values. Since we are in the same method scenario, we can not use polymorphism, in this case we can replace the separate methods with a single method that handles the variation by parameters.

The parts of code corresponding to the difference are the assignments of local vari-

Upper Class: **CodeParticle**

Upper Method: **mergeUsingOr:**

```
mergeUsingOr: aCodeParticle
(self safetyCheckWith: aCodeParticle)
  ifTrue:
    [| start end trueVal falseVal newCFun |
     start := lowerBound min: aCodeParticle lowerBound.
     end := upperBound max: aCodeParticle upperBound.
     trueVal := self nonHoleSymbol.
     falseVal := self holeSymbol.
     newCFun := self createEmptyCharactFun.
     start to: end do: [:pos | (self charactFunAt: pos) = trueVal
     | ((aCodeParticle charactFunAt: pos) = trueVal)
       ifTrue: [newCFun addLast: trueVal]
       ifFalse: [newCFun addLast: falseVal]].
     charactFun := newCFun.
     lowerBound := start.
     upperBound := end]
  ifFalse: [self sourceObject = aCodeParticle sourceObject
    & self isEmpty ifTrue: [self copyFrom: aCodeParticle]]
```

```
mergeUsingAnd: aCodeParticle
(self safetyCheckWith: aCodeParticle)
  ifTrue: [(self boundariesOverlapWith: aCodeParticle)
    ifTrue:
      [| start end trueVal falseVal newCFun |
       start := lowerBound max: aCodeParticle lowerBound.
       end := upperBound min: aCodeParticle upperBound.
       trueVal := self nonHoleSymbol.
       falseVal := self holeSymbol.
       newCFun := self createEmptyCharactFun.
       start to: end do: [:pos | (self charactFunAt: pos) = trueVal
       & ((aCodeParticle charactFunAt: pos) = trueVal)
         ifTrue: [newCFun addLast: trueVal]
         ifFalse: [newCFun addLast: falseVal]].
       charactFun := newCFun.
       lowerBound := start.
       upperBound := end]]
  ifFalse: [self sourceObject = aCodeParticle sourceObject
    & aCodeParticle isEmpty ifTrue: [self makeEmptyParticle]]
```

Lower Class: **CodeParticle**

Lower Method: **mergeUsingAnd:**

Figure 3.7: Application of Parameterization.

ables start and end, and the conditional: (self charactFunAt: pos) = trueVal & ((aCodeParticle charactFunAt: pos) = trueVal) and (self charactFunAt: pos) = trueVal | ((aCodeParticle charactFunAt: pos) = trueVal). We can create one method that uses a parameter for the different values. For example:

```

merge: aCodeParticle usingOperation: aSymbol

    | start end trueVal falseVal newCFun aBoolean result|
aBoolean := aSymbol = #And.
aBoolean ifTrue: [ start := lowerBound max: aCodeParticle lowerBound.
                  end := upperBound min: aCodeParticle upperBound]
              ifFalse:[ start := lowerBound min: aCodeParticle lowerBound.
                       end := upperBound max: aCodeParticle upperBound]
trueVal := self nonHoleSymbol.
falseVal := self holeSymbol.
newCFun := self createEmptyCharactFun.
start to: end do: [:pos |
aBoolean ifTrue: [ result:= (self charactFunAt: pos) = trueVal
                        & ((aCodeParticle charactFunAt: pos) = trueVal)]
              ifFalse:[ result:= (self charactFunAt: pos) = trueVal
                       | ((aCodeParticle charactFunAt: pos) = trueVal)].
result ifTrue: [newCFun addLast: trueVal]
              ifFalse: [newCFun addLast: falseVal].
charactFun := newCFun.
lowerBound := start.
upperBound := end

```

This method could be used in the original methods:

Upper Class: **CodeParticle**

Upper Method: **mergeUsingOr:**

```

mergeUsingOr: aCodeParticle
(self safetyCheckWith: aCodeParticle)
  ifTrue:
    [self merge: aCodeParticle usingOperation: #Or]
  ifFalse: [self sourceObject = aCodeParticle sourceObject
            & self isEmpty ifTrue: [self copyFrom: aCodeParticle]]

```

```

mergeUsingAnd: aCodeParticle
(self safetyCheckWith: aCodeParticle)
  ifTrue: [(self boundariesOverlapWith: aCodeParticle)
            ifTrue:
              [self merge: aCodeParticle usingOperation: #And]
            ifFalse: [self sourceObject = aCodeParticle sourceObject
                      & aCodeParticle isEmpty ifTrue: [self makeEmptyParticle]]

```

Lower Class: **CodeParticle**

Lower Method: **mergeUsingAnd:**

Figure 3.8: Duplication refactored with Parameterization.

3.5 Duplication between Sibling Classes

Description. By sibling classes (see Figure 3.9) we refer to all classes with the same direct superclass and with the same hierarchical level. The highlighted rectangles represent the classes in which the methods containing duplicated code were found.

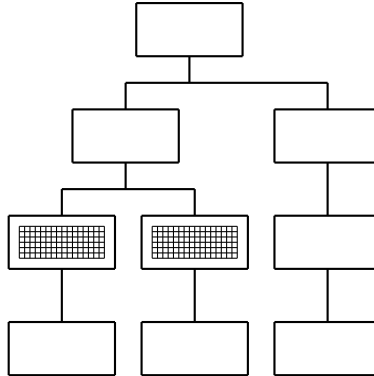


Figure 3.9: Duplication between sibling classes.

Proposed Refactorings:

- *Pull Up Method* (see Section A.1.2).
- *Parameterization* (see Section A.1.5).
- *Extract Method* (see Section A.1.1).
- *Substitute Algorithm* (see Section A.1.10).
- *Form Template Method* (see Section A.1.4).
- *Replace Subclass with Field* (see Section A.1.9).
- *Extract Superclass* (see Section A.1.7).

Discussion. The experiments leads in this work (see Section 6.1) show the trend to pull up into the superclass the extracted duplication by using *Form Template Method* and *Pull Up Method*.

We illustrate the *Form Template Method* refactoring with the following example:

```
import
  "self importMyself"

| class |
class := [
  self startUIProgressFeedback: 'Import (by querying Smalltalk
    repository)' maxProgressMeasure: smalltalkClasses size.
  classCounter := 0.
  smalltalkClasses
  collect:
    [:class |
      classCounter := classCounter + 1.
      self nextUIProgressFeedback: class name
      progressMeasure: classCounter.
    ]
  self perform: classCreatorMessage with: class class with: true.
  self perform: classCreatorMessage with: class with: false]]
valueNowOrOnUnwindDo: [ self terminateUIProgressFeedback].
self optimize.
^class
```

```
import
  "self importMyself"

| classEntities |
classEntities := [
  self startUIProgressFeedback: 'Import (by parsing
    method-sources)' maxProgressMeasure: smalltalkClasses size.
  classCounter := 0.
  smalltalkClasses
  collect:
    [:class |
      classCounter := classCounter + 1.
      self nextUIProgressFeedback: class name
      progressMeasure: classCounter.
    ]
  self buildEntitiesFromSmalltalkClass: class ]]
valueNowOrOnUnwindDo: [ self terminateUIProgressFeedback].
self optimize.
^classEntities
```

Figure 3.10: Application of Form Template Method.

The duplication in Figure 3.10, which is a duplication in sibling classes, is a good candidate for the creation of a template method: both methods in subclasses perform similar steps in the same order, yet the steps are different. We must get the steps into methods with the same signature, so that the original methods become the same.

```

import
  "self importMyself"

  | classEntities |
classEntities := [
  self startUIProgressFeedback: self myComment
  maxProgressMeasure: smalltalkClasses size.
classCounter := 0.
smalltalkClasses
  collect:
    [:class |
      classCounter := classCounter + 1.
      self nextUIProgressFeedback: class name
      progressMeasure: classCounter.
self buildEntitiesFrom: class ]]
valueNowOrOnUnwindDo: [ self terminateUIProgressFeedback].
self optimize.
^classEntities

```

Figure 3.11: Duplication refactored with Form Template Method.

We create in each subclass the methods #myComment and #buildEntitiesFrom: which are as follow defined:

```
MSEVisualWorksImporter>>#myComment
```

```
  ^Import (by querying Smalltalk repository)'
```

```
MSEVisualWorksImporter>>#buildEntitiesFrom: class
```

```
  self perform: classCreatorMessage with: class class with: true.
  self perform: classCreatorMessage with: class with: false
```

```
MSEVisualWorksParsingImporter>>#myComment
```

```
  ^Import (by parsing method-sources)'
```

```
MSEVisualWorksParsingImporter>>#buildEntitiesFrom: class
```

```
  self buildEntitiesFromSmalltalkClass: class
```

In both subclasses, we have now exactly the same method. We can allow inheritance and polymorphism to play their role by pulling up the common method.

3.6 Duplication with Superclass

Description. This scenario describes a duplication between a class and its direct superclass.

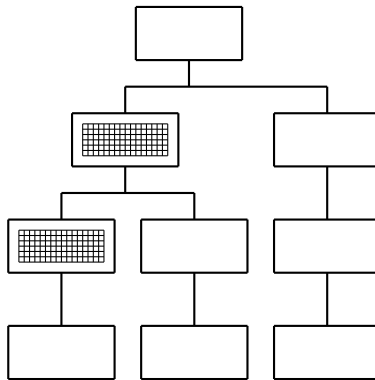


Figure 3.12: Duplication with Superclass.

Proposed Refactorings:

- *Insert Super Call*
- *Parameterization* (see Section [A.1.5](#)).
- *Pull Up Method* (see Section [A.1.2](#)).
- *Push Down Method* (see Section [A.1.3](#)).
- *Form Template Method* (see Section [A.1.4](#)).

Discussion. If both methods have the same name, we can think on a template method for the refactoring or the duplication could also be eliminated by extracting method from both classes and then by putting it into the superclass.

3.7 Duplication with Ancestor

Description. This scenario describes the case where one class inherits from the other but not directly (see Figure 3.13).

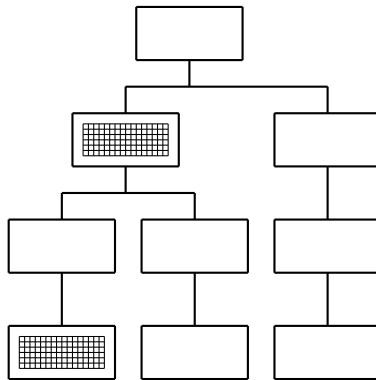


Figure 3.13: Duplication with Ancestor.

Proposed Refactorings:

- *Extract Method* (see Section A.1.1).
- *Parameterization* (see Section A.1.5).
- *Pull Up Method* (see Section A.1.2).
- *Form Template Method* (see Section A.1.4).

Discussion. The difference to the previous scenario (with superclass) is that if we modify something in the ancestor, all classes between the ancestor and the concerned subclass are also affected by the change. We must be more vigilant for where we define the new created method. If we put it into the ancestor class, more classes are affected than in the case of superclass scenario.

3.8 Duplication with First Cousin

Description. This scenario describes the case where both classes have the same hierarchical level and their superclasses are sibling classes (see Figure 3.14).

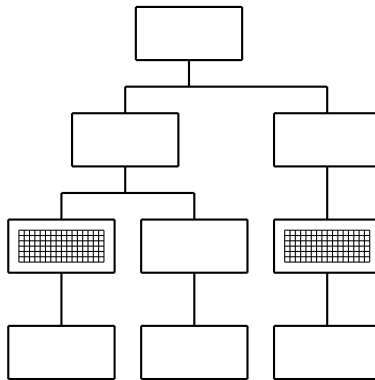


Figure 3.14: Duplication with First Cousin.

Proposed Refactorings:

- *Pull Up Method* (see Section A.1.2).
- *Form Template Method* (see Section A.1.4).
- *Extract Method* (see Section A.1.1).
- *Parameterization* (see Section A.1.5).
- *Extract Superclass* (see Section A.1.7).

Discussion. Using inheritance we can also pull up the extracted method two levels upper in the hierarchy. We must check if there are other classes with the same ancestor involved in the duplication. If yes, “a flawed design” is a probability. May all subclasses containing the same code need a common superclass (*Extract Superclass*). One possibility is to extract a new superclass up to all concerned class and to put into it the new created component.

3.9 Duplication in Common Hierarchy

Description. In this scenario, both classes have the same root or in languages that have multiple inheritance, the sets of ancestors of each class are not disjoint.

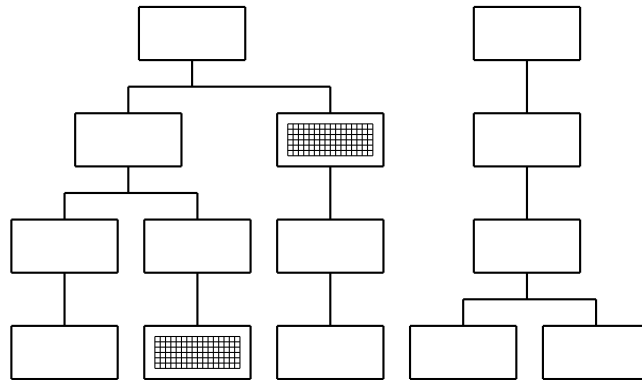


Figure 3.15: Duplication in Common Hierarchy.

Discussion. The classes in this scenario generally have different hierarchical levels. The consequences of a refactoring is more complicated to predict because more classes are involved. It is difficult to say which refactoring is appropriate for this scenario. A graphical representation of the hierarchical inheritance tree where the concerned classes are indicated is a good help for the decision.

Proposed Refactorings:

- *Pull Up Method* (see Section [A.1.2](#)).
- *Parameterization* (see Section [A.1.5](#)).
- *Extract Method* (see Section [A.1.1](#)).
- *Form Template Method* (see Section [A.1.4](#)).
- *Extract Superclass* (see Section [A.1.7](#)).

3.10 Duplication in Unrelated Classes

Description. This scenario describes the case where both classes do not have any common ancestor (see Figure 3.16).

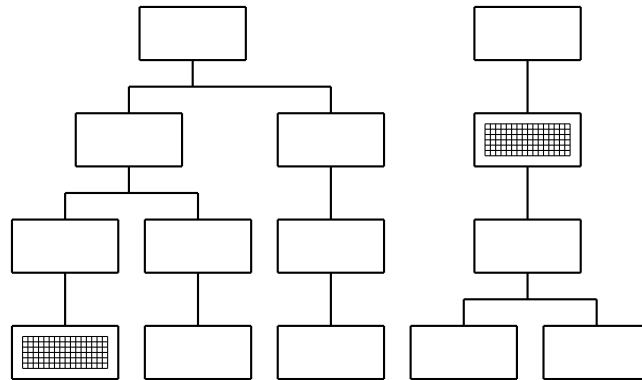


Figure 3.16: Duplication in Unrelated Classes.

Proposed Refactorings: Proposing a solution for this situation is the most difficult one. If you have duplicated code in two unrelated classes, consider extracting a class from one class and then use the new component. If the method really belongs only in one of the classes, the other class should invoke it. You have to decide where the method makes sense and ensure it is there and nowhere else.

3.11 Summary

In the table 3.1 are grouped the refactorings we propose pro scenario.

	Ancestor	Common Hierarchy	First Cousin	Same Method	Sibling	Single Class	Superclass	Unrelated
Extract Method	✓	✓	✓	✓	✓	✓		
Insert Method Call				✓		✓		
Insert Super Call							✓	
Parameterization	✓	✓	✓	✓	✓	✓	✓	
Pull Up Method	✓	✓	✓		✓		✓	
Form Template Method	✓	✓	✓	✓	✓	✓	✓	
Push Down Method							✓	
Extract Superclass		✓	✓		✓			

Table 3.1: Proposition of Refactoring depending on Scenario.

Chapter 4

SUPREMO: Tool Support for Duplication Elimination

4.1 Introduction

SUPREMO (SUPport for REfactoring Method Objects) is the prototype implemented during this work. It has been written entirely in Smalltalk VISUALWORKS in the Envy 4.0 environment. The tool, based on the scenario approach, can:

1. analyze and categorize duplications
2. compute the impact of duplications
3. navigate between duplications
4. display the structure of a system
5. localize in this view the duplications
6. compute metrics
7. propose refactorings and apply them for some cases

There are two main user interfaces: the textual viewer and the graphical viewer, which are described in Section 4.5 and in Section 4.6. When the analyzed application is written in Smalltalk, from the textual viewer the REFACTORINGBROWSER can be called directly opened on the method the developer want to refactor. He has thus the possibility to extract, push up, push down, remove methods. He can also move the methods to an other class or see the senders of the method.

4.2 Requirements and functionality

Recall: In this work we define the duplication or clone as an association between two pieces of code that are considered as copied by the detection tool. The scenario characterize the relationship between the classes where the copied code was found.

When we began the implementation of SUPREMO, we had some requirements and functionality in mind which we thought would be necessary for the program to have:

Language Independence. Techniques for detecting and removing duplicated code exist but are only applicable for a given language or dialect. For example Balazinska in [17] proposes a tool that works only with Java programs.

Legacy systems, however, often have a number of different languages. In industrial software development contexts, resources are finite and one does not have the luxury to choose the development platform. Consequently any maintenance tool must integrate well with whatever program language already in place.

We wanted to circumvent this hindrance by applying a language independent and visual approach; a tool that requires no parsing.

Information access. The user interface of SUPREMO should provide the developer the following properties of the duplications:

- He should see the origin of each duplicated code: the class, the name of the method. Was the method entirely copied or not? How many lines of code were exactly the same in the concerned pieces of code?
- Each duplication should be classified in term of scenarios. The user should have the relationship between the entities of the system.
- The duplication could be repeated more than one time (see Section 3.1). He must be able to find all copied code that form clusters. The refactoring of a duplication in a cluster must take in account the other clones. The developer should have the impact of each duplication on the other classes.

Interactivity. Through means of direct-manipulation interfaces, we wanted to give the user of SUPREMO the possibilities listed below. He should be able

- to navigate through the duplicated code, to remove from the list of clones to be refactored, those that stay in the program.
- to select the duplications with a given scenario. He could want to work with only the scenario in the same method, for example.
- to choose the length and/or the density of duplications he wants to see.
- to visualize, in a global view, all the hierarchical relationships between classes. This representation should convey, in a overview fashion, information about the duplication found.
- to refactor directly from the main window where the duplicated code is displayed.

4.3 Required Applications

The frameworks that SUPREMO needs to run are presented bellow:

4.3.1 HotDraw

HOTDRAW is a two-dimensional graphics framework for structured drawing editors. A HOTDRAW application edits drawings that are made up of figures. Figures are graphics elements such as lines, boxes, and text, and they can represent other objects. A drawing editor built from HOTDRAW contains a set of tools that are used to manipulate the drawing. When a figure is selected by the selection tool, it presents a set of handles. Manipulating a handle changes the properties of its figure or performs some action. For further information on the HOTDRAW framework, which is still being maintained, see also [9, 25, 26]

4.3.2 Duploc

DUPLOC [11] is a language independent approach for detecting duplicated code. The approach is based on (1) *simple* line-based string matching, (2) *visual presentation* of the duplicated code and (3) detailed *textual reports* from which overview data can be synthesized.

DUPLOC transforms the source code slightly using string manipulation operations into an internal format, and to compare the transformed lines, it uses basic string matching. The transformation reduces the entire source code file to an ordered collection of effective lines that will be compared against itself and line collections from other files.

The comparison of two lines is done by string matching. The result is a boolean `true` for an exact match and a `false` otherwise. This value is stored in a matrix, taking the coordinates that the two compared entities have in their respective ordered collections as the matrix coordinates for the comparison result. Comparison sequences, e.g. sequences of matched lines, are then extracted from the matrix in a separate pass.

The algorithm as stated above does not catch duplicated code that was changed inside one line of code. In a sequence of copied code that is compared with the original sequence, a changed line shows up as a hole in the diagonal match pattern. To cope for this weakness when extracting whole copied sequences, a pattern matcher is run over the matrix which captures diagonal lines and allows holes up to a certain size in the middle of the line.

4.3.3 FAMIX

The implementation of the FAMIX metamodel (see Figure 4.1) written in Small talk is called MOOSE and has been developed at the University of Bern. The model itself is a language-independent database of Object Oriented entities. Once a model has been built, we can make queries to the model and its entities. Suppose we have an entity representing a class. We can now ask this class to give us all its methods, attributes etc. We can build models out of systems written in other Object Oriented programming languages than Small talk through an interface called CDIF.

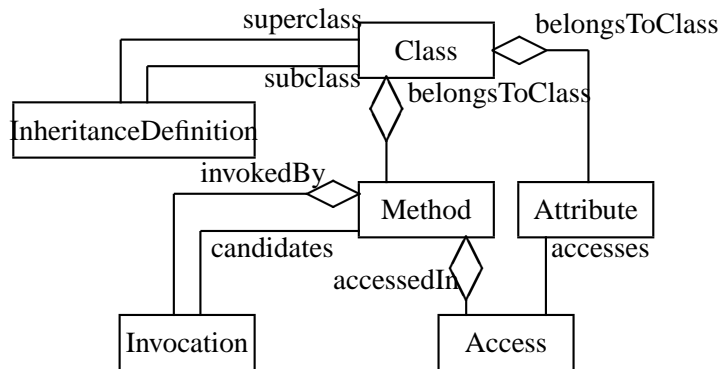


Figure 4.1: The FAMIX Data Model

The representation of Object Oriented source code named FAMIX (FAMoos Information EXchange model [24]) is also defined in the context of the FAMOOS project and exploits meta-modelling techniques to make the data model extensible.

4.4 Tool Architecture

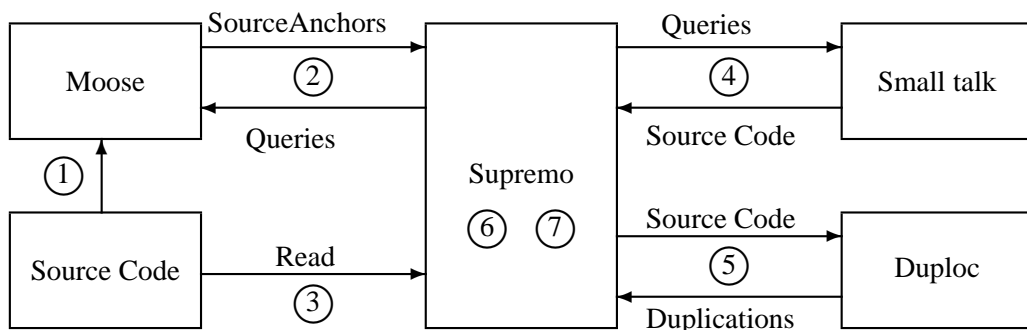


Figure 4.2: Implementation of Supremo

In Figure 4.2 we can see the process SUPREMO follows.

1. Create the model.
The source code is read by MOOSE, which creates and stores a model of the system (for details see [38]).
2. Query the model.
SUPREMO can now query the entities of the system to obtain the position of the source code.
3. Read the sources.
With the positions in the source file, SUPREMO collects the source code of each

method of the system. For the moment we analyze all methods without exception. In the future, we will give the user the possibility to choose which entities he wants to investigate.

4. Read Smalltalk sources.

The programs written in Small talk can be read directly by SUPREMO without passing through MOOSE. We use the metamodel of Small talk to obtain the source code of the methods.

5. Feed DUPLOC.

The collected source code is given to DUPLOC for the duplication detection. DUPLOC give as output a collection of sequences of lines of code (for details see [11]). The sequences given by DUPLOC associate two different pieces of code with a characteristic function which gives for each sequence the lines that match in the two sources.

6. Analyze the duplications.

SUPREMO queries the model (MOOSE or Smalltalk) to find the relationship between entities associated in the duplications given by DUPLOC and thus finds which scenario corresponds to the duplication.

7. Reduce noise.

The output of DUPLOC contains noisy dates that we muss eliminate (see section 4.4.2).

4.4.1 Implementation

This section deals with a few aspects regarding the implementation of SUPREMO. We present below some of the most important classes of SUPREMO.

Creation and Management of the Object DuplicationUnit

DuplicationUnit is the most important class of the application (see Figure 4.3). It contains the information of the metamodel, the scenario, the source code and the status of the duplication. It is the class that is connected with MOOSE and DUPLOC via its instance variable metaModel and comparisonSequence.

DuplicationUnit computes the properties of duplication like: the length, the number of matches and the density (see definition in Section 4.4.2).

Metamodel is the connection with MOOSE or Smalltalk depending of the language of analyzed application. MOOSE serves primarily as a database for the relation between entities and for the localization of the source code to analyze. If the application is written in Smalltalk, we don't need MOOSE and find directly the information from the Smalltalk metamodel.

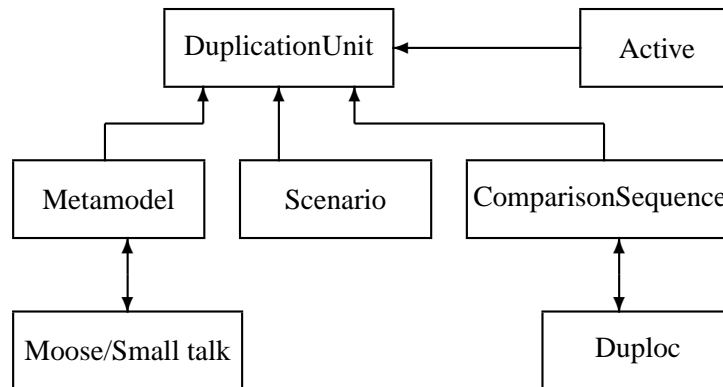


Figure 4.3: DuplicationUnit Class.

ComparisonSequence is the connection with DUPLOC, which gives its results as a collection of sequences of code. SUPREMO groups together all duplication concerning the same source code and create a duplicationUnit.

Scenario With the comparisonSequence and the metaModel, SUPREMO find the corresponding scenario and point out the other duplications that form a cluster with the current duplication.

Active indicates if the duplication is disabled or not. We need this state for the selection and navigation in the textual viewer(see Figure 4.6).

DuplicationUnitsManager is the class that manages the duplicationUnits. It contains three collections:

1. **duplicationUnits** contains the duplications (active or not) that the developer wants to refactor.
2. **removed** contains the duplication the developer decided to delete from the list. Those duplications are not visible any more from the user interface.
3. **analyzedClasses** contains only the analyzed classes of the application. In Smalltalk for instance, we must isolate the classes of the analyzed application from the other classes of Smalltalk otherwise we could not find unrelated classes because all classes inherit from the class Object.
This class is responsible for the operations with duplications: selection, removing from the list, computing metrics, etc.

User Interfaces

Textual Viewer displays only active duplications (which were selected by the developer) and is responsible for the navigation.

Graphical Viewer shows a global view of analyzed classes with the inheritance hierarchy. The color of the class informs if the class contains duplications, if it participates to a cluster of duplications with the current duplication displayed in the textual viewer. The graphical viewer is the connection with HOTDRAW. SUPREMO subclasses three classes of HOTDRAW:

1. **DrawingEditor.** This is done through the class `SupremLayout` which is the main class of the graphical viewer.
2. **RectangleFigure.** The subclass is named `ClassFigure` and is the graphical representation of a class and directly references it through its instance variable *model*.
3. **Tool.** The class `SupremTool` implements the method which is responsible for displaying the class name in the lower left corner of the graphical viewer, when the mouse pointer is floating above a `ClassFigure`.

4.4.2 Noise Filtering

The algorithm of duplication detection of DUPLOC allows the presence of holes in the duplication. That means we have sequences with lines that do not match inside the clone (see Figure 4.4). If the ratio of matched lines to unmatched lines drops too low, the sequence of duplicated code becomes uninteresting to refactor. We then consider the sequence as noise.

Definitions: we define three parameters we will need to filter out the noise.

- The *length* of a duplication is the number of lines forming the sequence DUPLOC has considered as duplication. In Figure 4.4 the sequence begins at the line containing *mbline*. and terminate with the line containing *mb*. The length of duplication in this example is 6.
- The *number of matches* is the number of lines that match inside a duplication. In Figure 4.4 the number of matches is 2.
- The *density of duplication* is the number of matches divided by the length of duplication. In Figure 4.4 the density is 0.3333.

We classified the noise in three categories:

1. Low density
2. Low number of matches
3. Redundancy

In the next subsections we discuss how SUPREMO manages this problem.

4.4.3 Duplications with Low Density

The Figure 4.4 presents an example of duplication with low density. This duplication has no signification, it could be considered as a false positive. It is a coincidence that the lines that match are separated with the same number of line and be considered as duplication by the detection tool. This kind of duplication should be filtered out by SUPREMO.

```

addColorsSubMenuTo: mb
...
mb line.
mb add: 'Progressive Weighted Edges Coloring' -> [drawing progressiveEdgeColoring].
mb line.
mb add: 'Color Settings' -> [self openColorSettings].
mb endSubMenu.
^mb

```

```

addTransformationSubMenuTo: mb
...
mb line.
mb add: 'Translate All Items To Origin' -> [drawing translateToOrigin].
mb add: 'Translate All Items...' -> [drawing translateAllByValue].
mb add: 'Translate Selected Nodes...' -> [drawing translateSelectedByValue].
mb endIndexSubMenu.
^mb

```

Figure 4.4: Example of noise: a duplication with low density.

4.4.4 Duplications with Low Number of Matches

Depending on the programming language, some idioms are repeated along in the source code (see Figure 4.5) and are not considered as duplication.

```

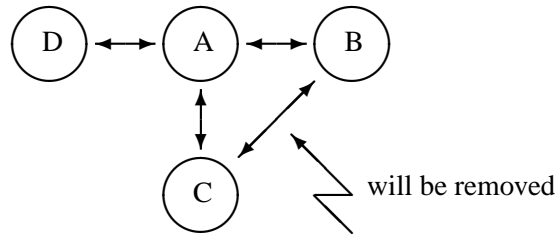
aMethod
...
  aCollection
    inject: 0
    into:
...

```

Figure 4.5: Example of noise: a duplication with low number of matches.

4.4.5 Redundancies

The duplication detection tool we use matches all lines versus all. This imply that if a piece of code of an entity A is copied in B and C, the output of DUPLOC will contain three different duplication: (A,B), (A,C), (B,C). The third duplication (B,C) is redundant and SUPREMO removes it from the collection of duplications.



To remove redundancy, we collect from the list of detected duplication all entities associated in a duplication. In our case, the result is $\{A, B, C, D\}$. For each element, we apply the following algorithm:

- Let be A the first candidate. We select all duplications that contain A $\rightarrow \{(A,B), (A,C), (A,D)\}$
- From the obtained collection we collect all entities associated with A in a duplication $\rightarrow \{B, C, D\}$
- We select from the list of duplications, those having both associated entities in the collection $\{B, C, D\} \rightarrow \{(B,C)\}$. Those duplications are the candidates for redundancy.
- We select in $\{(A,B), (A,C), (A,D)\}$ the duplications that overlap with (B, C) $\rightarrow \{(A,B), (A,C)\}$.
- $\{(A,B), (A,C), (B,C)\}$ forms a cluster with redundant duplication. We select the duplication having the smallest length and remove it from the original list of duplications
- We repeat the operation with the following candidate and the new list of duplications

4.5 The Textual Viewer

The textual viewer is the interface where the source code is displayed. The Figure

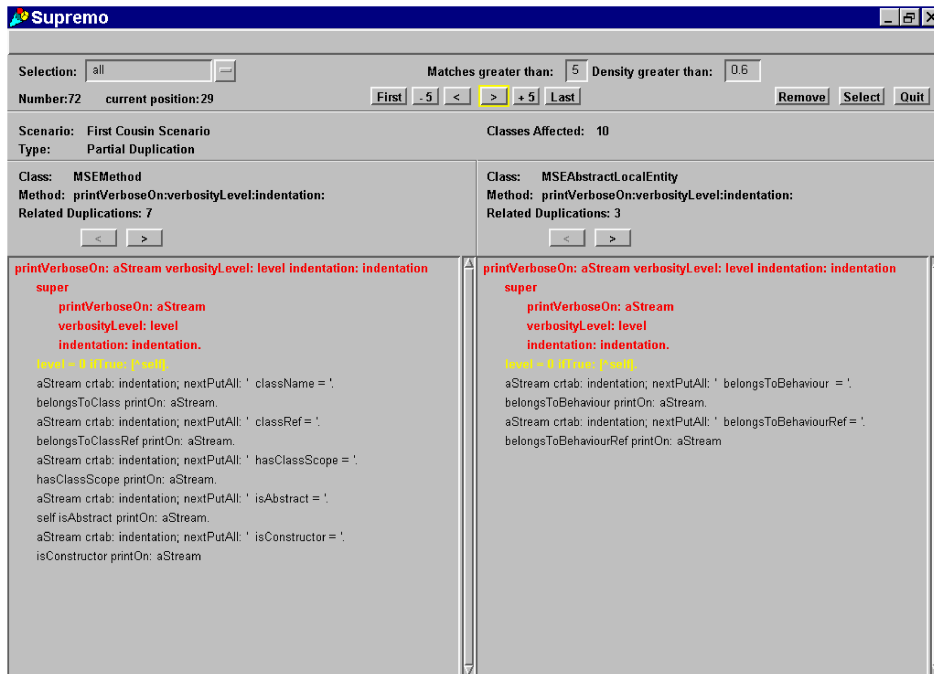
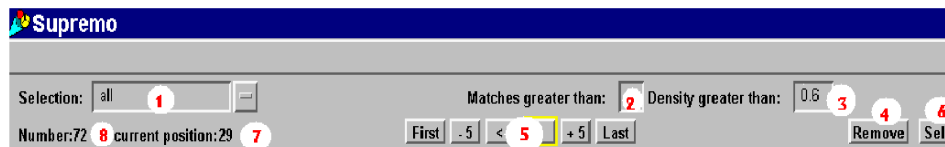


Figure 4.6: The Supremo Textual Viewer

4.6 shows a view of the main window. In the following sections, we explain the main window from the top to the bottom.

4.5.1 The Selection Part

This part is designed to select and navigate between duplications.

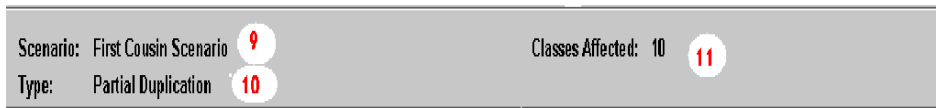


It includes seven functional zones

1. a drop down list for the selection of the scenario of the duplications (*item # 1*). The user can select the duplications with a given scenario.
2. a widget for the selection of the minimum of number of matches (*item # 2*).
3. a widget for the selection of the minimal density (*item # 3*).
4. remove-Button: the duplication is removed from the system (*item # 4*).
5. select-Button: starts the selection of the duplications depending on the selected values (*item # 6*).

6. six buttons for the navigation between the duplications (*item # 5*).
7. the number of selected duplications (*item # 8*).
8. the position of the current displayed duplication in the selected duplications (*item # 7*).

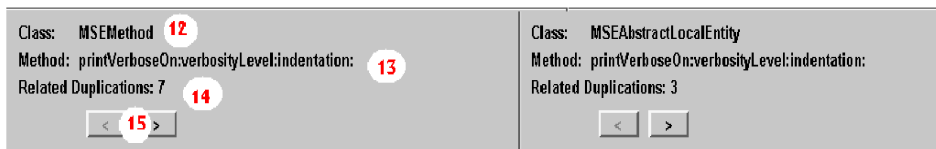
4.5.2 The Qualification Part



This part gives the characterization of the duplications. Three parameters are displayed:

1. the scenario (*item # 9*),
2. the type (*item # 10*). SUPREMO can detect if a method is entirely duplicated or not.
3. the impact (*item # 11*) of the current duplication expressed in number of classes. The impact measures how many classes are affected by a duplication. All those classes are involved if the user refactor the duplication.

4.5.3 The Information Part

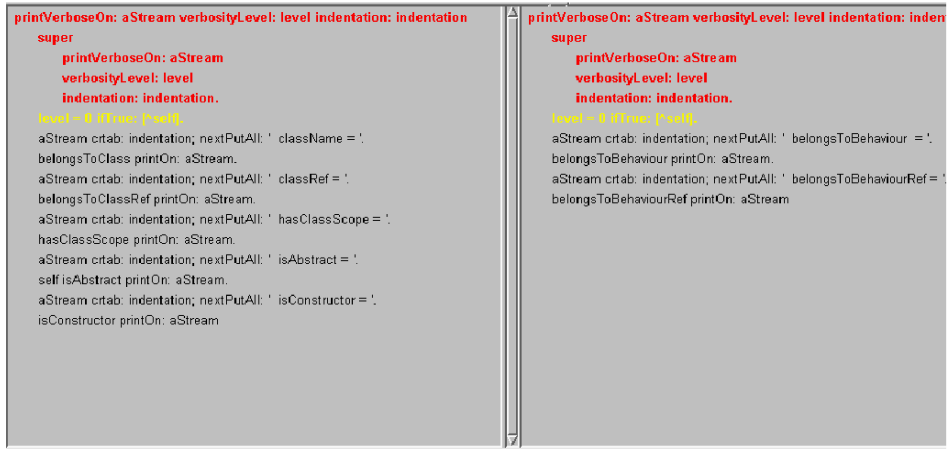


The third part gives information concerning the two methods associated in the duplication.

It includes four zones:

1. the class name (*item # 12*)
2. the method name (*item # 13*)
3. the occurrence of the copied code in all duplications (*item # 14*). This is also a measure of impact of the duplication but on the methods level. The user can see how many times a method is part of a clone.
4. navigation between all occurrences of the copied code (*item # 15*). The navigation gives the opportunity to see the other clones of which the method is part.

4.5.4 The Source Code Part



```
printVerboseOn: aStream verbosityLevel: level indentation: indentation
super
  printVerboseOn: aStream
  verbosityLevel: level
  indentation: indentation.
level := 0 ifTrue: {self}.
aStream crtab: indentation; nextPutAll: ' className = '.
belongsToClass printOn: aStream.
aStream crtab: indentation; nextPutAll: ' classRef = '.
belongsToClassRef printOn: aStream.
aStream crtab: indentation; nextPutAll: ' hasClassScope = '.
hasClassScope printOn: aStream.
aStream crtab: indentation; nextPutAll: ' isAbstract = '.
self isAbstract printOn: aStream.
aStream crtab: indentation; nextPutAll: ' isConstructor = '.
isConstructor printOn: aStream
```

```
printVerboseOn: aStream verbosityLevel: level indentation: inden
super
  printVerboseOn: aStream
  verbosityLevel: level
  indentation: indentation.
level := 0 ifTrue: {self}.
aStream crtab: indentation; nextPutAll: ' belongsToBehaviour = '.
belongsToBehaviour printOn: aStream.
aStream crtab: indentation; nextPutAll: ' belongsToBehaviourRef = '.
belongsToBehaviourRef printOn: aStream
```

This last part displays the source code. To improve the visibility, the lines that match are colored in red.

4.6 The Graphical Viewer

The graphical viewer displays the scenario in the context of the complete application.

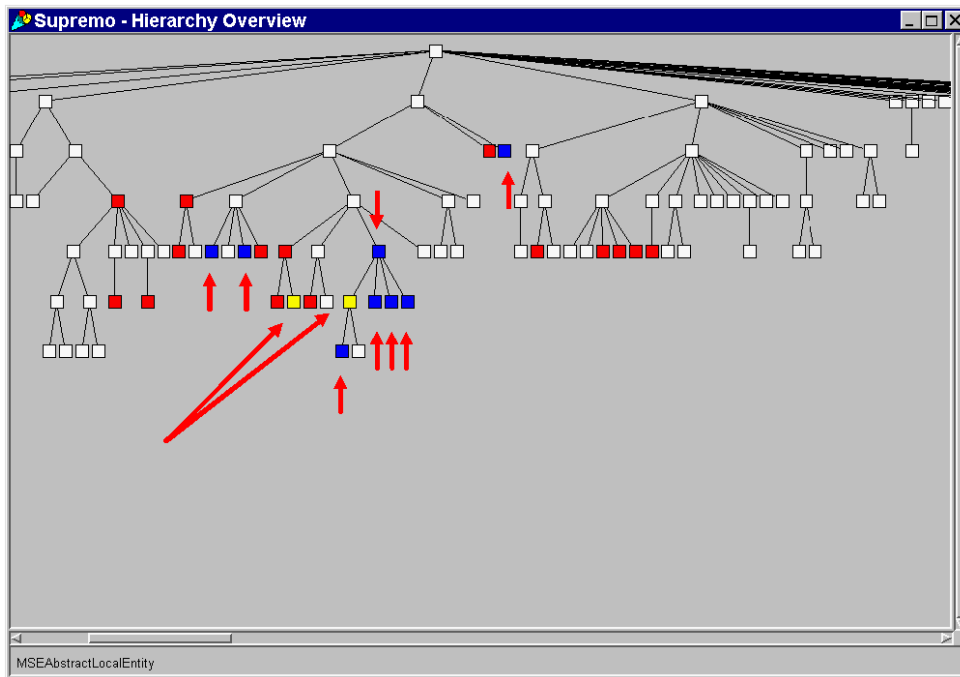


Figure 4.7: The graphical viewer of SUPREMO

The essential idea is that visual representations help make understanding software easier [30]. The graphical viewer is a window that represents the inheritance tree of all classes in the analyzed program.

To emphasize the visualization, the classes are colored. The classes containing clones are colored in red, those implicated with the displayed duplication are blue and the two classes associated in the displayed duplication are yellow.

It is necessary to see the hierarchy where the current displayed duplication is detected. This guides the choice of the refactoring. The representation in Figure 4.7 shows a duplication with a first cousin scenario.

The current duplication has an impact of 10 (see *item # 11*). The two classes pointed by the big arrows are those containing the methods with the current duplication. The eight other classes that are implicated are pointed by the small arrows in the Figure 4.7.

In the presented example, one cannot envisage refactoring the current duplication without considering the involved duplications in the other eight classes. The graphical viewer helps to find out for instance which class is the common ancestor of the all 10 classes. A window at the bottom of the window displays the class name of the class where is positioned the mouse.

Chapter 5

Validation: Statistical Analysis

This chapter presents a statistical analysis of the case studies we have done to validate our approach. To verify the language independence of SUPREMO, we analyzed

- seven applications written in Smalltalk
- one application in C++
- one application in Java

We discuss first the results of the analysis of the applications in Smalltalk, which constitute the biggest part of the validation. In the second part of this chapter we present the results of the C++ and Java case studies.

5.1 Applications in Smalltalk

This section presents

- the analyzed applications
- the noise filtration (see Subsection [4.4.2](#))
- the duplication metrics

5.1.1 Analyzed Applications

In the table [5.1](#), are listed the statistical dates of applications we have studied: number of classes, methods and number of lines of code.

The applications are divided in tree groups:

- The first group is constituted of tree applications developed at the University of Bern, in the SCG group.
Those applications are the most important of the validation. They were chosen because the developers were available and we could discuss with them the qualitative assessment of the identified duplications and relevance of the scenarios.
The applications are:

1. CODECRAWLER [20].
An application that combines metrics and graphs for Object Oriented Reverse Engineering.
2. DUPLOC [11] (see Section 4.3.2).
Detects duplicated code and gives a matrix based representation of the results.
3. MOOSE [24] (see Section 4.3.3).
A framework that extracts structure information of Object Oriented systems.

- The second group is constituted of the VISUALWORKS application framework and the REFACTORINGBROWSER. The good reputation of the two applications incites us to consider them as reference applications.
- The last group contains PDP and MAF, which play the role of real world applications.

Application	Number of Classes	Number of Methods	Lines of Code
CODECRAWLER 2.912	82	1552	9745
DUPLOC 2.14g	269	4768	36526
MOOSE 1.45	254	4592	36566
VISUALWORKS 3.0	883	30689	278347
REFACTORINGBROWSER 3.5	238	5693	39825
PDP 2.6	62	4147	45289
MAF	269	6076	55251

Table 5.1: Number of Classes, Methods, Lines of Code in the Smalltalk applications.

5.1.2 Noise Filtration

In the Subsection 4.4.2 we discuss the reduction of noise in the data coming from the duplication detection tool. The filtration is empirical by essence. We observed the kind of duplications for different values of the density and number of matches (see definition in Paragraph 4.4.2). The distribution of those parameters in the seven applications are presented in the following subsections.

Distribution of Density

Table 5.2 shows the results of the analysis of the seven Smalltalk applications before filtering of noise.

The decision of eliminating the duplication with low density was made from the ob-

Density	CodeCrawler 2.912	Duploc 2.14g	Moose 1.45	VisualWorks 3.0	RefactoringBrowser 3.5	PDPApp 2.6	MAF	Total	Percent
]0 , 0.1]	0	0	0	17	0	0	0	17	0
]0.1 , 0.2]	0	6	4	243	11	1	3	268	3
]0.2 , 0.3]	7	41	15	388	12	2	6	471	6
]0.3 , 0.4]	10	116	25	580	77	4	24	836	11
]0.4 , 0.5]	15	180	64	631	92	10	66	1058	14
]0.5 , 0.6]	14	41	21	213	68	5	23	385	5
]0.6 , 0.7]	30	123	134	658	198	8	68	1219	15
]0.7 , 0.8]	48	95	116	630	220	11	108	1228	15
]0.8 , 0.9]	18	25	25	270	31	4	45	418	5
]0.9 , 1.0]	71	170	330	820	494	26	136	2047	26
Total:	213	797	734	4450	1203	71	479	7947	100

Table 5.2: Distribution of Density before Filtering.

servation of the duplications in each range of density. In the following we presents examples of duplication encountered in each range of density to show why we eliminated them. All the presented example come from CODECRAWLER.

Density between 0.2 and 0.3 This range of the density corresponds to 6 % of all duplications.

Upper Class: **CCHorizontalHistogramLayout**
Upper Method: **layout**

```

layout
| posCollection yCoord xCoord min maxYPosMetric |
maxYPosMetric := self graph maxVerticalMetric.
posCollection := Array new: maxYPosMetric + 1 withAll: 0.
xCoord := position x.
yCoord := position y.
min := CConstants minimumNodeWidth.
nodes
  do:
    [ :each |
      | xPos yPos |
      yPos := each verticalMetric.
      xPos := posCollection at: yPos + 1.
      each layoutAt: xCoord + xPos @ (yCoord + (yPos + 1 * min)).
      posCollection at: yPos + 1 put: (posCollection at: yPos + 1)
        + each width]

```

```

layout
| xCoord1 xCoord2 yCoord1 yCoord2 spacing |
xCoord1 := position x.
xCoord2 := position x + 200.
yCoord1 := position y.
yCoord2 := position y.
spacing := CConstants nodeSpacing.
graph allAttributeNodes
  do:
    [ :each |
      each layoutAt: xCoord2 @ yCoord2.
      yCoord2 := yCoord2 + each height + spacing].
graph allMethodNodes
  do:
    [ :each |
      each layoutAt: xCoord1 @ yCoord1.
      yCoord1 := yCoord1 + each height + spacing]

```

Lower Class: **CCVerticalConfrontationLayout**
Lower Method: **layout**

Figure 5.1: Example of Density between 0.2 and 0.3.

Figure 5.1 shows a duplication with a sibling classes scenario and a length of 10 with three lines that match. The density is 0.3. The two last that match are constituted by keywords of the language (`do: [:each]`). Such keyword appear frequently in the program and give many noise during the duplication detection.

Density between 0.3 and 0.4 11 % of the analyzed duplications have a density between 0.3 and 0.4.

Figure 5.2 a duplication between unrelated classes with a length of 5 and a number

Upper Class: **CCMethodTreeNodePlugin**

Upper Method: **computeLevel:**

```
computeLevel: aLevel
| possibleRecursionCondition overlap reducedChildren |
children isEmpty
  ifTrue: [^self level: aLevel]
  ifFalse:
    [overlap := children asSet intersection: fathers asSet.
     possibleRecursionCondition := overlap notEmpty.
     possibleRecursionCondition
       ifFalse:
         [self level: aLevel.
          children do: [:each | each computeLevel: aLevel + 1].
          ^self].
     self level: aLevel - 1.
     reducedChildren := children.
     overlap do: [:each | reducedChildren remove: each].
     reducedChildren do: [:each | each computeLevel: aLevel + 1].
     ^self]
```

```
changedCheckClasses
| state |
state := chkClasses value.
state
  ifFalse:
    [| state1 |
     state1 := self chkMethods value | self chkFunctions value.
     state1
       ifFalse:
         [self disable: #chkInvocation.
          chkInvocation value: false].
     self disable: #chkInheritance.
     chkInheritance value: false]
  ifTrue:
    [self enable: #chkInheritance.
     self enable: #chkInvocation]
```

Lower Class: **CCGraphSubcanvas**

Lower Method: **changedCheckClasses**

Figure 5.2: Example of Density between 0.3 and 0.4.

of matches of 2. The two lines that match are only composed by a keyword of the programming language. Keywords that match by accident have no signification.

Density between 0.4 and 0.5 This range of density correspond to 14 % of the duplications, the most important that will be eliminated in the work. You only need to have two keywords that match separated by two other lines and you have a duplication with a density of 0.5.

Figure 5.3 illustrate a duplication in common hierarchy. The applications with graph-

Upper Class: **CodeCrawler**
Upper Method: **addSelectSubMenuTo:**

```
addSelectSubMenuTo: mb
mb beginSubMenuLabeled: '&Selection'.
mb add: '&Selection Manager' -> [self openSelectionViewer].
mb line.
mb add: '&Remove Selected Items'
-> [drawing removeSelectedEntitiesFromSelection].
mb line.
mb add: 'Select Highlighted &Nodes' -> [drawing selectHighlightedNodes].
mb add: 'Select Highlighted &Edges' -> [drawing selectHighlightedEdges].
mb line.
mb add: '&Inverse Current Selection' -> [drawing inverseCurrentSelection].
mb endSubMenu.
^mb
```

```
addPluginMenuTo: mb
super addPluginMenuTo: mb.
mb line.
mb add: 'Browse My Class' -> [self browseOwner].
mb add: 'Browse Text' -> [self browseText].
^mb
```

Lower Class: **CCMethodNodePlugin**
Lower Method: **addPluginMenuTo:**

Figure 5.3: Example of Density between 0.4 and 0.5.

ical user interface have many of such methods that handle menus. The same structure is repeated along the method and give a lot of detected duplications that could not be refactored.

Density between 0.5 and 0.6 5 % of the duplication have a density between 0.5 and 0.6.

Upper Class: **CCMethodInvocationTreeLayout**
Upper Method: **layoutAttributes**

```
layoutAttributes
| xCoord1 wid attSpacing hei |
xCoord1 := position x.
wid := self widthOfMethodTree.
hei := self heightOfMethodTree.
attSpacing := wid / (graph allAttributeNodes size + 1).
graph allAttributeNodes
  do:
    [:each |
      each layoutAt: xCoord1 @ (hei + 100).
      each figure fillColor: ColorValue blue.
      xCoord1 := xCoord1 + each width + attSpacing]
```

```
translateAccessors
| hei levelSpacing |
hei := self heightOfMethodTree.
levelSpacing := CCConstants levelSpacing.
graph allAccessorMethodNodes
  do:
    [:each |
      | xTemp |
      xTemp := each figure origin x.
      each figure fillColor: ColorValue red.
      each layoutAt: xTemp @ hei]
```

Lower Class: **CCMethodInvocationTreeLayout**
Lower Method: **translateAccessors**

Figure 5.4: Example of Density between 0.5 and 0.6.

Figure 5.4 shows an example of duplication in the same class. Like in Figure 5.1, we have the same keywords that match and we have a line that have the same method call in both methods: `#layoutAttributes` and `#translateAccessors`. This duplication can be considered noise.

Density between 0.6 and 0.7 This range of density represents 15 % of all duplications in the seven applications.

Figure 5.5 illustrate the fact that even in this high range of density we continue to

Upper Class: **CCAbstractNodePlugin**
Upper Method: **menuAt:**

```
menuAt: aPoint
| mb |
mb := MenuBuilder new.
self addPluginMenuTo: mb.
^mb menu
```

```
menuAt: aPoint
| mb m |
mb := MenuBuilder new. "mb beginSubMenuLabeled: 'Debug'."
m := super menuAt: aPoint.
mb addLabel: 'Do The Evolution! (at your own risk)' value: #doTheEvolution.
m menuItems do: [:each | mb addLabel: each label value: each value].
mb endSubMenu."
^mb menu
```

Lower Class: **CCDrawing**
Lower Method: **menuAt:**

Figure 5.5: Example of Density between 0.6 and 0.7 (Noise).

have noise. But the difference with the duplications of lower density is that, now appear some duplications that give opportunities to refactor (see Figure 5.6).

Upper Class: **CCMaximalBoundsLayout**
 Upper Method: **layout**

```

layout
| xCoord yCoord maxYSize spacing maxRecX xOffset |
xCoord := position x.
yCoord := position y.
spacing := CConstants nodeSpacing.
maxYSize := 0.
xOffset := 0.
maxRecX := graph codeCrawler drawing bounds extent x.
nodes do: [:each | xCoord + each width < maxRecX
  ifTrue:
    [xCoord := xCoord + xOffset.
     each layoutAt: xCoord @ yCoord.
     xOffset := each width + spacing.
     maxYSize := maxYSize max: each height]
  ifFalse:
    [xCoord := position x.
     yCoord := yCoord + maxYSize + spacing.
     each layoutAt: xCoord @ yCoord.
     xOffset := each width + spacing.
     maxYSize := maxYSize max: each height.
     maxYSize := 0]]

```

```

layout
| side xCoord yCoord lineCounter maxYSize xOffset spacing |
side := nodes size sqrt rounded.
xCoord := position x.
yCoord := position y.
spacing := CConstants nodeSpacing.
lineCounter := 0.
maxYSize := 0.
xOffset := 0.
nodes
do:
[:each |
lineCounter <= side
  ifTrue:
    [xCoord := xCoord + xOffset.
     each layoutAt: xCoord @ yCoord.
     xOffset := each width + spacing.
     maxYSize := maxYSize max: each height]
  ifFalse:
    [xCoord := position x.
     maxYSize := maxYSize max: each height.
     yCoord := yCoord + maxYSize + spacing.
     each layoutAt: xCoord @ yCoord.
     xOffset := each width + spacing.
     maxYSize := 0.
     lineCounter := 0].
lineCounter := lineCounter + 1]

```

Lower Class: **CCQuadraticLayout**
 Lower Method: **layout**

Figure 5.6: Example of Density between 0.6 and 0.7 (Refactorable).

The duplication of Figure 5.6 is a good candidate for refactoring. We can imagine extract methods from the first part of the method and from each part of `#ifTrue:` and `#ifFalse:` branch.

This last example explain why we decided to consider only duplications with density

greater than 0.6, which is represented by the dotted line in Figure 5.7. By applying this filtration, we eliminate about 40 % (see Figure 5.2) of all duplications.

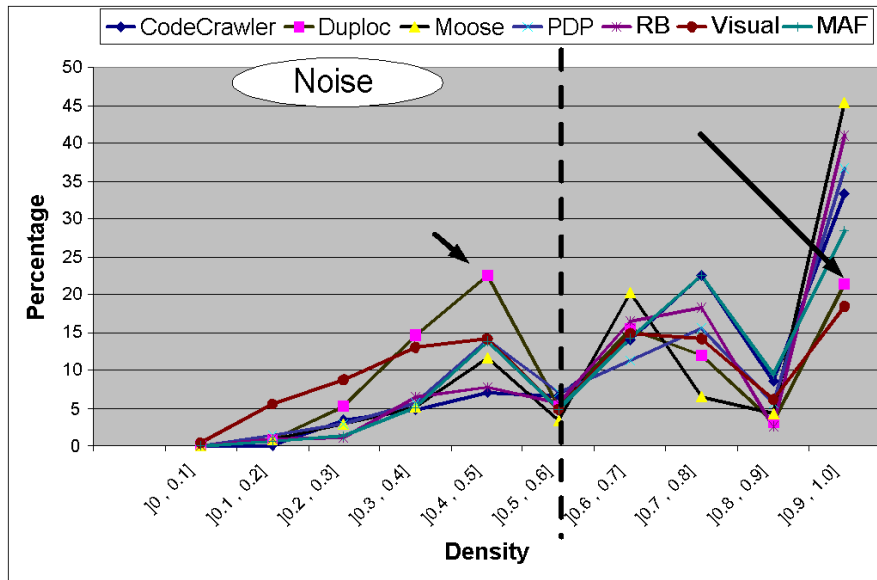


Figure 5.7: Noise reducing: density distribution.

Figure 5.7 is a graphical summary of measured density in the seven Smalltalk applications. On the abscissa, the density of duplication is divided on intervals; on the ordinate, are represented the percentage of duplication which are on the corresponding intervals of density. For example, the coordinate pointed by the small arrow means: about 22 % of the duplications detected in DUPLOC have a density between 0.4 and 0.5. The big arrow shows a point that means: 21 % of the duplications detected in DUPLOC have a density between 0.9 and 1.

It is curious to see the drop in number of detected duplications for the density between 0.5 and 0.6, and for those with density between 0.8 and 0.9. This is probably due to the idioms of Smalltalk and/or to the coding style.

Distribution of Number of Matches

We have seen in the previous chapter that only the density as filtration criterion is not sufficient to eliminate efficiently the noise. Two matches of successive keywords will be detected as a duplication with a density of 1. To improve the filtration, we have combined the density with the number of matches. The distribution of number of matches is presented in Table A.3.5.

We follow the same approach like in the previous chapter and present examples of duplications with different number of matches we encountered during the analysis of the seven applications.

Number of Matches = 3 This category of duplication corresponds to 25 % of all duplications.

This example of duplication in a sibling scenario (Figure 5.8) with number of matches

Upper Class: **CCCircleLayout**
Upper Method: **layout**

```
layout
| angleTemp angle rad center |
rad := (Dialog
  request: 'Radius:'
  initialAnswer: '300'
  onCancel: ['300']) asNumber.
center := Point x: rad y: rad.
angleTemp := 2 * Float pi / nodes size.
angle := 0.
nodes
  do:
    [:each |
      | point |
      point := center + (Point r: rad theta: angle).
      angle := angle + angleTemp.
      each layoutAt: point]
```

```
layout
| xCoord yCoord spacing yCoordBase |
xCoord := position x.
yCoord := position y.
yCoordBase := yCoord.
spacing := CCConstants nodeSpacing.
nodes
  do:
    [:each |
      xCoord := xCoord + each width + spacing.
      yCoord := yCoordBase + (each width * 20).
      each layoutAt: xCoord @ yCoord]
```

Lower Class: **CCHorizontalSequenceLayout**
Lower Method: **layout**

Figure 5.8: Example of Number of Matches = 3.

equals 3 is trivial. We have two matches of keywords applied to a collection which is, in this case, an instance variable.

Number of Matches = 4 14 % of the duplications have a number of matches equals 4.

Upper Class: **CCConfiguration**
Upper Method: **printOn:**

```
printOn: aStream  
super printOn: aStream.  
aStream nextPut: $(.  
graphName printOn: aStream.  
aStream nextPut: $)
```

```
printOn: aStream  
super printOn: aStream.  
aStream nextPut: $(.  
entity printOn: aStream.  
aStream nextPut: $)
```

Lower Class: **CCNode**
Lower Method: **printOn:**

Figure 5.9: Example of Number of Matches = 4.

Figure 5.9 shows a typical example of duplications we found with number of matches equals 4: a method with a supercall. Many observed duplications in this category are also constituted of menu methods for graphical user interfaces (see Figure 5.3).

Upper Class: **CodeCrawler**
 Upper Method: **addSpecialSubMenuTo:**

```

addSpecialSubMenuTo: mb
  mb beginSubMenuLabeled: 'E&xttras'.
  mb beginSubMenuLabeled: '&Grouping...'.
  mb add: '&Collapse Selected Nodes'
    -> [drawing selectedNodes isEmpty
        ifFalse: [drawing collapseSelectedNodes]].
  mb endSubMenu.
  mb beginSubMenuLabeled: '&Spring...'.
  mb add: 'Start Spring &With Nodes' -> [drawing startSpringWithNodes].
  mb add: 'Start Spring With&out Nodes' -> [drawing startSpringWithoutNodes].
  mb line.
  mb add: '&Fix Selected Nodes' -> [drawing fixSelectedNodes].
  mb add: '&Unfix Selected Nodes' -> [drawing unfixSelectedNodes].
  mb line.
  mb add: '&Spring Settings' -> [self openSpringSettingsPanel].
  mb endSubMenu.
  mb line.
  mb line.
  mb beginSubMenuLabeled: 'Print on Transcript...'.
  mb add: 'Displayed Node Names' -> [drawing printDisplayedNodeNames].
  mb add: 'Selected Node Names' -> [drawing printSelectedNodeNames].
  mb add: 'Metrics Table of Selected Nodes'
    -> [drawing printMetricsTableOfSelected].
  mb add: 'Metrics Table of Displayed Nodes'
    -> [drawing printMetricsTableOfDisplayed].
  mb endSubMenu.
  mb endSubMenu.
  ^mb

```

```

addTransformationSubMenuTo: mb
  mb beginSubMenuLabeled: '&Transformation'.
  mb add: 'Raise Nodes' -> [drawing raiseNodes].
  mb add: 'Raise Edges' -> [drawing raiseEdges].
  mb line.
  mb add: 'Zoom by 2' -> [drawing zoomByTwo].
  mb add: 'Zoom by 0.5' -> [drawing zoomByHalf].
  mb add: 'Zoom by ...' -> [drawing zoomByValue].
  mb line.
  mb add: 'Scale by 2' -> [drawing scaleByTwo].
  mb add: 'Scale by 0.5' -> [drawing scaleByHalf].
  mb add: 'Scale by ...' -> [drawing scaleByValue].
  mb line.
  mb add: 'Translate All Items To Origin' -> [drawing translateToOrigin].
  mb add: 'Translate All Items...' -> [drawing translateAllByValue].
  mb add: 'Translate Selected Nodes...' -> [drawing translateSelectedByValue].
  mb endSubMenu.
  ^mb

```

Lower Class: **CodeCrawler**
 Lower Method: **addTransformationSubMenuTo:**

Figure 5.10: Example of Number of Matches = 5 (Noise).

Number of Matches = 5 Duplications with number of matches equals 5, which correspond to 8 % of all duplications constitute the turning point of our observations. It is very difficult to say if we should eliminate those duplications or not. We continue to have noise like in Figure 5.10 but we have also refactorable duplications (see Figure 5.11).

Upper Class: **CCAttributeNodePlugin**
 Upper Method: **browseOwner**

```

browseOwner
| myClassName myClass |
myClassName := node entity belongsToClass.
myClass := MSEUtilities classOrMetaclassNameToClass: myClassName.
HierarchyBrowser newOnClass: myClass

```

```

browseOwner
| myClassName myClass |
myClassName := node entity belongsToClass.
myClass := MSEUtilities classOrMetaclassNameToClass: myClassName.
HierarchyBrowser newOnClass: myClass

```

Lower Class: **CCMethodNodePlugin**
 Lower Method: **browseOwner**

Figure 5.11: Example of Number of Matches = 5 (Refactorable).

We decided however to eliminate duplications with number of matches equals 5 to improve the efficiency of filtration but we know that some duplications have not been seen in the following of our work.

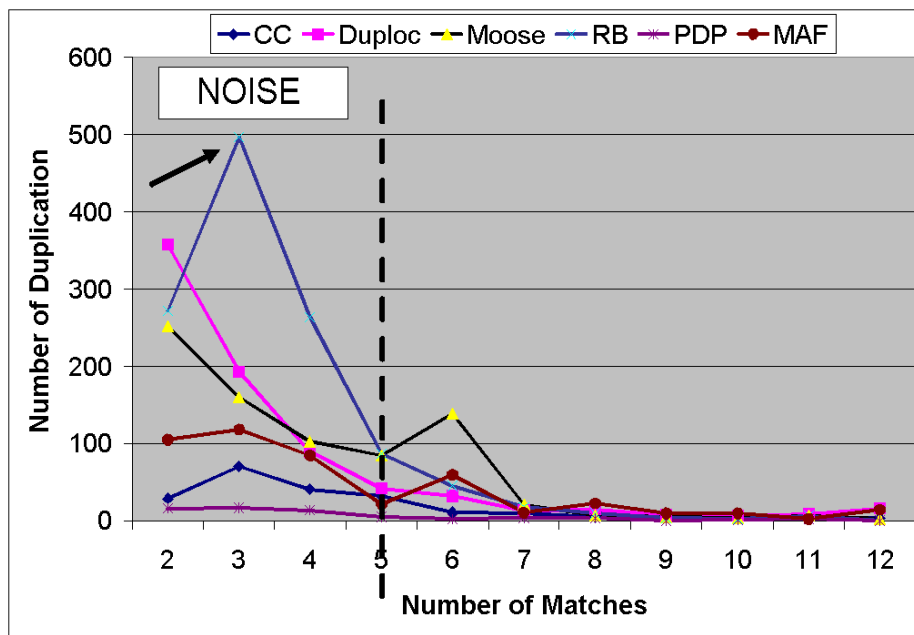


Figure 5.12: Number of Matches Distribution in Analyzed Applications.

Figure 5.12 is a graphical summary of measured number of matches in the seven Smalltalk applications. The number of matches are on the abscissa, and the number of corresponding duplications are represented on the ordinate. For example, the coordinate pointed out by the arrow means: about 500 duplications detected in REFACTORINGBROWSER have a number of matches of 3.

5.1.3 Metrics

After the elimination of the noise, we quantitatively analyzed the seven applications. We computed the following metrics: the scenario, the impact on classes, the number of matches and the length of duplication.

In the following subsections, we present graphically only the results of the three application of the SCG group. For the details of all results see Appendix in Section A.3.2.

Scenario Distribution

The scenario represented most (see Figure 5.13) in all studied applications is the single class scenario, representing a third of all duplications. It is followed by the sibling scenario that represents a fourth of all duplications.

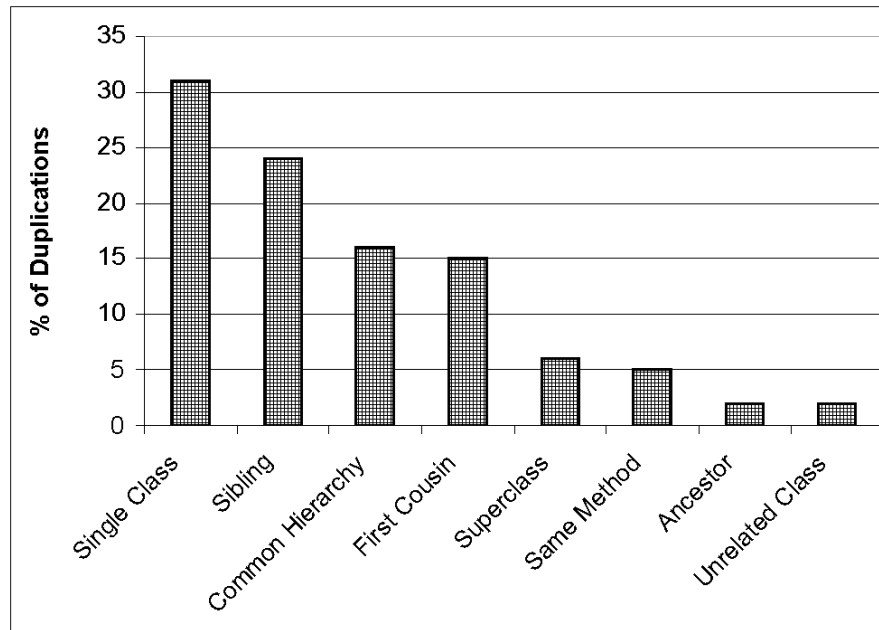


Figure 5.13: Distribution of Scenarios in Analyzed Applications.

We can say that by refactoring only two kinds of scenario, we remove the 50% of the duplications. For the detailed numbers, see the Subsection A.3.1.

5.1.4 Distribution of Impact on Classes

The impact on classes measures how many classes contain a clone of a duplication. The Figure 5.14 shows the distribution of this metrics in the three applications.

MOOSE has the biggest average value of impact on classes: 4.2. The maximum value

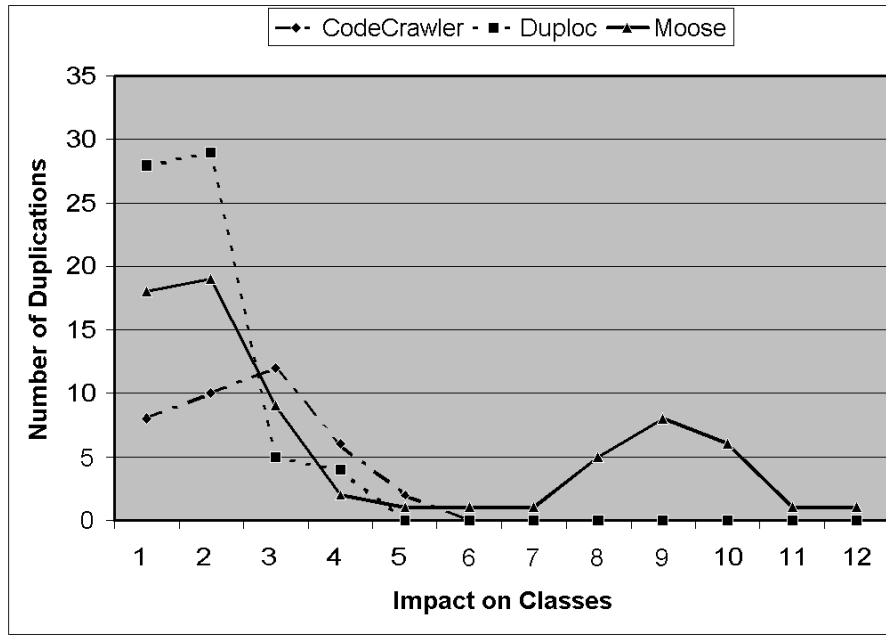


Figure 5.14: Distribution of Impact on Classes in Analyzed Applications.

is 12 (see Table A.9). 75% of impact in MOOSE are smaller than 8 whereas in CODE-CRAWLER 75% are smaller than 3 and for DUPLOC the value is 2. The value from the reference group (VISUALWORKS and REFACTORINGBROWSER) is 2.

5.1.5 Distribution of Number of Matches

In Figure 5.15, the number of matches are on the abscissa, and on the ordinate, are represented the number of duplications corresponding to this number of matches. This is a continuation of Figure 5.12 (after elimination of noise).

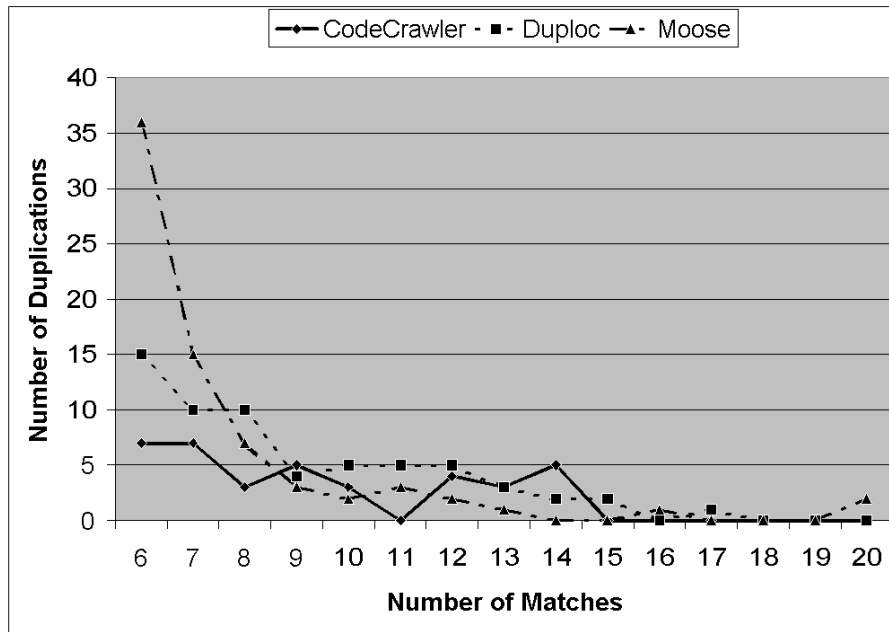


Figure 5.15: Distribution of Number of Matches in Analyzed Application.

75% of the duplication has a number of matches smaller than 8 for MOOSE and 12 for CODECRAWLER and DUPLOC (see Table A.10). The preference's values are 8 for REFACTORINGBROWSER and 11 for VISUALWORKS.

5.1.6 Distribution of Length of Duplication

In the Figure 5.16, the lengths of duplication are on the abscissa, and on the ordinate, are represented the corresponding number of duplications.

75% of the duplication has a length smaller than 10 for MOOSE and 14 for CODE-

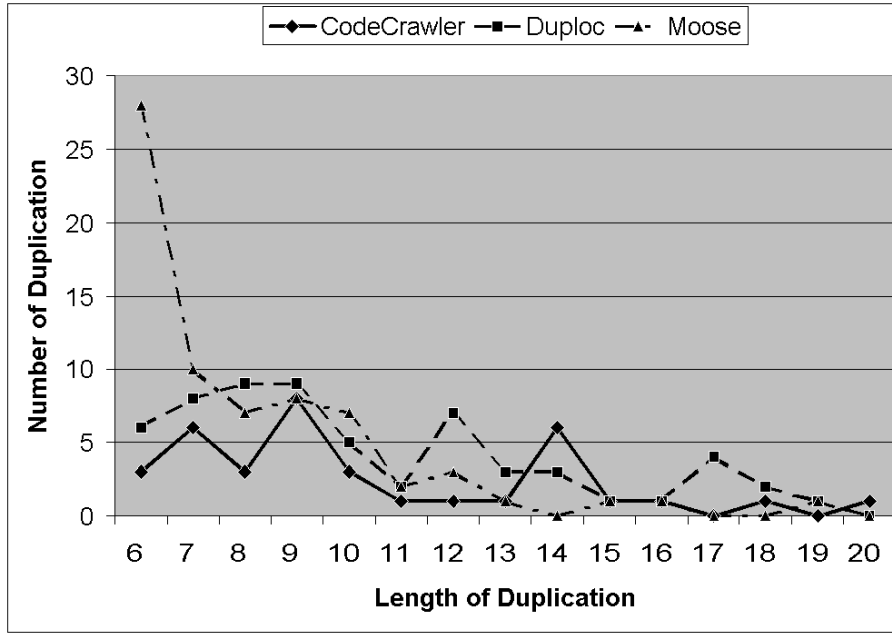


Figure 5.16: Distribution of Length of Duplication in Analyzed Applications.

CRAWLER and DUPLOC (see Table A.11). The references values are 10 for REFACTORIZATIONBROWSER and 13 for VISUALWORKS.

5.1.7 Conclusion

In conclusion, the applications of SCG group contain very few duplications. In comparison with the reference group the values obtained are very good.

5.2 Applications in C++ and Java

This section presents the results of the C++ and Java case studies which are performed to validate the language independence of SUPREMO. Table 5.3 lists the statistical data of the two applications: number of classes, methods and lines of code.

The first application, Jpeg, is a little application written in C++ developed by a student to manipulate image files. The second, written in Java, is a part of the Swing framework which was suggested as good candidate because of its great density of duplications per file.

Application	Language	Number of Classes	Number of Methods	Lines of Code
Jpeg	C++	14	145	2954
Swing	Java	30	387	6101

Table 5.3: Number of Classes, Methods, Lines of Code in the C++ and Java applications.

To compare with the results of Smalltalk applications, we first analyzed the distribution of density (see Figure 5.17).

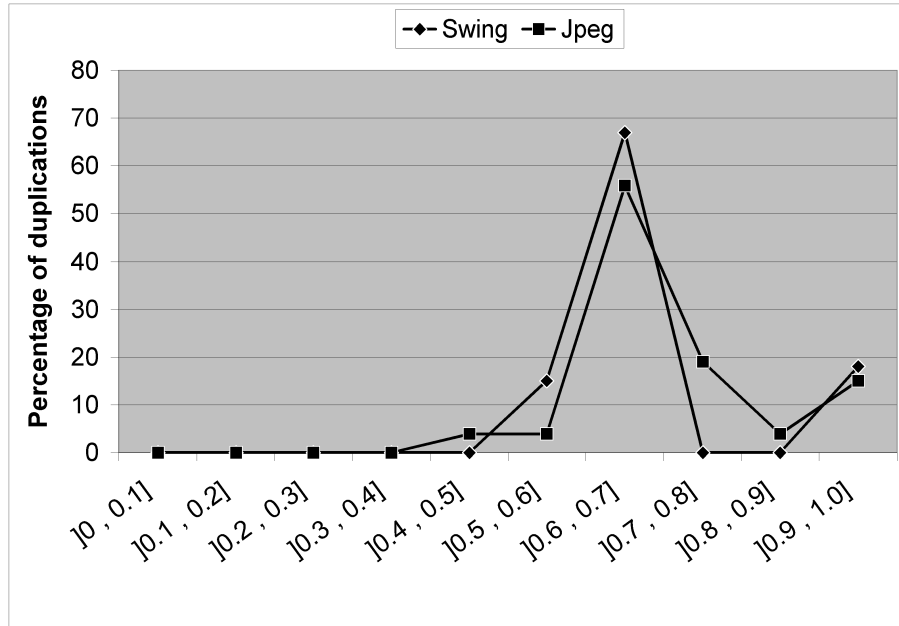


Figure 5.17: Density distribution in the C++ and Java applications.

The two drops in number of detected duplications observed in Figure 5.7 are not anymore present in Figure 5.17.

Figure 5.18 shows the distribution of number of matches in both applications. The two analyzed applications are too small to have enough cases of duplication. The number of classes in both programs does not allow abundance of scenarios. For the Swing application we had only sibling classes scenarios. We must analyze more applications to compare with the results of the previous chapter.

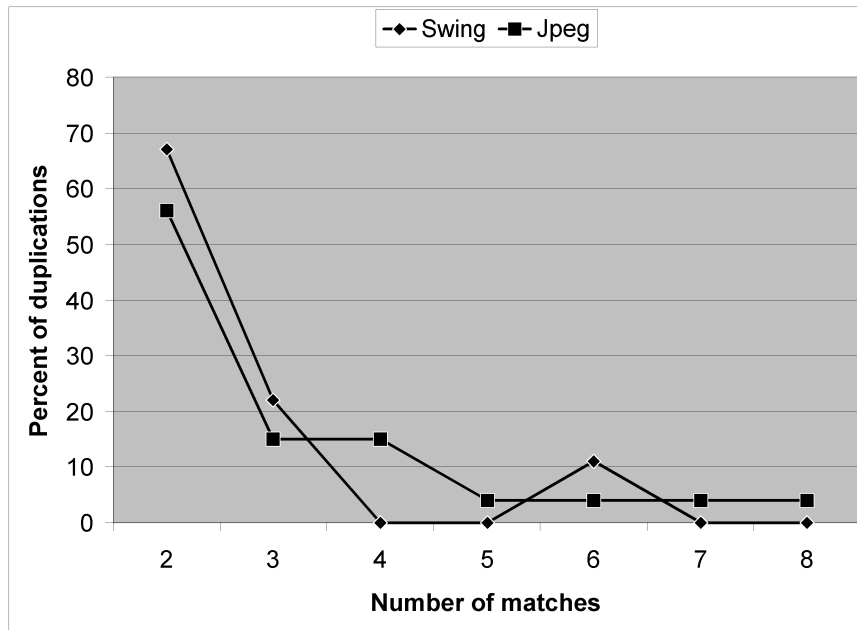


Figure 5.18: Number of Matches Distribution in the C++ and Java applications.

Chapter 6

Validation: Qualitative Aspects

This section describes the experiments we performed with the developers of the SCG applications: CODECRAWLER, DUPLOC, and MOOSE. As we have already noted, those applications were chosen because the developers were available. This constitutes the quality part of the validation. It is important to discuss with the developers for the qualitative assessment of the identified duplications. The process we followed is:

1. We analyzed the applications and presented the results to the developer.
2. He decided for each presented duplication, if it should be refactored or not.
3. We discussed then the kind of possible refactoring that would be applicable and compared with the proposed solution in our approach.
4. When the developer decided not to remove the duplicated code, a discussion of the reasons of this decision were reported (see Section 6.1). We discussed also to evaluate how such duplication could be automatically detected and thus improve SUPREMO.

Presentation of the results. In Chapter 3 we presented the refactorings we can use depending on encountered scenario. For the presentation of the results, we need more qualifiers for two cases:

- the action **Not changed**
represents the case where the developer does not remove the duplication.
- the action **Remove**
represents the case where dead code is found. There is no refactoring to do, we have only to remove the not any more used code.

6.1 Case Studies Results

The next three subsections show in summary the results of our analysis on DUPLOC, CODECRAWLER and MOOSE. We found the following five reasons to explain the presence of duplications:

1. performance issues,

2. readability of the program,
3. bad design,
4. VISUALWORKS specific GUI issues,
5. simulation of multiple inheritance.

6.1.1 Analysis of DUPLOC

	Ancestor	Common Hierarchy	First Cousin	Same Method	Sibling	Single Class	Superclass	Unrelated	Total
Not changed	1	1		4	1	8	2	2	19
Remove					20	4			24
Template + Parameterization						1			1
Extract Method				2		6			8
Insert Method Call						2			2
Parameterization						4			4
Pull Up					1				1
Insert Super Call							1		1
Template + Hook					3		1		4
Extract Method + Pull Up		1	1						2
Total	1	2	1	6	25	25	4	2	66

Table 6.1: Analysis of DUPLOC.

The table 6.1 presents the cases we found in DUPLOC. The developer decided to refactor 71 % of the 66 duplications and to apply nine different refactorings. 36 % of all duplications were dead code that was then removed. Indeed, one class was used in a student experiment, and was not properly subclassed. The developer decided to not remove 29 % of duplications for the following reasons:

- 8 duplications stay for performance issues (see Figure 6.14).
- 9 duplications stay because they increase the readability of the program (see Figure 6.8).
- 2 duplications stay for bad design reasons. The whole class should be re-designed.

6.1.2 Analysis of CODECRAWLER

	Ancestor	Common Hierarchy	First Cousin	Same Method	Sibling	Single Class	Superclass	Unrelated	Total
Not changed	1		10		5	4	3	1	24
Remove		1	1		1	2		1	6
Extract Method						4			4
Parameterization						1			1
Extract Method + Pull Up					3				3
Total	1	1	11	0	9	11	3	2	38

Table 6.2: Analysis of CODECRAWLER.

The table 6.2 presents the cases we found in CODECRAWLER. The developer decided to refactor only 37 % of the 38 duplications and to apply four different refactorings. 16 % of all duplications was dead code.

He decided to not remove 63 % of duplications for the following reasons:

- 1 due to VisualWorks specific GUI issues.
- 9 multiple inheritance simulations (see Figure 6.6).
- 2 false positives.
- 6 for performance issues.
- 6 bad design, must be modified

6.1.3 Analysis of MOOSE

	Ancestor	Common Hierarchy	First Cousin	Same Method	Sibling	Single Class	Superclass	Unrelated	Total
Not changed	1	20	13	2	4	6	1		47
Remove		1				2			3
Template + Hook		4					1		4
Parameterization		1			1	8	2		12
Extract Method						2			2
Pull Up					3				3
Extract Method + Pull Up			1						1
Total	1	26	14	2	8	18	3	0	72

Table 6.3: Analysis of MOOSE.

The table 6.3 presents the cases we found in MOOSE. The developer decided to refactor only 35 % of the 72 duplications and to apply six different refactorings. 4 % of the duplications was dead code.

He decided to not remove 65 % of duplications for the following reasons:

- 2 due to VisualWorks specific GUI issues.
- 1 bad design.
- 9 multiple inheritance simulations.
- 35 super calls (see Figure 6.4).

We continue with a list of scenario based examples of duplication we found during the analysis of the three applications.

6.2 Scenario based Examples

This section shows a list of examples of duplications encountered during the analysis of the SCG applications grouped by scenario. The refactorings the developer decided to apply is also described.

The refactorings presented in those examples are:

- Template + Hook (see for example Section 6.2.2).
- Extract Method + Pull Up (see for example Section 6.2.3).
- Extract Method (see for example Section 6.2.6).
- Pull Up (see for example Section 6.2.5).
- Template + Parameterization (see for example Section 6.2.6).
- Insert Method Call (see for example Section 6.2.4).
- Insert Super Call (see for example Section 6.2.7).

In fact, this section simulates the functioning of SUPREMO. The displayed source code is shown and the available information for the refactoring phase is also presented if necessary.

Recall. In this thesis the duplication is an association of two pieces of code. Each of them is a sequence of lines of code that matches or not with a line in the other piece of code. The lines that matches are represented with a bold font.

To represent the duplication as displayed in the SUPREMO textual viewer, the two compared pieces of source code are shown in two superposed boxes. The class and the method name of the concerned source code are also noted.

6.2.1 Ancestor Scenario

Upper Class: **AbstractRawSubMatrix**

Upper Method: **abstractInitializeAt:**

```
abstractInitializeAt: aRectangle
"returns true if successfull, else it releases itself and returns false"
"Before calling this method assure the access to rawMatrix and call
initializeAttributes."

"Verify argument"
| tmpRegion |
aRectangle isNil
  ifTrue:
    [self release.
     ^false].
self topology rawMatrix size extent < (1 @ 1)
  ifTrue:
    [self release.
     ^false].
tmpRegion := self topology rawMatrix size intersect: aRectangle.
tmpRegion origin > tmpRegion corner
  ifTrue:
    [self release.
     ^false].
region := tmpRegion.
interestInFilteredContents := true.
^true
```

Figure 6.1: Upper part of: Ancestor Scenario.

```

abstractInitializeAsNthOV: aInteger At: aPoint FilteringContents: aBoolean
"Verify, if only one OV is necessary. Therefore only if this is the first,
it will be successfully instantiated."
"returns true if successfull, else false"

"Verify argument"
| tmpRegion |
aPoint isNil
  ifTrue:
    [self release.
     ^false].
aInteger isNil
  ifTrue:
    [self release.
     ^false]. "begin to initialize"
"Prepare the RSM region - this works for both cases - if SOV is
necessary or not."
tmpRegion := Rectangle origin: 1 @ 1 extent: (AbstractOverView RSM-
size: self topology rawMatrix).
(AbstractOverView forTwoLevel: self topology rawMatrix)
  ifTrue: ["Is a SOV needed and if true is the value of aInteger valid?"
          "Now, we have a valid rectangle for the RSM - aPoint is
          ignored, because the RSM must cover the whole RM."
          aInteger ~= 1
          ifTrue:
            [self release. "see comments above"
             ^false]]
  ifFalse: ["A SOV is necessary. Therefore tmpRegion with an already
          valid extent must be positioned at aPoint"
           tmpRegion moveTo: aPoint]. "Make necessary abstract initializations"
^super
abstractInitializeAt: tmpRegion
WithBinLen: (self class RSMbinSize: self topology rawMatrix) x
FilteringContents: aBoolean

```

Lower Class: **AbstractOverView**

Lower Method: **abstractInitializeAsNthOV:At:FilteringContents:**

Figure 6.2: Ancestor Scenario.

Not changed. In the duplication of the Figure 6.2, the upper class **AbstractRawSubMatrix** (Figure 6.1) is the ancestor class. The impact of the duplication is 6 classes. That means there are clones in other classes with different configurations of scenarios. The developer must take in to account all duplications to make a common refactoring. The other involved scenarios are sibling, superclass, and unrelated scenario. After discussion with the developer, we found out that this is an idiom of the programmer! There are different conditions, and the same 'ifTrue:' branch. The conditions could eventually be united with OR. Ultimately, since readability would suffer from the proposed refactoring, the developer decided to leave the duplication unchanged..

6.2.2 Common Hierarchy Scenario

Upper Class: `MSESTParseTreeAnnotator`
 Upper Method: `doMethod:selector:primitive:block:`

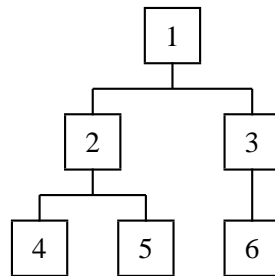
```
doMethod: aNode selector: sel primitive: prim block: block
currentMethod isPureAccessor isNil
  ifTrue:
    [self assessIsAccessorOf: currentMethod basedOn: aNode].
^super
doMethod: aNode
selector: sel
primitive: prim
block: block
```

```
doMethod: aNode selector: sel primitive: prim block: block
self assessIsAccessorOf: currentMethod basedOn: aNode.
^super
doMethod: aNode
selector: sel
primitive: prim
block: block
```

Lower Class: `MSESTParseTreeBuildingEnumerator`
 Lower Method: `doMethod:selector:primitive:block:`

Figure 6.3: Common Hierarchy Scenario.

Template + Hook. We represent below the class structure of the concerned duplication. The upper class `MSESTParseTreeAnnotator` corresponds to the number 5 and the class `MSESTParseTreeBuildingEnumerator` to the number 3.



In each of the classes with number 2, 3, 4, 5, 6 a method is implemented with the same name: `#doMethod:selector:primitive:block:.` The impact is 4 classes. That means the duplicated code is present in 4 classes (those with number 3, 4, 5, 6). The proposed refactoring is to create a template and hook methods in the class with number 1 which is root of the hierarchy.

Upper Class: **MSEMethod**

Upper Method: **printVerboseOn:verbosityLevel:indentation:**

```

printVerboseOn: aStream verbosityLevel: level indentation: indentation
super
  printVerboseOn: aStream
  verbosityLevel: level
  indentation: indentation.
level = 0 ifTrue: [^self].
aStream crtab: indentation; nextPutAll: '  className = '.
belongsToClass printOn: aStream.
aStream crtab: indentation; nextPutAll: '  classRef = '.
belongsToClassRef printOn: aStream.
aStream crtab: indentation; nextPutAll: '  hasClassScope = '.
hasClassScope printOn: aStream.
aStream crtab: indentation; nextPutAll: '  isAbstract = '.
self isAbstract printOn: aStream.
aStream crtab: indentation; nextPutAll: '  isConstructor = '.
isConstructor printOn: aStream

```

```

printVerboseOn: aStream verbosityLevel: level indentation: indentation
super
  printVerboseOn: aStream
  verbosityLevel: level
  indentation: indentation.
level = 0 ifTrue: [^self].
aStream crtab: indentation; nextPutAll: '  Name = '.
name printOn: aStream.
aStream crtab: indentation; nextPutAll: '  Value = '.
value printOn: aStream.
aStream crtab: indentation; nextPutAll: '  belongsToID = '.
belongsToID printOn: aStream

```

Lower Class: **MSEProperty**

Lower Method: **printVerboseOn:verbosityLevel:indentation:**

Figure 6.4: Common Hierarchy Scenario.

Not changed. The duplication of the Figure 6.4 could be considered as a false positive. In fact the duplication concerns only one function that is a super call. The method call was distributed over multiple lines by the automatic code formatter. If it would have been put on one line, the duplication would be filtered out.

As our duplication detection is based on line comparison matching, one method gives three lines of code and was not filtered by DUPLOC.

That is the drawback of the choice of our approach. We wanted to remain simple but simplicity comes at a certain cost.

6.2.3 First Cousin Scenario

Upper Class: **OverView**

Upper Method: **initialize:AsNthOV:OutOf:FilteringContents:ListeningTo:**

```

initialize: aRawMatrix AsNthOV: aIntegerPosition OutOf: aIntegerSize
FilteringContents: aBoolean ListeningTo: aPt
  aRawMatrix isNil
    ifTrue:
      [self release.
       ^nil].
  aPt isNil
    ifTrue:
      [self release.
       ^nil].
  protocolTransformer isNil ifFalse: ["release possible previous dependency"
   protocolTransformer removeDependent: self]. "begin to initialize"
  self initializeAttributes. "setup topology"
  rawMatrix := aRawMatrix.
  protocolTransformer := aPt.
  protocolTransformer addDependent: self.
  (super
   abstractInitializeAsNthOV: aIntegerPosition
   OutOf: aIntegerSize
   FilteringContents: aBoolean)
  ifTrue: [^self].
  self release.
  ^nil

```

```

initialize: aRawMatrix WithFilteredContents: aBoolean ListeningTo: aPt

  aRawMatrix isNil
    ifTrue:
      [self release.
       ^nil].
  aPt isNil
    ifTrue:
      [self release.
       ^nil].
  protocolTransformer isNil ifFalse: ["release possible previous dependency"
   protocolTransformer removeDependent: self]. "begin to initialize"
  self initializeAttributes. "setup topology"
  rawMatrix := aRawMatrix.
  protocolTransformer := aPt.
  protocolTransformer addDependent: self.
  (super abstractInitializeWithFilteredContents: aBoolean)
  ifTrue: [^self].
  self release.
  ^nil

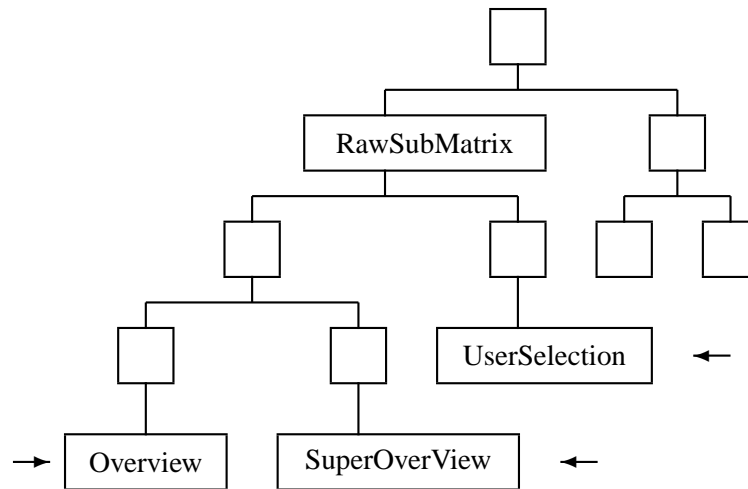
```

Lower Class: **SuperOverView**

Lower Method: **initialize:WithFilteredContents:ListeningTo:**

Figure 6.5: First Cousin Scenario.

Extract Method + Pull Up. There are three methods forming a cluster that must be refactored together. The inheritance hierarchy contains 12 classes and is drawn below:



The arrows point to the three concerned classes. The new extracted method will be pulled up into the class `RawSubMatrix`. Four classes that are between `RawSubMatrix` and the leaf classes inherit a new method they don't need.

Upper Class: **CCClassTreeNodePlugin**
 Upper Method: **computeLevel:**

```
computeLevel: currLevel
"computes the level of a node using a recursive approach. If the level
is the same or greater we need not proceed down"

currLevel > 100
  ifTrue:
    [Dialog warn: 'Severe Problem: Infinite Loop!!! This layout is not
appropriate for the current selection!'.
self halt].
self level >= currLevel ifTrue: [^self].
self level: currLevel.
self isLeaf ifFalse: [children do:
[:child | child computeLevel: currLevel + 1]]
```

```
computeLevel: currLevel
"computes the level of a node using a recursive approach. If the level
is the same or greater we need not proceed down"

currLevel > 100
  ifTrue:
    [Dialog warn: 'Severe Problem: Infinite Loop!!! This layout is not
appropriate for the current selection!'.
self halt].
self level >= currLevel ifTrue: [^self].
self level: currLevel.
self isLeaf ifFalse: [children do:
[:child | child computeLevel: currLevel + 1]]
```

Lower Class: **CCFunctionTreeNodePlugin**
 Lower Method: **computeLevel:**

Figure 6.6: First Cousin Scenario.

end of comment

Not changed. To simulate the multiple inheritance, which is not present in Smalltalk, the developer of CODECRAWLER makes many copies to approach this behaviour.

6.2.4 Same Method Scenario

Upper Class: **BatchJob**

Upper Method: **executeJobProcessingBuckets:logFile:**

```
executeJobProcessingBuckets: aBucketRange logFile: aLogFile
    "executes the job"

    logFile := aLogFile.
    self createComparisonCounter.
    jobID setBucketRange: aBucketRange.
    self isExecutable iffFalse: [^self reportFailureToLogFile].
    bjParameters codeReader configureForLanguage: self sourceLanguage.
    self openReportFile.
    self reportStartInLogFile.
    comparisonCounter openProgressMeter.
    "process the entire matrix that is defined by the axes sections"
    self
        processHAxis: self symmetricAxis
        vAxis: self symmetricAxis
        forBuckets: aBucketRange.
    self
        processHAxis: self horizontalAxis
        vAxis: self symmetricAxis
        forBuckets: aBucketRange.
    self
        processHAxis: self symmetricAxis
        vAxis: self verticalAxis
        forBuckets: aBucketRange.
    self
        processHAxis: self horizontalAxis
        vAxis: self verticalAxis
        forBuckets: aBucketRange.
    comparisonCounter closeProgressMeter.
    self closeReportFile.
    self reportEndInLogFile
```

Figure 6.7: Upper part of: Same Method Scenario.

```

executeJobProcessingBuckets: aBucketRange logFile: aLogFile
"executes the job"

logFile := aLogFile.
self createComparisonCounter.
jobID setBucketRange: aBucketRange.
self isExecutable ifFalse: [^self reportFailureToLogFile].
bjParameters codeReader configureForLanguage: self sourceLanguage.
self openReportFile.
self reportStartInLogFile.
comparisonCounter openProgressMeter.
"process the entire matrix that is defined by the axes sections"
self
    processHAxis: self symmetricAxis
    vAxis: self symmetricAxis
    forBuckets: aBucketRange.
self
    processHAxis: self horizontalAxis
    vAxis: self symmetricAxis
    forBuckets: aBucketRange.
self
    processHAxis: self symmetricAxis
    vAxis: self verticalAxis
    forBuckets: aBucketRange.
self
    processHAxis: self horizontalAxis
    vAxis: self verticalAxis
    forBuckets: aBucketRange.
comparisonCounter closeProgressMeter.
self closeReportFile.
self reportEndInLogFile

```

Lower Class: **BatchJob**

Lower Method: **executeJobProcessingBuckets:logFile:**

Figure 6.8: Same Method Scenario.

Not changed. Theoretically, a loop would be possible, but then the complexity of the data structure to hold the input to the different loop executions would be much more complex than the simplicity gained by the loop-abstraction. The code in its current form is also quite readable, and since there are only four iterations in this loop, it's bearable

6.2.5 Sibling Classes Scenario

Upper Class: `PMCSnormalMode`

Upper Method: `moveCursor:`

```

moveCursor: aPosition
    "this method changes the Cursor into a cross
    if the cursor is over the diagram in the view
    and into a bulls eye if the cursor is over a dot."

    | aMatrixPosition |
    aMatrixPosition := self viewState viewToMatrix: aPosition.
    (self viewState overMatrixArea: aPosition)
    ifTrue:
        [self controller cursorOverDiagram
         ifFalse: [self cursorEntersDiagram].
         self view cursorPosition
         changeCursorposColIndex: aMatrixPosition x
         rowIndex: aMatrixPosition y.
         self view cursorPosition booleanMatchValue
         ifTrue: [Cursor bull show]
         ifFalse: [Cursor crossHair show]]
    ifFalse: [self controller cursorOverDiagram
             ifTrue: [self cursorLeftDiagram]].
    self configureMenu

```

```

moveCursor: aPosition
    "this method changes the Cursor into a cross
    if the cursor is over the diagram in the view
    and into a bulls eye if the cursor is over a dot."

    | aMatrixPosition |
    aMatrixPosition := self viewState viewToMatrix: aPosition.
    (self viewState overMatrixArea: aPosition)
    ifTrue:
        [self controller cursorOverDiagram
         ifFalse: [self cursorEntersDiagram].
         self view cursorPosition
         changeCursorposColIndex: aMatrixPosition x
         rowIndex: aMatrixPosition y.
         self view cursorPosition booleanMatchValue
         ifTrue: [Cursor bull show]
         ifFalse: [Cursor crossHair show]]
    ifFalse: [self controller cursorOverDiagram
             ifTrue: [self cursorLeftDiagram]].
    self configureMenu

```

Lower Class: `PMCSpositioningMode`

Lower Method: `moveCursor:`

Figure 6.9: Sibling Class Scenario.

Pull Up. The Figure 6.9 presents a total copy-and-paste duplication without any change. The scenario is a sibling classes scenario and the solution is easy: a simple Pull Up refactoring. In this case however, the upper method is also duplicated with a method of a other sibling class (see next page).

Upper Class: **PMCSnormalMode**

Upper Method: **moveCursor:**

```

moveCursor: aPosition
"this method changes the Cursor into a cross
if the cursor is over the diagram in the view
and into a bulls eye if the cursor is over a dot."

| aMatrixPosition |
aMatrixPosition := self viewState viewToMatrix: aPosition.
(self viewState overMatrixArea: aPosition)
  ifTrue:
    [self controller cursorOverDiagram
     ifFalse: [self cursorEntersDiagram].
     self view cursorPosition
     changeCursorposColIndex: aMatrixPosition x
     rowIndex: aMatrixPosition y.
     self view cursorPosition booleanMatchValue
     ifTrue: [Cursor bull show]
     ifFalse: [Cursor crossHair show]]
  ifFalse: [self controller cursorOverDiagram
           ifTrue: [self cursorLeftDiagram]].
self configureMenu

```

```

moveCursor: aPosition
"this method changes the Cursor into a hand
if the cursor is over the diagram in the view.
If the movement is engaged, then the underlying object is moved."

| aMatrixPosition |
(self viewState overMatrixArea: aPosition)
  ifTrue:
    [self controller cursorOverDiagram
     ifFalse: [self cursorEntersDiagram].
     "update cursor with new position"
     aMatrixPosition := self viewState viewToMatrix: aPosition.
     self view cursorPosition
     changeCursorposColIndex: aMatrixPosition x
     rowIndex: aMatrixPosition y.
     self view cursorPosition booleanMatchValue
     ifTrue: [Cursor bull show]
     ifFalse: [Cursor crossHair show].
     spying & (prevPosition ~= self view cursorPosition position)
     ifTrue:
       [if spying, then invalidate view
        prevPosition := self view cursorPosition position copy.
        ^self view invalidate]]
     ifFalse: [self controller cursorOverDiagram
             ifTrue: [self cursorLeftDiagram]]

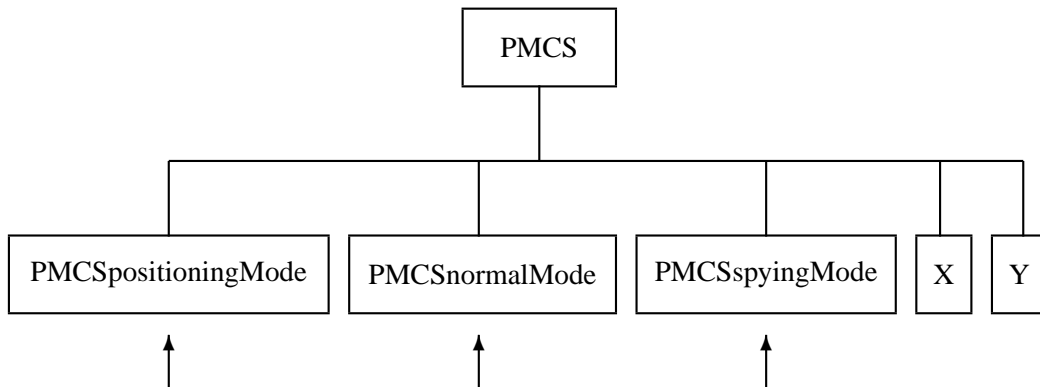
```

Lower Class: **PMCSspyingMode**

Lower Method: **moveCursor:**

Figure 6.10: Sibling Class Scenario.

Extract Method + Pull Up. The upper method is the same as the upper method of the precedent example. The new, third class in the cluster is also a direct sibling of the two others.



A check in the hierarchic view tell us that there are 5 sibling classes and all of them implement the method `#moveCursor` but only three contain the duplicated code. Our proposition is to extract the copied code into a new method and to pull it up in the superclass (`#PMCS`).

This example is good to describe the need of some functionalities that would be useful in SUPREMO. For example, we have the number of sibling classes and we know in which class we found the duplication. But we can not know if the other siblings (X and Y) also implements a method with a given name. We used the REFACTORING-BROWSER to find out this information.

The next version of SUPREMO should provide utilities like senders of method, implementors of methods, ...

6.2.6 Same Class Scenario

Upper Class: `CodeParticle`

Upper Method: `mergeUsingOr`:

```
mergeUsingOr: aCodeParticle
(self safetyCheckWith: aCodeParticle)
  ifTrue:
    [| start end trueVal falseVal newCFun |
     start := lowerBound min: aCodeParticle lowerBound.
     end := upperBound max: aCodeParticle upperBound.
     trueVal := self nonHoleSymbol.
     falseVal := self holeSymbol.
     newCFun := self createEmptyCharactFun.
     start to: end do: [:pos | (self charactFunAt: pos) = trueVal
      | ((aCodeParticle charactFunAt: pos) = trueVal)
        ifTrue: [newCFun addLast: trueVal]
        ifFalse: [newCFun addLast: falseVal]].
     charactFun := newCFun.
     lowerBound := start.
     upperBound := end]
  ifFalse: [self sourceObject = aCodeParticle sourceObject
    & self isEmpty ifTrue: [self copyFrom: aCodeParticle]]
```

```
mergeUsingAnd: aCodeParticle
(self safetyCheckWith: aCodeParticle)
  ifTrue: [(self boundariesOverlapWith: aCodeParticle)
    ifTrue:
      [| start end trueVal falseVal newCFun |
       start := lowerBound max: aCodeParticle lowerBound.
       end := upperBound min: aCodeParticle upperBound.
       trueVal := self nonHoleSymbol.
       falseVal := self holeSymbol.
       newCFun := self createEmptyCharactFun.
       start to: end do: [:pos | (self charactFunAt: pos) = trueVal
        & ((aCodeParticle charactFunAt: pos) = trueVal)
          ifTrue: [newCFun addLast: trueVal]
          ifFalse: [newCFun addLast: falseVal]].
       charactFun := newCFun.
       lowerBound := start.
       upperBound := end]]
  ifFalse: [self sourceObject = aCodeParticle sourceObject
    & aCodeParticle isEmpty ifTrue: [self makeEmptyParticle]]
```

Lower Class: `CodeParticle`

Lower Method: `mergeUsingAnd`:

Figure 6.11: Same Class Scenario.

Template + Parameterization. We have the same algorithm, once for an AND merge, once for an OR merge. Common parts can go into the template methods, and differences can be extracted into parameterized methods. Since we are in the same class, we cannot use polymorphism.

Upper Class: **BinValueColorerView**

Upper Method: **verticalAxeFrom:To:On:**

```
verticalAxeFrom: aRelStartPosition To: aRelEndPosition
On: aGraphicsContext
"draw a vertical axe line with: an open arrow at 10%
after the end point a 2.5% tail and the start and
a big horizontal marker at the start and at the end."
| tmpStart tmpEnd |
tmpStart := (self absolutePosition: aRelStartPosition)
+ (self percentYtranslation: self twoHalfPercent).
tmpEnd := (self absolutePosition: aRelEndPosition)
- (self percentYtranslation: self tenPercent). "draw axe line"
self
displayLineFrom: tmpStart
To: tmpEnd
On: aGraphicsContext. "add open arrow"
self
absVerticalUpArrow: tmpEnd
closed: false
withTail: false
on: aGraphicsContext. "add markers"
self bigHorizontalMarker: (self absolutePosition: aRelStartPosition)
On: aGraphicsContext.
self bigHorizontalMarker: (self absolutePosition: aRelEndPosition)
On: aGraphicsContext
```

```
horizontalReferenceLineFrom: aRelStartPosition To: aRelEndPosition
On: aGraphicsContext
"draw a horizontal reference line with:
a 5% tail at the start and the end."
| tmpStart tmpEnd |
tmpStart := (self absolutePosition: aRelStartPosition)
- (self percentXtranslation: self fivePercent).
tmpEnd := (self absolutePosition: aRelEndPosition)
+ (self percentXtranslation: self fivePercent). "draw line"
self
displayLineFrom: tmpStart
To: tmpEnd
On: aGraphicsContext
```

Lower Class: **BinValueColorerView**

Lower Method: **horizontalReferenceLineFrom:To:On:**

Figure 6.12: Same Class Scenario.

Extract Method. The entire copied sequence can be extracted as a new method with three parameters. To achieve that, it is necessary to change two other methods (`#percentXtranslation:`, `#percentYtranslation:`) by combining them and giving them an additional parameter (`percent: aFloat translation: aDirection`). There is a cluster of four methods participating in this cloning. The other two cluster-participants involved in this match are `#verticalReferenceLineFrom:To:On:` and `#horizontalAxeFrom:To:On:`.

Upper Class: **LineByLineReader**
 Upper Method: **getRecordUntilEmptyLine**

```

getRecordUntilEmptyLine
"returns a collection of all lines up to the occurrence
of an empty line (not returning this line if removeRecordDelimiter
is set to true) or it returns all the lines until the end of the input
if no empty line is found"

| lineColl found line |
lineColl := OrderedCollection new.
readStream isNil ifTrue: [^lineColl].
line := ''.
found := false.
[line isNil | found]
  whileFalse: [(line := self getNextLineFromBuffer) isNil
  ifFalse:
    [line = '' ifTrue: [found := true].
    found not | removeRecordDelimiter not
      ifTrue: [lineColl addLast: line]].
  ^lineColl
  
```

```

getRecordDelimitedBy: aStringOrNil
"returns a collection of all lines up to the occurrence
of a line prefixed by aString (including this line)
or it returns all the lines until the end of the input
if the
is nil"

| lineColl found line |
lineColl := OrderedCollection new.
readStream isNil ifTrue: [^lineColl].
line := ''.
found := false.
[line isNil | found]
  whileFalse: [(line := self getNextLineFromBuffer) isNil
  ifFalse:
    [aStringOrNil notNil
      ifTrue: [found :=
        (line indexOfSubCollection: aStringOrNil
          startingAt: 1) == 1].
      aStringOrNil isNil | (found not | removeRecordDelimiter not)
        ifTrue: [lineColl addLast: line]].
  ^lineColl
  
```

Lower Class: **LineByLineReader**
 Lower Method: **getRecordDelimitedBy:**

Figure 6.13: Same Class Scenario.

Insert Method Call. The upper method could call the lower method with a special parameter value. The lower method would not have to be changed at all.

Upper Class: **FastSparseMatrix**

Upper Method: **numOfMatchesWithoutDiagonal:**

```

numOfMatchesWithoutDiagonal: aBoolean

| sum xLink yLink xIndex yIndex noCount
leftDeletedIndices rightDeletedIndices |
matrix isEmpty ifTrue: [^0].
sum := 0.
noCount := symmetricMatrix & aBoolean.
leftDeletedIndices := self leftDeletedIndices.
rightDeletedIndices := self rightDeletedIndices.
xLink := matrix first.
yLink := nil.
[xLink notNil]
whileTrue:
  [xIndex := xLink index.
  (rightDeletedIndices at: xIndex) isZero
  ifFalse:
    [yLink := xLink value first.
    [yLink notNil]
    whileTrue:
      [yIndex := yLink index.
      (leftDeletedIndices at: yIndex) isZero
      | (xIndex = yIndex & noCount)
      ifFalse: [sum := sum + 1].
      yLink := yLink nextLink].
      xLink := xLink nextLink]].
  ^sum

```

```

forAllDotsPerform: aSymbol with: aDiagramView

| leftDeletedIndices rightDeletedIndices xLink yLink |
matrix isEmpty ifTrue: [^self].
leftDeletedIndices := self leftDeletedIndices.
rightDeletedIndices := self rightDeletedIndices.
xLink := matrix first.
[xLink notNil]
whileTrue:
  [(rightDeletedIndices at: xLink index) isZero
  ifFalse:
    [yLink := xLink value first.
    [yLink notNil]
    whileTrue:
      [(leftDeletedIndices at: yLink index) isZero
      ifFalse:
        [actElValue := yLink value.
        actElIndexX := rightDeletedIndices at: xLink index.
        actElIndexY := leftDeletedIndices at: yLink index.
        aDiagramView perform: aSymbol with: self].
        yLink := yLink nextLink]].
    xLink := xLink nextLink]

```

Lower Class: **FastSparseMatrix**

Lower Method: **forAllDotsPerform:with:**

Figure 6.14: Same Class Scenario.

Not changed. This duplication is included in a cluster of five clones and is not removed because of performance issues. The first part of the duplication concerns the storing of an often used object in a local variable to have a fast access (method must be

fast!). The second part of the code shows a structure to iterate over the data structure. The duplicated code could be abstracted into an iterator if performance would not suffer. However, it is not clear by quickly browsing over the code, if the redesigned system (iteration abstracted into iterators) would have significantly less code than the current solution. However, to find that out, the experiment should be carried out low-level, i.e. the code should really be refactored.

This would also help to find out, if the support of Supremo is enough, can be significantly improved, or if our approach is missing semantic information about the code.

6.2.7 Superclass Scenario

Upper Class: **PMVSOVerViewNormalMode**

Upper Method: **displayObjectsOn:**

```
displayObjectsOn: aGraphicsContext
self
  displayOn: aGraphicsContext
  ContainedObject: self model userSelection
  WithColor: ColorValue orange
  WithRegionCoding: true
  Movable: true
```

```
displayObjectsOn: aGraphicsContext
self
  displayOn: aGraphicsContext
  ContainedObject: self model userSelection
  WithColor: ColorValue orange
  WithRegionCoding: true
  Movable: false.
self
  displayOn: aGraphicsContext
  ContainedObject: self model overview
  WithColor: ColorValue blue
  WithRegionCoding: true
  Movable: true
```

Lower Class: **PMVSSuperOverViewNormalMode**

Lower Method: **displayObjectsOn:**

Figure 6.15: Superclass Scenario.

Insert Super Call. The upper method is the same as the lower one, only in the superclass. The subclass method could replace half of its code by a call to the superclass method. There might be an additional parameter needed, or this could be solved by implementing polymorphic hooks.

6.2.8 Unrelated Classes Scenario

Upper Class: **DuplocSmalltalkBrowser**
Upper Method: **requestForWindowClose**

```
requestForWindowClose
^super requestForWindowClose ifFalse: [false]
  ifTrue:
    ["remove the reference in the main application model"
     self changed: #browserCloses.
     true]
```

```
requestForWindowClose
^super requestForWindowClose ifFalse: [false]
  ifTrue:
    ["remove the reference in the main application model"
     self changed: #browserCloses.
     true]
```

Lower Class: **DuplocFileBrowser**
Lower Method: **requestForWindowClose**

Figure 6.16: Unrelated Class Scenario.

Not changed. With a lot of knowledge about Duploc, one sees the possibility to create an abstract superclass. In this case, it would be a 'DuplocTool' class. This class would define certain relationships between tools and the main Duploc window. For a person that is not familiar with the application, it would not be possible to infer this conclusion from just this single copied method. However, by looking at the hierarchy view, one can see that the two classes in question do not have a superclass in the Duploc hierarchy. By looking at the code, one can see that they are subclasses of Application-Model and of FileBrowser. I.e., to really make a common superclass, one would have to give up the superclass dependency to Filebrowser and write a DuplocFilebrowser from scratch.

6.2.9 Summary

The Table 6.4 contain a summary of the examples we presented in this section. For each scenario and refactoring, we gave at least an example of code found in the SCG applications.

	Ancestor	Common Hierarchy	First Cousin	Same Method	Sibling	Single Class	Superclass	Unrelated
Not changed	✓	✓	✓	✓		✓		✓
Template + Hook		✓						
Extract Method + Pull Up			✓		✓			
Pull Up					✓			
Extract Method						✓		
Template + Parameterization						✓		
Insert Method Call						✓		
Insert Super Call							✓	

Table 6.4: Examples of duplications found in SCG applications.

Chapter 7

Conclusions and Perspectives

7.1 Conclusions

Code duplication is one of the factors that severely complicates the maintenance and evolution of large software systems. In this work, we presented a simple scenario based approach to analyze, categorize and remove duplications. The scenario is defined as the relationship between classes containing methods where the duplications were found.

Our approach is based strongly on the concept of *human in the loop*, i.e. the balance between automation of, and human involvement in the process of reengineering duplication is inclined to the side of the human.

We implemented a tool named SUPREMO to validate our approach. It is characterized by the following aspects:

- The tool analyzes the instances of detected duplications, clusters them and presents these clusters to the user.
- The visualization of the source code in a textual viewer where a pop-up menu gives the user the opportunity to refactor.
- The visualization of the scenario in a graphical global context gives the developer the possibility to see the impact of the duplication.

We analyzed seven applications written in Smalltalk, one written in Java and one, in C++. We presented a statistical analysis and discussed the qualitative aspect of three applications developed in the SCG group. The qualitative validation was illustrated with a list of examples that simulate the functioning of SUPREMO.

As a result of our experiments we claim that scenarios provide useful information for the reengineer and help to guide her in the process of removing duplication. We believe that the refactorings, necessary to remove duplication, is very complex. It will not easily be possible to completely automate the process and thus the support for the human reengineer is the important focus of further research in the area.

7.2 Perspectives and Future Work

7.2.1 Limits of the Approach

Human involvement is costly and slow. An approach that is based on this premise is thus immediately limited to smaller systems, if one envisions reengineering an entire system at once. However, if duplication removal is integrated into the software development process, problems like a high number of duplicates that need attention can be ameliorated.

7.2.2 Future Work

When investigating the possible refactorings, we started from an existing catalogue of known refactorings [3]. We, however, have seen during our discussions with the developers, that the range of refactorings needed in a certain situation is much larger than this catalogue.

- We need to conduct further controlled experiments in order to broaden our knowledge of refactorings applicable for a certain scenario.

The work presented is based on investigations of programs written mainly in Smalltalk. We think that the range of applicable refactorings in a certain scenario is also constrained by the programming language.

- We need to conduct experiments with programming languages other than Smalltalk to an increasing degree.

In investigating duplication present in a system, we have focussed on the granularity of methods. We believe, that further analysis is possible if one looks at classes instead of methods.

- We need to combine the information gathered about duplicated methods to start an analysis of duplicated classes.

Improvements of the Tool As noted before, we believe that duplication removal should be integrated into the normal software development process in the same way as, for example, refactorings in general are part of the Extreme Programming [28] software development process. In order to support this, SUPREMO should be integrated into the normal development environment, like the Smalltalk Refactoring Browser [8] is integrated into the Smalltalk IDE.

- It should be possible to use IDE features like senders and implementors (cross reference tools in Smalltalk)
- It should be possible to execute refactorings directly in the textual viewer of SUPREMO.
- SUPREMO should be capable to incrementally adapt its duplication database to the changes effectuated by refactorings that the reengineer performs.

Appendix A

Appendix

A.1 Refactorings

Refactoring must be done systematically to avoid or reduce the risk of introducing bugs on the working code. Martin Fowler wrote a catalog of 72 refactorings [3]. This Section is an extract of that catalog. We list in this appendix those that are important for eliminating duplication. Each refactoring describes the motivation and mechanics of the code transformation.

A.1.1 Extract Method

If a code fragment can be grouped together, turn it into a method whose name explains the purpose of the method and replace the fragment with a call to the new method.

Mechanics:

1. Create a new method, and name it after the intention of the method (name it by what it does, not by how it does it).
2. Copy the extracted code from the source method into the new target method.
3. Scan the extracted code for references to any variables that are local in scope to the source method. These are local variables and parameters to the method.
4. See whether any temporary variables are used only within this extracted code. If so, declare them in the target method as temporary variables.
5. Look to see whether any of these local-scope variables are modified by the extracted code. If one variable is modified, see whether you can treat the extracted code as a query and assign the result to the variable concerned. If this is awkward, or if there is more than one such variable, you can't extract the method as it stands.
6. Pass into the target method as parameters local-scope variables that are read from the extracted code.

7. Compile when you have dealt with all the locally-scoped variables.
8. Replace the extracted code in the source method with a call to the target method.
9. Compile and test.

A.1.2 Pull Up Method

You have methods with duplicated code on subclasses.

You can eliminate the duplication by extracting method from both classes and then by putting it into a upper class in hierarchy.

Often *Pull Up Method* comes after other steps. You see two methods in different classes that can be parameterized in such a way that they end up as essentially the same method. A special case of the need for *Pull Up Method* occurs when you have a subclass that overrides a superclass method yet does the same thing.

The most awkward element of *Pull Up Method* is that the body of the methods may refer to features that are on the subclass but not on the superclass. If the feature is a method, you can create an abstract method in the superclass.

Mechanics:

1. Inspect the methods to ensure they are identical.
2. Create a new method in the superclass, copy the body of one of the methods to it, adjust, and compile.
3. Delete one subclass method.
4. Compile and test.
5. Keep deleting subclass methods and testing until only the superclass method remains.
6. Take a look at the callers of this method to see whether you can change a required type to the superclass.

A.1.3 Push Down Method

Behavior on a superclass is relevant only for the subclass. Push Down Method is the opposite of *Pull Up Method* (see Section A.1.2).

Mechanics:

1. Declare the method in the subclasses and copy the body.
2. Remove method from superclass.
3. Compile and test.

A.1.4 Form Template Method

You have two methods in subclasses that seem to carry out broadly similar steps in the same sequence, but the steps are not the same.

Move the sequence to the superclass and allow polymorphism to play its role in ensuring the different steps do their things differently. This kind of method is called a template method [21].

Mechanics:

1. Decompose the methods so that all the extracted methods are either identical or completely different.
2. Use *Pull Up Method* (see Section A.1.2) to pull the identical methods into the superclass.
3. For the different methods use *Rename Method* (see Section A.1.8) so the signatures for all the methods at each step are the same.
This makes the original methods the same in that they all issue the same set of method calls, but the subclasses handle the calls differently.
4. Compile and test after each signature change.
5. Use *Pull Up Method* on one of the original methods. Define the signatures of the different methods as abstract methods on the superclass.
6. Compile and test.
7. Remove the other methods, compile, and test after each removal.

A.1.5 Parameterize Method

Several methods do similar things but with different values contained in the method body.

We can create one method that uses a parameter for the different values.

Mechanics:

1. Create a parameterized method that can be substituted for each repetitive method.
2. Compile.
3. Replace one old method with a call to the new method.
4. Compile and test.
5. Repeat for all the methods, testing after each one.

A.1.6 Collapse Hierarchy

Refactoring the hierarchy often involves pushing methods and fields up and down the hierarchy. After you have done this, you can find you have a subclass that is not adding any value, so you need to merge the classes together.

Mechanics:

1. Choose which class is going to be removed: the superclass or the subclasses.
2. Use *Pull Up Method* (see Section [A.1.2](#)) or *Push Down Method* (see Section [A.1.3](#)) to move all the behavior of the removed class to the class with which it is being merged.
3. Compile and test with each move.
4. Adjust references to the class that will be removed to use the merged class. This will affect variable declarations, parameter types and constructors.
5. Remove the empty class.
6. Compile and test.

A.1.7 Extract Superclass

You have two classes with similar features.

Create a superclass and move the common features to the superclass.

Mechanics:

1. Create a blank abstract superclass; make the original classes subclasses of this superclass.
2. One by one, use *Pull Up Field* (see Section [A.1.11](#)), *Pull Up Method* (see Section [A.1.2](#)) and *Pull Up Constructor Body* (see Section [A.1.13](#)) to move common elements to the superclass.
3. Compile and test after each pull.
4. Examine the methods left on the subclasses. See if there are common parts, if there are you can use *Extract Method* (see Section [A.1.1](#)) followed by *Pull Up Method* on the common parts. If the overall flow is similar, you may be able to use *Form Template Method* (see Section [A.1.4](#)).
5. After pulling up all the common elements, check each client of the subclasses. If they use only the common interface you can change the required type to the superclass.

A.1.8 Rename Method

This refactoring is not directly related with the duplication removal but it is used after an extraction in order to name the newly extracted method.

Methods should be named in a way that communicates their intention. A good way to do this is to think what the comment for the method would be and turn that comment into the name of the method.

Mechanics:

1. Find a name for the new method you extract.
2. Check to see whether the method signature is implemented by a superclass or subclass. If it is, find an other name.⁸
3. Declare a new method with the new name. Copy the old body of code over to the new name and make any alterations to fit.
4. Compile
5. Change the body of the old method so that a call to the new created one replaces the extracted code.
6. Compile and test.

A.1.9 Replace Subclass with Field

You have subclasses that vary only in methods that return constant data. Change the methods to superclass fields and eliminate the subclasses.

Mechanics:

1. Use *Replace Constructor with Factory Method* (see Section A.1.12) on the subclasses.
2. If any code refers to the subclasses, replace the reference with one to the superclass.
3. Declare final fields for each constant method on the superclass.
4. Declare a protected superclass constructor to initialize the field.
5. Add or modify subclass constructors to call the new superclass constructor.
6. Compile and test.
7. Implement each constant method in the superclass to return the field and remove the method from the subclass.
8. Compile and test after each removal.
9. When all the subclass methods have been removed, use *Inline Method* (see Section A.1.14) to inline the constructor into the factory method of the superclass.
10. Compile and test.
11. Remove the subclass.
12. Compile and test.
13. Repeat inlining the constructor and eliminating each subclass until they are all gone.

A.1.10 Substitute Algorithm

You want to replace an algorithm with one that is clearer. Replace the body of the method with the new algorithm.

Mechanics:

1. Prepare your alternative algorithm. Get it so that it compiles.
2. Run the new algorithm against your tests. If the results are the same, you are finished.
3. If the results are not the same, use the old algorithm for comparison in testing and debugging. Run each test case with old and new algorithms and watch both results. That will help you see which test cases are causing trouble, and how.

A.1.11 Pull Up Field

Two subclasses have the same field.
Move the field to the superclass.

Mechanics:

1. Inspect all uses of the candidate fields to ensure they are used in the same way.
2. If the fields do not have the same name, rename the fields so that they have the name you want to use for the superclass field.
3. Compile and test.
4. Create a new field in the superclass.
5. Delete the subclass fields.
6. Compile and test.
7. Consider using *Self Encapsulate Field* (see Section [A.1.15](#)) on the field.

A.1.12 Replace Constructor with Factory Method

You want to do more than simple construction when you create an object.
Replace the constructor with a factory method.

Mechanics:

1. Create a factory method. Make its body a call to the current constructor.
2. Replace all calls to the constructor with calls to the factory method.
3. Compile and test after each replacement.
4. Declare the constructor private.
5. Compile.

A.1.13 Pull Up Constructor Body

You have constructors on subclasses with mostly identical bodies.
Create a superclass constructor; call this from the subclass methods.

Mechanics:

1. Define a superclass constructor.
2. Move the common code at the beginning from the subclass to the superclass constructor.
3. Call the superclass constructor as the first step in the subclass constructor.
4. Compile and test.

A.1.14 Inline Method

A method's body is just as clear as its name.

Put the method's body into the body of its callers and remove the method.

Mechanics:

1. Check that the method is not polymorphic.
2. Find all calls to the method.
3. Replace each call with the method body.
4. Compile and test.
5. Remove the method definition.

A.1.15 Self Encapsulate Field

You are accessing a field directly, but the coupling to the field is becoming awkward.

Create getting and setting methods for the field and use only those to access the field.

Mechanics:

1. Create a getting and setting method for the field.
2. Find all references to the field and replace them with a getting or setting method.
3. Make the field private.
4. Double check that you have caught all references.
5. Compile and test.

A.2 Structure of the Smalltalk Applications

A.2.1 SCG Group

MOOSE

Application	Number of Classes	Number of Methods	Lines of Code
MooseAbstractBase 1.21	5	60	491
MooseCDIFReader 1.24	4	77	675
MooseDocSupport 1.2	3	25	165
MooseEnvyImporter 1.10	2	93	579
MooseExternalOperators 1.5	2	11	67
MooseFinder 1.13	12	164	1771
MooseGroups 1.7	4	23	75
MooseImporters 1.40	9	244	1667
MooseLoaderUI 1.14	4	68	534
MooseMetrics 1.29	12	257	1790
MooseModel 1.76	33	900	7356
MooseOperators 1.33	12	99	777
MooseOperatorsUI 1.2	2	18	185
MooseParseTree 1.13	6	104	902
MooseQueries 1.24	16	184	979
MooseStorage 1.25	13	160	2319
MooseUUIDGenerator 1.5	3	82	629
MooseWidgets 1.0	3	32	238
MooseExplorerApp 2.7u	1	4	25
MEX_AuxiliaryRelationships 0.6d	46	272	2100
MEX_AuxiliaryStructures 0.2b	6	109	455
MEX_FunctionProfiles 0.2b	8	102	574
MEX_Functions 0.4r	18	671	7905
MEX_Interfaces 2.2u	5	178	1590
MEX_InterfaceSupport 2.4n	35	969	6893
Total	264	4906	40741

Table A.1: Number of Classes, Methods, Lines of Code in MOOSE

CODECRAWLER

Application	Number of Classes	Number of Methods	Lines of Code
CodeCrawlerApp 2.912	12	448	2139
CCAlgorithms 1.9	4	45	240
CCExtensions 2.001	3	162	1637
CCFamixPlugins 1.1	8	108	413
CCGraphComposer 1.014	5	106	1383
CCHotDraw 2.058	15	261	957
CCLayouts 1.022	18	71	495
CCPlayground 1.2	2	7	45
CCWidgets 2.017	7	88	857
Total	74	1296	8166

Table A.2: Number of Classes, Methods, Lines of Code in CODECRAWLER

DUPLOC

Application	Number of Classes	Number of Methods	Lines of Code
Duploc_API 0.4	2	23	96
Duploc_Base 2.14g	3	94	579
Duploc_ComparisonMatrix 2.14l	20	315	3072
Duploc_ComparisonProcess 1.5	9	89	820
Duploc_Experiments 2.14b	4	7	42
Duploc_MatchPatterns 1.2	9	122	1109
Duploc_MatrixAnalysis 0.4	5	52	308
Duploc_NotesSupport 0.3a	5	31	181
Duploc_RawMatrix 2.14h	13	415	3600
Duploc_Report 2.14g	19	341	2517
Duploc_Sandbox 0.1	2	2	7
Duploc_SystemExtensions 2.14c	9	66	365
Duploc_SystemInfo 0.2	5	216	1616
Duploc_Uilities 2.14f	19	377	3610
Duploc_GUI 2.14k	19	680	5891
Duploc_GUISupport 1.1	18	180	1034
Duploc_SourceInterface 2.14k	48	844	5955
Duploc_InformationMural 2.14c	60	919	5763
Total	269	4773	36565

Table A.3: Number of Classes, Methods, Lines of Code in DUPLOC

A.2.2 Reference Group

REFACTORINGBROWSER

Application	Number of Classes	Number of Methods	Lines of Code
RBaseUIApp R3.5.0	7	283	1926
RBaseUIVWApp R3.5.0	9	287	1962
RChangeObjectsApp R3.5.0	22	246	1093
RChangeObjectsVWApp R3.5.0	5	95	427
REnvironmentsApp R3.5.0	11	245	1290
REnvironmentsVWApp R3.5.0	2	67	298
RRefactoringUI R3.5.0	9	228	1218
RRefactoringVWUI R3.5.0	6	115	737
RBrowserUIApp R3.5.0+PDP2.5x	30	920	7606
RBrowserUIVWApp R3.5.0	27	913	8068
RBrowserUIVW30App R3.5.0	3	217	1883
RParserApp R3.5.1	41	959	5256
RRefactoringsApp R3.5.1	59	968	6910
RRefactoringsVWApp R3.5.0	3	102	881
RRefactoringsVW30App R3.5.0	4	53	309
Total	238	5698	39864

Table A.4: Number of Classes, Methods, Lines of Code in REFACTORINGBROWSER

VISUALWORKS

ENVY Developer 4.00

Application	Number of Classes	Number of Methods	Lines of Code
ColorEditing 1.0.0	11	386	4807
Compilation R4.00 + PDP-2.5	63	1781	17310
EmClassDevelopment R4.00 + PDP-2.5	19	1419	14762
EmClassDevelopment80 R4.00	18	1235	11621
EmImageSupport R4.00	15	1152	12712
EmImageSupport80 R4.00	16	1435	14794
EmLibraryInterface R4.00 + PDP-2.5	17	1189	13610
EmLibraryAccess R4.00	5	171	177
EmLibraryAccess80 R4.00	6	93	177
EmLibraryInterface80 R4.00 + PDP-2.5	13	1123	12730
EmLibraryObjects R4.00	6	630	7464
EmLibraryObjects80 R4.00	7	630	7464
EmLibrarySchema R4.00	34	753	71
EmLibrarySchema80 R4.00	10	411	1231
DragAndDrop R4.00	17	460	2901
EmMethodDecoding R4.00	7	280	2770
EmMethodDecoding80 R4.00	25	1170	7869
EtConfigurationManagement80 R4.00	5	436	5515
EtDevelopment80 R4.00 + PDP-2.5	14	905	15623
EtTools80 R4.00 + PDP-2.5	19	956	9213
EtWindowSystemExtensions R4.00	23	486	4689
ExternalIPC R4.00	6	99	666
FileSystem R4.00	29	574	4914
FileSystemTools R4.00	4	238	2357
ExternalIPC R4.00	6	99	666
FileSystem R4.00	29	574	4914
Events R4.00	98	2181	15052
Graphics R4.00 + PDP-2.5	112	2863	24459

The table continues on the next page...

Application	Number of Classes	Number of Methods	Lines of Code
UIBuilderSpecifications R4.00	45	1423	10822
UIBuilderSupport R4.00	14	448	3238
UIInterfaceBuilding R4.00	57	2044	15284
VisualWorksEnhancements R4.00 + PDP-2.5	10	528	4322
UIBasicComponents R4.00 + PDP-2.5	45	1054	7093
UIBasicSupport R4.00	38	688	4319
UIModels R4.00	12	199	1531
Printing R4.00	14	458	3859
Sockets R4.00	5	145	1461
SUnit 2.6.3	5	90	355
SUnitSCG 1.0b	2	22	196
UnixIPC R4.00	4	51	512
JBWidgets R1.08	2	29	213
JBWidgetsVisualWorksSpecific R1.06	8	237	2221
JBWidgetsVisualWorksSpecific30 R1.00	1	3	16
WindowSystemCompatibility R4.00	21	284	2261
Total	917	31432	278241

Table A.5: Number of Classes, Methods, Lines of Code in VISUALWORKS (continued)

A.2.3 Industrial Group

PDP

Application	Number of Classes	Number of Methods	Lines of Code
PDPAppInspector 2.6	14	833	7830
PDPAppTools 2.5	14	1160	17527
PDPAppMiscSupport 2.5	31	1847	16231
PDPAppRB 2.6+RB3.5.1	3	311	3736
Total	62	4151	45324

Table A.6: Number of Classes, Methods, Lines of Code in PDP

MAF

Application	Number of Classes	Number of Methods	Lines of Code
MAF_Button 1.0	2	3	23
MAF_ComboBox 1.5	11	93	814
MAF_CompositePart 1.2	5	8	81
MAF_InputField 1.14	4	77	915
MAF_List_DisplayDefinition 1.11	12	204	1678
MAF_List_Model 1.15	10	207	1637
MAF_MenuButton 1.3	3	23	287
MAF_Notebook 1.17	1	3	23
MAF_DynamicSubCanvas_Widget 1.2	4	244	2288
MAF_Notebook_Model 1.8	4	133	1116
MAF_Notebook_Widget 1.11	12	443	4446
MAF_PassiveLabel 1.1	2	3	23
MAF_ProgressBar 1.0	4	207	2163
MAF_Resizer 1.5	10	262	2704
MAF_TextEditor 1.7	3	23	194
MAF_Window 1.2	2	86	544
MAF_Basics 1.39	15	417	4254
MAF_Commands 1.18	21	514	4433
MAF_Labels 1.9	12	302	2177
MAF_Menus 1.12	12	180	1375
MAF_Models 1.1	2	12	104
MAF_OnlineHelp 1.3	9	59	530
MAF_ToolBar 1.11	9	259	2469
MAF_Tooltips 1.3	2	23	222
MAF_Win4Look 1.32	30	364	3391
MAF_CompositeComponents 2.0	3	193	1971
MAF_DateSelector 1.4	4	337	3366
MAF_SelectionEditor 1.17	15	403	3789
MAF_SpinBox 2.0	18	652	5937
MAF_TaskBasedEditor 1.12	10	86	797
MAF_DragAndDrop 1.2	2	13	71
MAF_EnablingAndDisabling 1.7	15	243	1429
Total	269	6076	55251

Table A.7: Number of Classes, Methods, Lines of Code in MAF

A.3 Validation: Statistical Analysis

A.3.1 Distribution of Scenarios

	CodeCrawler 2.912	Duploc 2.14g	Moose 1.45	VisualWorks 3.0	RefactoringBrowser 3.5	PDPApp 2.6	MAF
Scenarios							
Ancestor Scenario	1	1	1	18	2	0	1
Common Hierarchy Scenario	1	2	26	15	6	0	0
First Cousin Scenario	11	1	14	17	2	0	4
Same Class Scenario	11	25	18	202	26	7	33
Same Method Scenario	0	6	2	24	3	0	2
Sibling Scenario	9	26	8	27	17	0	3
Superclass Scenario	3	4	3	42	4	6	5
Unrelated Class Scenario	2	1	0	44	2	0	24
Total	38	66	72	389	62	72	13
Result in Percent							
Ancestor Scenario	3	2	1	5	3	0	1
Common Hierarchy Scenario	3	3	36	4	10	0	0
First Cousin Scenario	29	2	19	4	3	0	6
Same Class Scenario	29	38	25	52	42	54	46
Same Method Scenario	0	9	3	6	5	0	3
Sibling Scenario	24	39	11	7	27	0	4
Superclass Scenario	8	6	4	11	6	46	7
Unrelated Class Scenario	5	2	0	11	3	0	33

Table A.8: Number of Duplications in Analyzed applications.

A.3.2 Impact on Classes

	CodeCrawler 2.912	Duploc 2.14g	Moose 1.45	VisualWorks 3.0	RefactoringBrowser 3.5	PDPApp 2.6	MAF
Impact On Classes							
1	8	28	18	187	27	4	34
2	10	29	19	134	22	9	13
3	12	5	9	39	8	0	14
4	6	4	2	25	3	0	9
5	2	0	1	3	2	0	2
6	0	0	1	0	0	0	0
7	0	0	1	0	0	0	0
8	0	0	5	0	0	0	0
9	0	0	8	0	0	0	0
10	0	0	6	0	0	0	0
11	0	0	1	0	0	0	0
12	0	0	1	0	0	0	0
Average	2.6	1.8	4.2	1.8	1.9	1.7	2.1
Maximum	5	4	12	2	5	5	5
75%-Quantil	3	2	8	2	2	2	3

Table A.9: Impact On Classes.

A.3.3 Number of Matches

	CodeCrawler 2.912	Duploc 2.14g	Moose 1.45	VisualWorks 3.0	RefactoringBrowser 3.5	PDPApp 2.6	MAF
6	7	15	36	101	31	0	23
7	7	10	15	57	14	2	7
8	3	10	7	59	7	3	10
9	5	4	3	37	2	0	2
10	3	5	2	26	1	1	6
11	0	5	3	17	6	0	2
12	4	5	2	16	0	0	8
13	3	3	1	10	0	2	0
14	5	2	0	13	0	1	1
15	0	2	0	7	0	2	2
16	0	0	1	5	0	0	1
17	0	1	0	4	0	0	0
18	0	0	0	7	1	0	4
19	0	0	0	3	0	1	1
20	0	0	2	1	0	0	2
21	0	1	0	2	0	0	0
22	0	0	0	3	0	0	0
23	0	1	0	3	0	0	0
24	0	0	0	4	0	0	2
25	0	0	0	1	0	0	0
26	0	0	0	1	0	1	1
27	1	0	0	1	0	0	0
28	0	0	0	5	0	0	0
30	0	1	0	2	0	0	0
32	0	1	0	1	0	0	0
40	0	0	0	1	0	0	0
41	0	0	0	1	0	0	0
45	0	0	0	1	0	0	0
Average	9.8	10.0	7.6	10.0	7.3	12.5	10.1
Standard deviation	4.0	5.1	2.9	5.7	2.1	5.3	5.0
Maximum	27	32	20	45	18	26	26
75%-Quantil	12	12	8	11	8	15	12

Table A.10: Number of Matches.

A.3.4 Length

Length	CodeCrawler 2.912	Duploc 2.14g	Moose 1.45	VisualWorks 3.0	RefactoringBrowser 3.5	PDPApp 2.6	MAF
6	3	6	28	35	10	0	13
7	6	8	10	42	13	2	4
8	3	9	7	74	19	2	9
9	8	9	8	49	4	1	12
10	3	5	7	33	3	0	4
11	1	2	2	30	8	0	4
12	1	7	3	16	1	0	5
13	1	3	1	18	0	1	2
14	6	3	0	9	1	0	0
15	1	1	1	7	0	0	2
16	1	1	1	8	1	1	1
17	0	4	0	8	1	2	1
18	1	2	0	3	1	0	2
19	0	1	1	12	0	1	3
20	1	0	0	7	0	1	1
21	1	0	0	2	0	0	1
22	0	1	1	1	0	0	2
23	0	2	0	3	0	1	1
24	0	0	1	5	0	0	2
25	0	0	0	4	0	0	2
26	0	0	0	2	0	0	0
27	0	0	0	1	0	0	1
28	0	0	0	3	0	0	0
29	1	0	0	1	0	0	0
30	0	0	1	0	0	0	0

continue on the next page...

	CodeCrawler 2.912	Duploc 2.14g	Moose 1.45	VisualWorks 3.0	RefactoringBrowser 3.5	PDPApp 2.6	MAF
31	0	1	0	3	0	1	0
32	0	0	0	1	0	0	0
33	0	0	0	2	0	0	0
34	0	0	0	0	0	0	0
35	0	1	0	0	0	0	0
36	0	0	0	1	0	0	0
37	0	0	0	1	0	0	0
38	0	0	0	0	0	0	0
39	0	0	0	1	0	0	0
40	0	0	0	2	0	0	0
41	0	0	0	1	0	0	0
42	0	0	0	0	0	0	0
43	0	0	0	0	0	0	0
44	0	0	0	1	0	0	0
45	0	0	0	0	0	0	0
46	0	0	0	1	0	0	0
47	0	0	0	0	0	0	0
48	0	0	0	0	0	0	0
49	0	0	0	0	0	0	0
50	0	0	0	1	0	0	0
51	0	0	0	0	0	0	0
52	0	0	0	0	0	0	0
53	0	0	0	0	0	0	0
54	0	0	0	0	0	0	0
55	0	0	0	0	0	0	0
56	0	0	0	0	0	0	0
57	0	0	0	1	0	0	0
Average	11	12	9	12	9	15	12
Standard deviation	5	6	4	7	3	7	6
Maximum	29	35	30	57	18	31	27
75%-Quantil	14	14	10	13	10	19	15

Table A.11: Length of Duplications.

A.3.5 Number of Matches before Filtering

Nb of Matches	CodeCrawler 2.912	Duploc 2.14g	Moose 1.45	VisualWorks 3.0	RefactoringBrowser 3.5	PDPApp 2.6	MAF	Total	Percent
2	29	357	199	1745	272	16	105	2723	34
3	70	193	160	948	496	17	118	2002	25
4	40	89	102	519	263	13	84	1110	14
5	32	42	84	354	87	5	22	626	8
6	11	32	138	275	45	2	59	562	7
7	9	13	20	138	18	3	11	212	3
8	5	14	7	125	10	3	23	187	2
9	7	7	5	83	3	0	10	115	1
10	4	6	3	59	2	1	10	85	1
11	3	8	6	32	6	2	2	59	1
12	3	15	2	24	0	0	14	58	1
13	0	7	1	25	0	4	0	37	0
14	0	3	0	30	0	1	3	37	0
15	0	3	1	13	0	2	3	22	0
16	0	0	2	11	0	0	1	14	0
17	0	1	0	12	0	0	0	13	0
18	0	2	0	9	1	0	7	19	0
19	0	0	0	6	0	1	1	8	0
20	0	0	2	3	0	0	2	7	0
21	0	2	0	2	0	0	0	4	0
22	0	0	0	4	0	0	0	4	0
23	0	1	0	5	0	0	1	7	0
24	0	0	2	5	0	0	2	9	0
25	0	0	0	1	0	0	0	1	0
26	0	0	0	2	0	1	1	4	0
27	0	0	0	2	0	0	0	2	0
28	0	0	0	6	0	0	0	6	0
29	0	0	0	3	0	0	0	3	0
30	0	1	0	1	0	0	0	2	0
31	0	0	0	2	0	0	0	2	0
32	0	1	0	1	0	0	0	2	0
35	0	0	0	1	0	0	0	1	0
40	0	0	0	1	0	0	0	1	0
41	0	0	0	1	0	0	0	1	0
45	0	0	0	1	0	0	0	1	0
48	0	0	0	1	0	0	0	1	0
Total:	213	797	734	4450	1203	71	479	7947	100

Table A.12: Distribution of Number of Matches before Filtering.

Bibliography

- [1] Brenda S. Baker.
A program for Identifying Duplicated Code.
Computing Science and Statistics, 24:49-57, 1992.
- [2] Brenda S. Baker.
Parameterized Duplication in Strings: Algorithms and an Application to Software Maintenance.
SIAM Journal of Computing, October 1997.
- [3] Martin Fowler.
Refactoring. Improving the Design of Existing Code.
Booch Jacobson Rumbaugh, 1999.
- [4] W.Opdyke.
Refactorings Object-Oriented Frameworks.
PhD Thesis, University of Illinois, 1992.
- [5] R. Johnson and W. Opdyke.
Refactoring and Aggregation.
Object Technologies for Advanced Software, LNCS, vol. 742, Springer-Verlag, 1993.
- [6] W.Opdyke and R.Johnson .
Creating Abstract Superclasses by Refactoring.
Proceedings of the 1993, ACM Conference on Computer Science, ACM Press, 1993, pages 66-73.
- [7] D. Robert, J. Brant and R. Johnson.
Using the Refactoring Browser Will it increase your efficiency ?.
The Smalltalk Report, Vol 6, Sep, 1997.
- [8] D.Roberts, J. Brant and R. Johnson.
A Refactoring Tool for Smalltalk.
Theory and Practice of Object Systems special Issue on Software Reengineering, 1998.
- [9] J.M. Brant.
HotDraw.
Master's thesis, University of Illinois, (pp22, 28), 1995.

- [10] Brenda S. Baker.
On Finding Duplication and Near-Duplication in Large-Software-Systems.
In Proceedings Second Working Conference on Reverse Engineering, pages 86-95, IEEE Computer Society, 1995.
- [11] S. Ducasse, M. Rieger and S. Demeyer.
A Language independent approach for detecting duplicated code.
In Proceedings of the International Conference on Software Maintenance, pages 109-118, 1999.
- [12] J. H. Johnson.
Identifying redundancy in source code using fingerprints.
CASCON'93, pages 171-183, October 1993.
- [13] K. Kontogiannis.
Evaluation experiments on the detection of programming pattern using software metrics.
Proceedings of the 4th Working Conference on Reverse Engineering, Ira Baxter and Alex Quilici and Chris Verhoef, IEEE Computer Society, pages 44-54, 1997.
- [14] J. Mayrand, C. Leblanc and E. Merlo.
Experiment on the automatic detection of function clones in a software system using metrics.
In Proceedings of the International Conference on Software Maintenance, 1996.
- [15] M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, and K. Kontogiannis.
Measuring clone based reengineering opportunities.
In International Symposium on Software metrics. METRICS'99. IEEE Computer Society Press, November 1999.
- [16] M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, and K. Kontogiannis.
Partial redesign of Java software systems based on clone analysis.
In Proceedings of the 6th Working Conference on Reverse Engineering, pages 326-336. IEEE Computer Society Press, October 1999.
- [17] M. Balazinska.
Reconception de systèmes orientés-object basée sur l'analyse des clones.
Mémoire de diplôme ès Sciences Appliquées, Novembre 1999.
- [18] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna and L. Bier.
Clone detection using abstract syntax trees.
In Proceedings of the International Conference on Software Maintenance, pages 368-377. IEEE Computer Society Press, 1998.
- [19] S. A. Alpert, K. Brown, B. Woolf.
The Design Patterns - Smalltalk Companion.
Addison Wesley, 1998.

-
- [20] Michele Lanza.
Combining Metrics and Graphs for Object Oriented Reverse Engineering.
Master Thesis University of Bern, 1999.
- [21] E. Gamma, R. Helm, R. Johnson and J. Vlissides.
Design Patterns: Elements of Reusable Object Oriented Software.
Addison-Wesley, 1995
- [22] Ivan Moore.
Automatic Inheritance Hierarchy Restructuring and Method Refactoring.
OOPSLA96 CA, USA. pages 235-250, 1996
- [23] Demeyer & al.
The FAMOOS Object-Oriented Reengineering Handbook.
Version: October 15, 1999
- [24] S. Tichelaar and S. Demeyer.
An Exchange Model for Reengineering Tools.
In Object-Oriented Technology (ECOOP'98 Workshop Reader), LNCS 1543.
Springer-Verlag, July 1998.
- [25] Ralph. E. Johnson.
Documenting Frameworks using Patterns.
In Proceedings OOPSLA'92, ACM SIGPLAN Notices. pages 63-76, October 1992.
- [26] K. Beck and R. Johnson.
Patterns Generate Architectures. In M. Tokoro and R. Pareschi, editors, Proceedings ECOOP'94, LNCS 821, pages 139-149, Bologna, Italy, July 1994. Springer-Verlag.
- [27] Kent Beck. *Smalltalk Best Practice Patterns.*
Prentice Hall, 1997.
- [28] Kent Beck.
Extreme Programming Explained: Embrace Change.
Addison-Wesley, 1999.
- [29] M. M. Lehman and L. Belady.
Program Evolution - Processes of Software Change.
London Academic Press, 1985.
- [30] T. Ball and S. E. Eick.
Software Visualization in the Large.
IEEE Computer. pages 33-43, April 1996.
- [31] J Howard Johnson.
Substring Matching for Clone Detection and Change Tracking.
In Proceedings of the International Conference on Software Maintenance (ICSM), pages 120-126, 1994.

-
- [32] Santanu Paul and Atul Prakash.
A Framework for Source Code Search Using Program Patterns.
IEEE Transactions on Software Engineering, pages 463-475, 1994.
- [33] H.T. Jankowitz.
Detecting Plagiarism in Student PASCAL Programs.
Computer Journal, pages 1-8, 1988.
- [34] S. Grier.
A Tool that Detects Plagiarism in PASCAL Programs.
SIGSCE Bulletin, 1981.
- [35] M.H. Halstead.
Elements of Software Science.
Elsevier North-Holland, 1977.
- [36] Stéphane Ducasse, Matthias Rieger and Serge Demeyer.
A Language Independent Approach for Detecting Duplicated Code.
Proceedings ICSM'99 (International Conference on Software Maintenance)
pages 109-118. IEEE Computer Society Press, september, 1999.
- [37] Ian Sommerville.
Software Engineering.
Addison-Wesley, 1996.
- [38] Stéphane Ducasse, Michele Lanza and Sander Tichelaar.
*Moose: an Extensible Language-Independent Environment for Reengineering
Object-Oriented Systems.*
Proceedings of the Second International Symposium on Constructing Software
Engineering Tools (CoSET 2000), jun, 2000.
- [39] Charles W. Krueger.
Software Reuse.
ACM Computing Surveys, pages 133-183, jun, 1992.