SOFTWARE PLAGIARISM
DETECTION
USING MODEL-DRIVEN SOFTWARE
DEVELOPMENT
IN ECLIPSE PLATFORM

A thesis submitted to the University of Manchester
for the degree of Master of Science
in the Faculty of Engineering and Physical Sciences

2008

By
Pierre Cornic
School of Computer Science
## Contents

- Abstract ........................................... 7
- Declaration ....................................... 8
- Copyright ......................................... 9
- Acknowledgements ................................ 10
- 1 Introduction ................................. 11
- 2 Background .................................... 13
  - 2.1 Plagiarism detection techniques .......... 13
    - 2.1.1 Attribute-counting and structure-metrics systems .......... 13
    - 2.1.2 History of source-code plagiarism detection .............. 14
    - 2.1.3 Algorithms used in plagiarism detection ................. 15
      - 2.1.3.1 Winnowing ........................................ 15
      - 2.1.3.2 Greedy-String-Tiling ................................ 18
      - 2.1.3.3 Running-Karp-Rabin algorithm in Greedy-String-Tiling 19
    - 2.1.4 Plagiarism detection methods .................. 20
      - 2.1.4.1 Token and string-based systems ...................... 21
      - 2.1.4.2 Abstract Syntax Tree .............................. 22
      - 2.1.4.3 Program Dependence Graph ......................... 23
      - 2.1.4.4 Other methods ..................................... 25
  - 2.2 Eclipse Modelling Project ................ 26
    - 2.2.1 What is Eclipse? ............................. 26
    - 2.2.2 Model-Driven Software Development .............. 26
    - 2.2.3 Eclipse Modelling Framework (EMF) ................. 27
2.2.4 Textual Concrete Syntax (TCS) .................................. 28
2.2.5 ATLAS Transformation Language ............................... 28
2.3 Summary ............................................................... 29

3 Design ................................................................. 30
3.1 Principles of the software ........................................... 30
3.1.1 Top level architecture ........................................... 31
3.1.2 Generic front-end ................................................. 31
3.2 Generic model ....................................................... 32
3.3 Plagiarism detection ................................................ 34
3.3.1 Core comparison ................................................ 34
3.3.2 Engine ............................................................. 35
3.3.3 Results ............................................................ 35
3.3.4 Graphical User Interface ....................................... 36
3.4 Application integrated into Eclipse Platform .................... 37
3.4.1 Eclipse Platform ................................................ 37
3.4.1.1 Internal structure of Eclipse Platform ................. 37
3.4.1.2 Adding functionalities to Eclipse ....................... 38
3.4.2 Eclipse Rich Client Platform application .................... 40
3.5 Summary ............................................................... 40

4 Implementation ....................................................... 42
4.1 Developing Eclipse Plug-ins ...................................... 42
4.1.1 Eclipse and OSGi ................................................. 43
4.1.2 User Interface contributions ................................... 44
4.2 Loading models ..................................................... 45
4.2.1 Code hosting plug-in .......................................... 46
4.2.2 XMI files to Java objects ...................................... 47
4.2.3 Source files view ............................................... 49
4.3 Choice of the matcher .............................................. 50
4.3.1 Registering preferences ...................................... 50
4.3.2 Engine’s extension point ..................................... 51
4.3.2.1 Definition of the extension point ....................... 51
4.3.2.2 Contributing to the extension point ................... 53
4.3.2.3 Using the contributions .................................. 53
4.3.2.4 Preference page ............................................ 56
4.4 Comparing programs ........................................ 57
    4.4.1 Comparison job ..................................... 57
    4.4.2 Level algorithm ..................................... 58
    4.4.3 Token algorithm ..................................... 59
4.5 Display of results ........................................ 59
    4.5.1 Results data structure .............................. 60
    4.5.2 Histograms .......................................... 60
    4.5.3 Creation of the views .............................. 61
    4.5.4 View of programs side-by-side ..................... 61
4.6 Summary ..................................................... 62

5 Results and testing ........................................ 64
    5.1 Sample data results ................................... 64
        5.1.1 Simple plagiarism hiding techniques ............. 65
        5.1.2 More advanced techniques ....................... 65
            5.1.2.1 Statements reordering ....................... 65
            5.1.2.2 Blocks reordering and control structure modifications .... 66
        5.1.3 Code deportation into procedures ................ 67
        5.1.4 Variation of the parameters ..................... 68
    5.2 Real data results ...................................... 69
        5.2.1 Presentation of the data ......................... 69
        5.2.2 Analysis of the data ............................. 70
            5.2.2.1 First analysis ............................. 70
            5.2.2.2 Second analysis ............................ 72
    5.3 Performance testing .................................... 73
        5.3.1 Presentation ...................................... 73
        5.3.2 Time and parameters ............................. 74
    5.4 Summary .................................................. 75

6 Conclusion .................................................. 77
    6.1 Possible improvements ................................. 78

Bibliography .................................................. 79

A Plagiarism detection algorithms .......................... 81
    A.1 Greedy-String-Tiling algorithm ....................... 81
B Interfaces used for extension points 82
   B.1 IMatcher interface ........................................... 82
   B.2 IParametersForm interface ................................. 83

C Sample programs used for testing 84
   C.1 Original program ............................................. 84
   C.2 Statement reordering ......................................... 88
   C.3 Blocks reordering and control structures changes .... 92
   C.4 Deportation of code into procedures ...................... 96
List of Figures

2.1 Scanpattern procedure using Karp-Rabin algorithm . . . . . . . . 21
2.2 Abstract Syntax Tree . . . . . . . . . . . . . . . . . . . . . . . . 22
2.3 Example of Program Dependence Graph . . . . . . . . . . . . . . 24
2.4 EMF import and generation of the model . . . . . . . . . . . . . 27

3.1 Top-level architecture . . . . . . . . . . . . . . . . . . . . . . . . 31
3.2 Generic meta-model class diagram . . . . . . . . . . . . . . . . . . 33
3.3 Eclipse SDK architecture . . . . . . . . . . . . . . . . . . . . . . . 39

4.1 Workbench graphical elements . . . . . . . . . . . . . . . . . . . 44
4.2 Extension point’s entry in plugin.xml for a view . . . . . . . . . 45
4.3 List of exported package as shown in Eclipse PDE manifest editor 46
4.4 Browse filesystem in the Load programs view . . . . . . . . . . . 49
4.5 List of loading programs . . . . . . . . . . . . . . . . . . . . . . . 50
4.6 Creating an extension point . . . . . . . . . . . . . . . . . . . . . 51
4.7 Schema of an extension point . . . . . . . . . . . . . . . . . . . . 52
4.8 Contributions to the Engine.matchers extension point . . . . . 53
4.9 Preference page for the matcher’s selection . . . . . . . . . . . . 56
4.10 Job’s progress window . . . . . . . . . . . . . . . . . . . . . . . . 57
4.11 View of two programs side-by-side . . . . . . . . . . . . . . . . . 62

5.1 Plagiarism detection’s result on statement reordering . . . . . . 66
5.2 Plagiarism detection’s result on blocks reordering . . . . . . . . 67
5.3 Detection with code deportation into procedures . . . . . . . . . 68
5.4 Real data analysis with parameters 6/6 . . . . . . . . . . . . . . . 70
5.5 Results of the analysis as a list . . . . . . . . . . . . . . . . . . . 71
5.6 Real data analysis with parameters 2/2 . . . . . . . . . . . . . . . 72
5.7 Match between programs 5 and 10 . . . . . . . . . . . . . . . . . 73
5.8 Match between programs 5 and 13 . . . . . . . . . . . . . . . . . 73
Abstract

With the development of internet and electronic contents, plagiarism has become a serious issue in both professional and academic world. To overcome this issue many automated detection systems have been developed in the past thirty years. One of the area in which they are the most successful is the detection of source-code plagiarism. Because of the strict structure of programming languages, plagiarism is easier to detect in programs than in essays.

This thesis describes the conception, the design and the development of a software plagiarism detection application based on the Eclipse Platform. A generic front-end is used to convert source programs from different programming languages into generic models. Eclipse Modeling Framework is used to generate a Java implementation of these models and the plagiarism detection is applied on these models permitting to reuse the same comparison engine for many different programming languages. The results are finally displayed in a dedicated user interface providing facilities for their exploration. The report also includes accuracy and performance testing.
Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.
Copyright

i The author of this thesis (including any appendices and/or schedules to this thesis) owns any copyright in it (the “Copyright”) and s/he has given The University of Manchester the right to use such Copyright for any administrative, promotional, educational and/or teaching purposes.

ii Copies of this thesis, either in full or in extracts, may be made only in accordance with the regulations of the John Rylands University Library of Manchester. Details of these regulations may be obtained from the Librarian. This page must form part of any such copies made.

iii The ownership of any patents, designs, trade marks and any and all other intellectual property rights except for the Copyright (the “Intellectual Property Rights”) and any reproductions of copyright works, for example graphs and tables (“Reproductions”), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property Rights and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property Rights and/or Reproductions.

iv Further information on the conditions under which disclosure, publication and exploitation of this thesis, the Copyright and any Intellectual Property Rights and/or Reproductions described in it may take place is available from the Head of School of School of Computer Science (or the Vice-President).
Acknowledgements

I owe a great deal to my supervisor, my students friends and my family who helped and supported me during this year.

I would like to thank my supervisor, Dr Andrew Carpenter, who guided me along this project in particular through his knowledge of Eclipse Platform. His kindness and his availability allowed me to work in a studious and congenial atmosphere.

I also would like to thanks Nicolas Barithel who worked on the generic front-end used by this application. Our cooperation was always pleasant and fruitful and working with him on this project was highly motivating. He also made a particular commitment to provide me with test data at the early stages of the implementation of the detection engine.

This year in Manchester was a great and enjoyable experience in which Romain, Bun-Ny, Constant, Anthony, Maria, Su and Ioanna played an important role by their friendship.

Finally, I deeply thank my parents who gave me the opportunity to live this experience and the rest of my family who helped me through the most stressful moments.
Chapter 1

Introduction

During the 20th century, Plagiarism has become a serious problem, particularly in the academic area. Universities are more and more concerned about this because of the development of Internet and the increasing opportunity for students to copy and paste electronic content found on the web. But first of all, what is plagiarism? The term plagiarism is defined on dictionary.com as:

“the unauthorized use or close imitation of the language and thoughts of another author and the representation of them as one’s own original work.”

The meaning of this definition is easy to understand but what is more difficult is to decide, given a specified material, whether it is plagiarism or not. Due to the large number of possible sources, such a fact is impossible to determine manually. In order to solve this difficulty, software has been developed. This can be used to compare a set of documents and determine if one is plagiarized from one of the others, or to search, over the Internet or another database for plagiarism matches.

This project is focused on the detection of a particular type of plagiarism: software plagiarism, also called, source-code plagiarism. A lot of work has been done in computer science during the past 30 years concerning software plagiarism detection. Indeed, due to the structured nature of programming languages, it is much easier to identify plagiarism between two programs than between two essays. First it is necessary to explain what is meant by source-code plagiarism. Of course, two programs designed for the same task will be relatively similar,
particularly if they are only a few tens of lines long. Then, when is a program considered as plagiarism? According to Alan Parker and James O. Hamblen (1989) [11]:

“A plagiarized program can [be] defined as a program which has been produced from another program with a small number of routine transformations.”

By “routine transformations” they mean changes such as text substitutions or content reordering which do not require an understanding of the whole program. This report describes the design, implementation and testing of a source-code plagiarism detection system.

First, the background chapter gives some knowledge about software plagiarism detection. The differences between two types of plagiarism detection systems are explained, a brief history of detection software is given and the main methods as well as the algorithms they use are detailed. Then, the concept of Model-Driven Software Development is presented and several tools used for the development of the application are described.

The following chapter presents the design of the application. The top-level architecture is given and the main features are detailed. Then, the platform on which the development is realised and the application used is introduced.

The implementation chapter gives an outline of the development. The implementation of the various parts of the application is described, the algorithms used for the comparison of programs are presented and several key elements are highlighted.

Finally, a last chapter details the results of several tests. First, tests on sample data are described and the effect of several types of plagiarism hiding attempts is studied. Then the complete analysis process of a set of real programs is covered and finally, performance tests give information about the computational cost of the comparison.
Chapter 2

Background

The aim of this chapter is to give some background knowledge about plagiarism detection and to present the tools that have been used for this project. In a first section, the differences between the two main kinds of source-code plagiarism detection techniques are explained, a brief overview of detection systems over the past 30 years is given, and then some of the principal algorithms and techniques currently used are detailed. In a second section, the concept of model-driven software development is presented and some of the features provided by Eclipse Modeling Tools are introduced.

2.1 Plagiarism detection techniques

2.1.1 Attribute-counting and structure-metrics systems

It is possible to distinguish two types of source-code plagiarism detection systems. The first one is attribute-counting, where several values are computed from the source-code and programs that present similar values are considered as plagiarism. In fact, these systems try to capture the essence of a program in several numerical values. Depending on the measures used, these techniques can be very effective on basic plagiarism. However they cannot detect partial plagiarism: when a student copies only a part of another program.

An alternative is structure-metrics systems who put side-by-side the structures of the programs considered. After some pre-processing, the source-codes of the
different programs are directly compared to find matches. A similarity value is then calculated from the kind and number of matches found. All the software plagiarism detection systems currently in use exploit structure-metrics. Indeed, these techniques were shown more effective than attribute-counting by Verco and Wise (1996) [16]. In their paper they explain that attribute-counting systems are effective on plagiarism realized by inexperienced developers, where the original program is not heavily modified. Therefore they cannot be employed for all students.

2.1.2 History of source-code plagiarism detection

The first source-code plagiarism detection software was developed in 1976 by Ottenstein [10]. It was used to detect plagiarism in FORTRAN programs. This was an attribute-counting system that used Halstead’s software metrics:

- \( n_1 \): number of distinct operators
- \( n_2 \): number of distinct operands
- \( N_1 \): total number of operators
- \( N_2 \): total number of operands

From these four numbers, five attributes were calculated. Halstead created them to measure the complexity of programs. Ottenstein used the assertion that the probability of two programs written independently having the same attributes was really slim. Thus, the pairs which had the same attributes values were considered as plagiarism and selected to be examined by human eyes.

In 1980, Robinson and Soffa [13] developed a new system named ITPAD (Instructional Tool for Program ADvising) that was designed to help the course instructor with the assessment process of FORTRAN programs by verifying the quality of the code, detecting possible plagiarism and suggesting how the student could improve his code. The design of ITPAD consists in three phases. First a lexical analysis computes fourteen characteristics, including Halstead’s metrics, which provide elements for plagiarism detection and indicators of the quality of the code. The second phase is an analysis of the structure of the program using flow graphs and the third phase analyzes the results of the second phase.
CHAPTER 2. BACKGROUND

The first detection system using structure-metrics was created by Donaldson, Lancaster and Sposato in 1981 [5]. This piece of software uses attribute-counting metrics but also scans the source file to store information about several types of statements. This information is then coded using characters into string which are compared to find similarities. Since then, many structure-metrics systems have been released.

Whale developed a piece of software called Plague in 1990 [17]. It generates structure profiles from the input programs composed of structural information and transforms the code into sequences of tokens. The similar structure profiles are matched and their sequences of tokens are compared to find commons subsequences. Wise (1996) [19] created an algorithm called Running-Karp-Rabin Greedy-String-Tiling (presented in 2.1.3.3) to match sequences of tokens. He used it in a system called YAP3 that is still in use today. The strength of this algorithm are its robustness against code reordering and its low complexity. JPlag was presented in 2000 by Prechelt et al. It transforms the programs into sequences of tokens and compares the sequences with the Greedy-String-Tiling algorithm to determine their similarity.

2.1.3 Algorithms used in plagiarism detection

Most of the recent software presented in the previous section encode the structure of programs as sequences of characters or tokens. This section presents algorithms used to compare these sequences.

2.1.3.1 Winnowing

The winnowing algorithm is a method improving the efficiency of the comparison process based on documents fingerprinting. The concept of fingerprinting consists of obtaining a subset of a document in order to process the comparison on a smaller set of data. The fingerprint of a document is composed by hash values of k-grams. A k-gram is a substring of the document of length k. To obtain the fingerprint of a document, the text is divided into k-grams, the hash value of each k-gram is calculated and a subset of these values is selected to be the fingerprint of the document.
Listing 2.1 shows an original sample text. Then the whitespaces and undesirable characters are removed as in listing 2.2. And finally listing 2.3 shows the 5-grams obtained: every possible sequence of length 5 with consecutives characters.

**Listing 2.1**: Sample text

```plaintext
example of the winnowing algorithm
```

**Listing 2.2**: Text without whitespaces

```plaintext
exampleofthewinnowingalgorithm
```

**Listing 2.3**: Sequence of 5-grams obtained from the text

```plaintext
examp xampl ample mpleo pleof leofth ofthe fthew thewi hewin ewinn winno innow nnowi nowin owing winga ingal ngalg galgo algor lgori gorit orith rithm
```

A simple but incorrect approach would be to use the selection of every $i^{th}$ hash. However, such a selection is not robust against reordering or insertions/deletions. For instance, adding a simple character at the beginning of the file would shift the positions of all k-grams and would cause that the modified file will share none of its selected hashes with the original. Therefore, the selection of the hashes cannot rely on their position in the document. Winnowing avoids this by using only the data close to the considered location of the document to determine which hashes to select. It is the definition of a local algorithm.

Schleimer, Wilkerson and Aiken [14] name the interval between two consecutive selected hashes of a document a gap. A selection algorithm makes a compromise between the size of the fingerprint (the ratio of selected hashes) and the length of the gaps. A k-gram shared by two documents remains undetected if it is in a gap, leading to undiscovered plagiarism. The winnowing algorithm guarantees that, at least part of any common substring will be detected, subject to a minimal length of the common substring. In their paper, Schleimer, Wilkerson and Aiken enounce two properties they want to be satisfied when comparing a set of documents using fingerprints.

1. If there is a substring match at least as long as the guarantee threshold, $t$, then this match is detected.
2. Any match shorter than the noise threshold, k, is not detected.

The second property is verified by using hashes of k-grams. The constants t and k have to be chosen such that k is large enough to avoid noise matches and reduce the time of the comparison. The value of t should be large enough to avoid an excess of false positives and short enough to be robust against content’s reordering. If \( h_1...h_n \) is a sequence of hashes, with \( n > t - k \), at least one of the \( h_i \) must be selected to guaranty the first property.

Let’s define a window of size w as w consecutives k-gram hashes. Let’s take \( w = t - k + 1 \). If the document contains n hashes, each hash \( h_i \) with \( 1 \leq i \leq n - w + 1 \) defines the beginning of a window of size w. To guaranty property 1, one hash value should be selected in every window. The winnowing algorithm selects the minimum value of each window. If several values are minimal, the last one, i.e. the rightmost of the window, is selected. This choice is based on the fact that the minimal hash value of a window is very likely to remain the minimal value of the contiguous windows. Thus, the number of hashes selected is dramatically reduced and the property is guaranteed.

In their paper, Schleimer, Wilkerson and Aiken give the expected density of the winnowing algorithm. The density of a fingerprinting algorithm is the ratio between the number of hashes selected and the total number of hashes computed from a set of random values. They show that the density of the winnowing algorithm is:

\[
d = \frac{2}{w + 1}
\]

They compare this value to the density of the popular fingerprinting method consisting in the selection of the hashes that are 0 mod p. They present a modified version of this method which provides the same guarantee as the winnowing algorithm: at least one hash value of every substring of length greater than the threshold t is selected. The expected density of this algorithm is at least:

\[
d = \frac{1 + \ln w}{w}
\]

what is greater than the winnowing’s density, i.e. less efficient.
2.1.3.2 Greedy-String-Tiling

Greedy String Tiling is an algorithm introduced by Wise in 1993 [18]. This algorithm compares two strings and determines their degree of similarity. The strength of this algorithm is its ability to deal with transpositions. First, one should clarify the terms used in the description of the algorithm. In the following, the shorter string of the two compared is referred as the pattern string or pattern and the longer as the text string. Given P the pattern string and T the text string, Wise introduces several definitions:

A maximal-match is where a substring $P_p$ of the pattern string starting at $p$, matches, element by element, a substring $T_t$ of the text string starting at $t$. The match is assumed to be as long as possible, i.e until a non-match or end-of-string are encountered, or until one of the elements is found to be marked [explained later]. A maximal-match is denoted by the triple $\text{max\_match}(p, t, s)$, where $s$ is the length of the match. Maximal-matches are temporary and possibly not unique associations, i.e. a substring involved in one maximal-match may form part of several other maximal-matches.

A tile is a permanent and unique (one-to-one) association of a substring from P with a matching substring from T. In the process of forming a tile from a maximal-match, tokens of the two substrings are marked, and thereby become unavailable for further matches. A tile of length $s$ starting at $P_p$ and $T_t$ is written as $\text{tile}(p, t, s)$.

Wise also introduce the minimum match length, a parameter representing the length under which the maximal matches are ignored. This value is destined to improve the efficiency of the algorithm by eliminating insignificant matches. The aim of the algorithm is to find a set of tiles that maximize the coverage of T by P. The greedy-string-tiling algorithm is based on the idea that long matches are more interesting than short ones because they are more likely to represent significant similarities between the strings rather than coincidental resemblances. The pseudo code of the algorithm is presented in appendix A.1.

The algorithm performs multiples passes on the data, each of them is composed of two phases. In the first phase, what Wise calls scanpattern, all the maximal-matches above a certain length, initially the minimum-match-length, are collected and stored in lists, according to their lengths. The second phase
constructs tiles using the maximal-matches from the first phase beginning with the longest. For each match, the algorithm tests if it is marked, that means, already used by other tiles. If not, a tile is created with this match and the corresponding substrings in P and T are marked. Wise calls this phase markstrings. When all the matches of the considered length have been treated, a new smaller length is chosen and the search starts again from the first phase.

The algorithm stops when no unmarked substrings longer than minimum-match-length are found. Wise uses the term “token” in the description and the pseudo-code of the algorithm because it is not designed to be applied straight on the characters but after some pre-processing of the data. Wise shows that this algorithm is optimal for maximizing the coverage of the strings. He also shows that the worst case complexity of Greedy-String-Tiling is $O(n^3)$.

The most expensive phase is the first one, the search for matches. Therefore, this scanpattern phase is improved by using a second algorithm: Running-Karp-Rabin string matching.

### 2.1.3.3 Running-Karp-Rabin algorithm in Greedy-String-Tiling

Karp-Rabin string matching was created by Richard M. Karp and Michael O. Rabin in 1987 [8]. It uses fingerprints to find the occurrence of one string within another. The main idea of this method is the use of a hash function that can quickly compute the hash value of the $i^{th}$ k-gram from the hash of the $(i-1)^{th}$ k-gram. If a k-gram $c_1...c_k$ is considered as a k-digit number in a base $b$, let its hash value $H(c_1...c_k)$ be the number:

$$c_1 * b^{k-1} + c_2 * b^{k-2} + ... + c_{k-1} * b + c_k$$

Thus $H(c_2...c_{k+1})$ is obtained by subtracting $c_1 * b^{k-1}$ from the previous hash value, multiplying by $b$ and adding $c_{k+1}$. One obtains the following formula:

$$H(c_2...c_{k+1}) = (H(c_1...c_k) - c_1 * b^{k-1}) * b + c_{k+1}$$

This method allows computing of the hash values of all k-grams in a document in linear time. The algorithm computes the hash-value of the pattern string and compares it with each hash value of the k-grams in the document.
Wise extends Karp-Rabin algorithm to use it in the scanpattern phase of the Greedy-String-Tiling. He made the following changes:

- A hash value is computed for each unmarked k-gram of the pattern string P instead of only one value for the entire pattern. The same thing is done with all unmarked k-grams of the text string T.

- The hash value of each k-gram of P is compared to the hash values of the k-grams of T. To reduce the complexity of this operation, a hash-table of Karp-Rabin hash values is created and a search in this table returns all the positions of the k-grams with the same hash value. Once a match is found, the algorithm tries to extend it to the contiguous k-grams.

- After each iteration the length of the string searched (here, k) is reduced down to the minimum-match-length.

The first phase of the Greedy-String-Tiling iteration is now a separated procedure: \(\text{scanpattern}(k)\) where \(k\) is the minimum length of the matches searched. \(\text{scanpattern}(k)\) means that the Karp-Rabin hash algorithm will use k-grams. The pseudo-code of the procedure is reproduced in figure 2.1.

This procedure is designed to be used with small values of \(k\) because in practice very long matches are rare. If the maximal match length becomes much greater than \(k\), the test \(l > 2 \times k\) stops the procedure so the top-level algorithm could restart it with the new value. The \text{mark_token} procedure remains almost the same. After these optimizations, the complexity of the Running-Karp-Rabin Greedy-String-Tiling algorithm is \(O(n)\).

This algorithm is a good example of the use of hash values computed by the Karp-Rabin algorithm to optimize the comparison of two sets of data.

### 2.1.4 Plagiarism detection methods

The previous section presented algorithms used for the comparison of programs’ structures. This section details the main techniques used in plagiarism detection and how they exploit the algorithms described above.
CHAPTER 2. BACKGROUND

2.1.4.1 Token and string-based systems

The first method is to consider the program as a normal text. A pre-processing phase removes all comments, white spaces and punctuation, and renames variables into a common token. Then a string sequences comparison is performed. MOSS [1] is an on-line system for plagiarism detection. It performs a string-based comparison using the Karp-Rabin algorithm.

Alternative systems using token based methods parse the source-code and transform the elements into tokens. Then they perform a comparison on the sequences of tokens obtained in order to find the longest common substrings. The idea is that the set of tokens represents the key elements of a program. Comments and white-spaces are ignored because they are the first elements that a plagiarizer will alter. The main advantage of tokens is their generality; they discard all unnecessary information such as variables or methods names. Therefore, the
token-based systems are insensible to “search and replace” changes.

After tokenising the program, the strings of tokens are compared two-by-two to find common substrings. The most famous token-based software are JPlag [12] and YAP3 [19]. They both use the “Greedy String Tiling” algorithm presented in 2.1.3.2. Due to the properties of this algorithm they are robust against code reordering.

2.1.4.2 Abstract Syntax Tree

An Abstract Syntax Tree or AST is a hierarchical representation of a program. Each node represents a programming language construct and its children are the parameters of this construct. The nodes of an AST can be mathematical operators, function calls or other programming structures, the leaves are variables or constants. Figure 2.2, reproduced from Redhat website (www.redhat.com), shows the AST of the expression \( x = 3 \times (y - z) \).

![Figure 2.2: Abstract Syntax Tree](image)

AST are used by compilers as internal representation of data. They are an intermediate representation between a parser tree and a data structure. The difference between and AST and a parser tree, or concrete syntax tree, is that an AST contains only the nodes and edges that affect the semantic of the program. All unessential syntactic information is removed. For example, an AST does not need parenthesis to represent unambiguous mathematical expressions. Compilers perform optimizations on AST before generating lower-level code.

Because of this property, AST can be used in plagiarism detection. They allow changes at the lexical level to be automatically ignored. In this case,
detection consists of finding common sub-trees of the ASTs representing two programs. When an AST is used for plagiarism detection, variable names and constants are discarded to provide robustness against renaming. However, AST-based systems are often vulnerable to code insertion or reordering. Another difficulty is the complexity of the comparison between the trees. Searching for the largest common sub-tree of two trees requires comparison of all sub-trees two-by-two, which is inapplicable for large data sets.

Baxter et al. [2] presents a plagiarism detection system using AST designed for large software. Their comparison process is composed of three levels of detection. The first level identifies almost exact matches excepting variables and identifiers renaming. This first level of detection is performed by comparing hash values of sub-trees. An artificially bad hash function is used to detect similar sub-trees by ignoring sub-trees under a certain size threshold. They use the assumption that when reusing code, the programmer will do small modifications that will probably appear only near the leaves of the tree. Therefore, only parts of the tree bigger than a specified threshold are considered. After this phase, very similar sub-trees are detected and all the sub-trees are classified into buckets.

The second level uses a similarity value computed from the number of node shared against the size of the sub-trees. It computes the similarity for all trees in the same bucket and spots those with a similarity above a specified threshold. The last phase uses the results of the previous one; it looks at the parents of the detected similar sub-trees and tries to match them by combining the existing pairs. Baxter et al. implementation works well for detecting clones in source-code and seems to be applicable to plagiarism detection. Others tools using AST have been developed but they are not well documented and seems to be less effective than Baxter et al.’s software.

2.1.4.3 Program Dependence Graph

The Program Dependence Graph or PDG is a graph representation of a function or procedure’s source-code. In this representation, the statements are represented by nodes and the edges incident to a node represents the data modified or used in this statement and the control conditions on which the execution of the operation depends. PDGs show the deep structure of the programs. Figure 2.3 from the description of GPlag [9], shows an example of PDG.
Figure 2.3: Example of Program Dependence Graph

To the contrary of other representations used in plagiarism detection, a PDG stores no syntactical information. It presents the relationship between variables and operations. Thus, the alteration of the code by a plagiarizer will cause no changes of the PDG. Typically, plagiarizers perform modifications with no effect on the execution of the program such as variable renaming and statements reordering. These modifications cannot change the relations between these statements if the program’s correctness is preserved. In order to modify a PDG by altering the code, it would be necessary to understand and modify the functioning of the program, what will require a good understanding of it. Finally, the alteration of a PDG is very likely to require more work than the rewriting of the whole program, what is in contradiction with the main reason of plagiarism: spend less time than required to perform a task, or achieve results that one is not capable to do by himself.

The PDG-based plagiarism detection software uses this invariance property of PDGs. GPlag is one of these systems. The programs are transformed into set of PDGs and GPlag try to find sub graph isomorphism between the PDGs of the two programs. To reduce the computational cost, several filters are introduced such a threshold on the sub graph’s size or a heuristic function that performs a pre-detection on the PDGs and exclude from the comparison those that are too different.

Due to the properties of Program Dependence Graphs, PDG-based detection seem to be an interesting method that could be used in the future. However, there is not much documentation available on this subject; in particular, one cannot find statistical tests of GPlag. The developers of GPlag claims that it is able to detect even the most tricky plagiarism but it would be interesting to
The main effect of the first level of optimization is to include in the code all functions less than 600 lines long. This optimization is interesting for plagiarism detection because one of the plagiarism hiding methods widely used by plagiarizers is to deport some pieces of code into external functions. As such changes modify heavily the structure of the code, they can fool many detectors. The interest of this software is to use a compilation suite to translate the code from different languages into a lower-level common language. Though, the results of the tests performed on this system are not sufficient to decide whether this approach is accurate enough or not.

Another way of detecting plagiarism is to use compression software. As compression software tends to suppress repetitions in a file, the similar parts of a program will be compressed accordingly. To compare two files, A and B, one compresses them separately and then compresses the concatenation A+B of A and B. Let c(A) be the size of the compressed version of A. If c(A+B) is much smaller than c(A)+c(B), A and B share many similar portions of data. A measure of similarity distance can be defined as:

\[ d(A, B) = 1 - \frac{c(A) + c(B) - c(A + B)}{\max c(A)c(B)} \]

If A and B share many similar parts, their distance will be close to 0 and they
will be considered as plagiarism.

2.2 Eclipse Modelling Project

This section gives a presentation of the main tools used for the development part of the work described in this thesis. First a brief presentation of Eclipse will be given, then the concept of Model-Driven Software Development will be defined and finally, the purposes and functions of three components of Eclipse Modeling Project will be detailed.

2.2.1 What is Eclipse?

Eclipse is mainly known as an Integrated Development Environment for Java, but has become a lot more than that. Eclipse is now an open source community that merges more than 60 open source projects. These projects are classified into eleven top-level projects including: Web Tools Platform, Modeling, Service Oriented Architecture Platform and Equinox.

All these projects contribute to the functionalities of Eclipse, covering static and dynamic languages; thick clients, thin clients and server side frameworks; modeling and business reporting; embedded and mobile. The tools that have been used in this thesis are part of the Modeling category.

2.2.2 Model-Driven Software Development

Model-Driven Software Development or MDSD provides a range of approaches for developing software based on the use of modeling techniques. Instead of being first modelled with more or less details and then hand-coded, parts-of or whole software is designed as models and code automatically generated by code-generators.

For defining the various models, generating code, and every other function used in MDSD, several tools exist. The model-driven approach has been launched by the Object Management Group (OMG). This organisation produces specifications for the various processes needed in model development and their implementations are realised by privates companies or open source groups.
In a model-driven approach, the specifications are defined in a platform independent model using a corresponding domain-specific language. Then a platform definition model is used to translate this platform independent model into a platform specific model that can be run by a computer. The platform specific model could be the implementation of the platform independent model in a programming language, e.g. Java.

As one can see, model-driven software development uses a lot of different models. But how are these models expressed? A model conforms to a unique meta-model. A meta-model is a higher abstraction than a model, defining the properties of the model. In OMG’s world, the meta-models are written in the Meta Object Facility (or MOF) language. This is called a meta-metamodel as it is used to define meta-models. However they are no meta-meta-metamodels in OMG standards because the meta-metamodels are self-defined.

### 2.2.3 Eclipse Modelling Framework (EMF)

The generation of the implementation of a model is performed by the Eclipse Modelling Framework (EMF) [4, 15]. This framework is an Eclipse plugin used to create models and generate Java code from them. EMF consists of a meta-meta model called Ecore and tools for importing and generating code from source models. Ecore corresponds to a subset of MOF called Essential MOF or EMOF.

![Figure 2.4: EMF import and generation of the model](image)

The source model for an EMF project contains a description of an application data: objects, attributes, operations and constraints. This model can be in any
of several formats. Given a model as an UML class diagram, an Ecore metamodel, an XML Schema or Java interfaces, the framework generate the core code implementing the data structure. Figure 2.4 illustrates the import and the generation of the model. It shows that it is also possible to generate two higher-level layers. The “Edit code” provides undo/redo features and the “Editor” gives a Graphical User Interface to create and manage instances of the model.

In the work described, EMF is used to generate the Java implementation of the models representing the structures of the programs and the results of the detection. The Edit and Editor layer are not used in this project.

2.2.4 Textual Concrete Syntax (TCS)

TCS [6] is an eclipse component. It enables the formalization of textual concrete syntaxes for Domain Specific Languages. TCS allows a bi-directional transformation, text-to-model or model-to-text. Given an abstract syntax (meta-model) and a concrete syntax, the component generates a set of Java classes which implements the both transformations. The meta-model is described in KM3 (Kernel Meta Meta Model), a language developed by INRIA and used to describe abstract syntaxes for DSLs; this is similar to Ecore.

The concrete syntax describes how the syntactic elements are linked to the KM3 meta-model. After the generation of the parser, the meta-model is imported into EMF and transformed into an Ecore meta-model. Within the Eclipse Modeling Framework, these conversions are provided as one-click commands. EMF can then be used to generate the implementation of the model.

2.2.5 ATLAS Transformation Language

The ATLAS Transformation Language [3, 7] is a model transformation language and environment developed by the ATLAS Group (INRIA & LINA). It provides tools to define and process model-to-model transformations. ATL is a declarative language but it allows using imperative constructs for operations that are too complex to be done declaratively.

An ATL transformation is a set of rules specifying which objects of the source models are matched and which objects of the target model have then to be created
and initialized. To create an executable file to launch the transformation, ATL toolkit includes a specific transformation from ATL meta-model to the virtual machine bytecode.

2.3 Summary

In the first section of this chapter, the main techniques used for software plagiarism detection have been presented. Three algorithms were described. The winnowing algorithm is used to produce fingerprints from documents. It uses two parameters that give the maximum size of a gap and the minimum size of a match and minimizes the density of the fingerprint produced. The Greedy String Tiling algorithm compares two strings and returns a value representing their similarity. This algorithm is also controlled by a parameter and its computational time can be improved by using the Karp Rabin algorithm. The main plagiarism detection techniques, using these algorithms, have been detailed.

The second section has given an overview of Eclipse. The concept of model-driven software development has been presented and several notions such as meta-model have been introduced. Three tools belonging to Eclipse Modeling project have then been presented. Textual Concrete Syntax, or TCS is a parser generation tool based on meta-models and syntax descriptions. Eclipse Modeling Framework, or EMF, provides meta-models editor and code generation facilities and finally, the ATLAS Transformation Language permits the definition of model-to-model transformations.
Chapter 3

Design

This chapter details the design of the application developed in this project to detect source-code plagiarism. In the first section the top-level architecture of the application is described and the various components’ roles are explained. The second section presents how the generic meta-model links the two main components of the application, the third section details the plagiarism part’s architecture and the last section explains how and why the application is integrated into Eclipse Platform and the repercussions on the design. The Eclipse framework and the concept of Eclipse Rich Client Platform application are presented.

3.1 Principles of the software

The aim of the application developed for this project is to perform plagiarism detection over a set of programs and to perform this on sets of programs written in different programming languages. This detection includes the loading of the programs, their transformation in order to be ready for processing, the pairwise comparison over the set of programs, the display of the results and their exploration with a dedicated user interface. In order to be able to deal with multiple languages, the application uses a generic front-end. Its role is to make available the programs in a generic form to the comparison engine.

In a first part, the top-level architecture of the application is presented and in a second part the functioning of the generic front-end is briefly described.
3.1.1 Top level architecture

As presented in figure 3.1, the application is composed of two main parts. The first part consists in a generic front-end, allowing the programs written in all the kind of languages to appear in the same manner to the plagiarism detection engine. At the end of this part, the programs are expressed into a generic model and stored in XMI files. The generic front-end was developed by Nicolas Barithel; his work is outlined in more details in 3.1.2.

This thesis is focused on the plagiarism detector and the graphical user interface, not shown on this figure. This part loads as Java classes the source programs, performs the plagiarism detection and stores the results into a Java data structure. The method chosen is a structure-metrics system, shown by Verco and Wise to be more efficient. The models are compared two-by-two and a similarity value is computed. The information of all the steps of the pairwise detection is stored in a data structure used for the display of results.

A graphical user interface is also built on top of the two previous parts. It allows the parsing and the transformation of programs into generic models, the choice of the parameters of the comparison process, running the comparison and afterwards, explore the results. The design of the plagiarism detection part and the graphical interface are presented in 3.3.

3.1.2 Generic front-end

The generic front-end of the application is composed of two main parts. It has been realized by Nicolas Barithel. First the source-code is parsed into a
language specific model. The parsing is performed by a parser generated with TCS, presented in 2.2.4. Then, an ATL transformation (2.2.5) converts this language specific model into a generic model. This generic model is later used for the detection.

This front-end currently supports three languages: Java, Emfatic and Delphi. Therefore it includes at this point three parsers and three ATL transformations. For each language, an independent component (a plug-in as explained later) is created, hosting the parser and the transformations code.

To allow future enhancements this front-end has to be extendable. Thus, the core of this front-end defines an extension point. The nature of an extension point is described in 3.4.1.2. It discovers automatically the installed parsers usable by the front-end. Using an extension point permits to add a language to the front-end without changing one single line of code.

The front-end code component publishes a class to launch the parsing and transformation on all the files of a specified directory. When this class is used, the extension point finds all the available parsers. A list of supported languages, with associated file extension, is then computed from the discovered parsers and all the acceptable files are transformed into the generic model conforming to the generic meta-model.

### 3.2 Generic model

The generic model is the link between the two sections of the software. Therefore, its meta-model was designed jointly by Nicolas Barithel and Pierre Cornic. This meta-model was created principally for object-oriented programming languages. However, it can be used with other types of languages. It was inspired by Java but the creation of models from Emfatic and Delphi programs corresponding to this meta-model shows that it is generic enough to represent programs from many languages.

The class diagram of the meta-model is presented in figure 3.2. It shows that the root element is a Program. A Program contains a list of Statements. Both Program and Statement extends LocatedElement. This type is automatically created by TCS to host source file position information. This class was kept in the generic meta-model to give the plagiarism detection system such information.
Figure 3.2: Generic meta-model class diagram
The central element of this meta-model is the Statement. It is extended by
two sets of classes: the blocks and the simple statements. Block is an abstract
element representing all the instructions of the program that start a block of
instructions (e.g. conditions, loops, classes, etc.). By constrast, SimpleStatement
represents all the simple instructions such as declarations and assignments.

In order to use and browse the models for plagiarism detection, EMF is used
to generate a Java implementation of the models. The XMI files containing the
models are loaded by EMF API and represented as a set of Java classes. The
meta-model and the Java implementation are hosted by a plug-in.

3.3 Plagiarism detection

3.3.1 Core comparison

The core comparison component is the central part of the detection system. It
is responsible of comparing two programs. The results are stored in a hierarchical
data structure. The input values are the two programs and a set of parameters
that give the user control of the process. The output is the result data structure
giving details about the various phases of the comparison and the final result:
the similarity value. It is a number between 0 and 1 representing the amount of
similarities between the two programs. A 1.0 similarity shows that the programs
have exactly the same structures. A similarity of 0 indicates two very different
structures.

The Java implementation of the generic meta-model allows the extraction from
the models of many types of information. Therefore, many plagiarism detection
algorithms can be applied to a set of programs. In fact, there is not only one
core comparison component but many, each one implementing one plagiarism
detection algorithm. In the following, a component implementing an algorithm
will be called a matcher. The application allows the user to choose the matcher
used for the detection and to adjust its specific parameters.
3.3.2 Engine

The engine is the central part of the plagiarism detection system. It centralizes the data and results, and controls the execution of the whole process. The various step of the system are:

1. Select the input set of programs and load them.
2. Determine the comparison algorithm that should be used.
3. Adjust the parameters.
4. Compare each pair of programs, collect the results and report the progress of the comparison.
5. Analyse the results, compute specific result data and make them available.

The engine allows several matchers to be pluged in and makes them available for the comparison process. This is realised by an extension point, as described in more details in 3.4.1.2. Two matchers are currently implemented, their internal structure is detailed in the implementation chapter. Others algorithms can be added to the application without changing the code.

3.3.3 Results

The results structure is described by a model using the generic meta-model. One considers that the basic comparable element is a Block, therefore, the base element of the results structure is the result of the comparison between two blocks.

Each block’s comparison result contains references pointing to the blocks compared. These references allow the blocks to be flagged as plagiarized, or their positions retrieved and displayed in an editor. The results element also contains the overall size (number of statements) of each block including all its sub-blocks. The size can be used to calculate the overall similarity of several blocks comparisons.

The presentation of the results is critical for the user. One must keep in mind that the results of the software are not pairs of programs which are plagiarism but pairs of programs that require further investigations by a human assessor in order to decide whether there is plagiarism or not. Therefore, the software does not
return a list of programs that the assessor has to read line by line and compare manually, but it provides functionality that helps the manual comparison process.

The first problem encountered by plagiarism detection software is that it is impossible to set an absolute threshold between what is plagiarism or not in terms of similarity values. If a similarity of 99% is obviously plagiarism, and 1% is not, what is a pair with 50% of similarity? Moreover, if a skeleton code is provided by the lecturer for the assignment it will increase the similarity of all the pairs. To avoid this issue, a histogram of the similarity values is displayed at the end of the analysis. This histogram shows the distribution of the compared pairs between 0% and 100% of similarity. Using this histogram, it is easier to spot the pairs with a similarity sensibly higher than the other pairs.

3.3.4 Graphical User Interface

The GUI is built on the top of the plagiarism detection engine. However, the detection system could be used programmatically as it has been designed separately from the user interface. The GUI is used to control and start the various steps of the detection process:

1. Source programs: The user is invited to choose a directory. The containing files with registered extensions are parsed using the corresponding TCS parser and converted into the generic model by the related ATL transformation. The resulting models are loaded in memory. The user can repeat the operation on another directory and manage the list of programs.

2. Algorithm: Choose the matcher that should be used.

3. Parameters: Adjust the parameters of the comparison algorithm.

4. Detection: Start the comparison process and eventually cancel the operation. If the comparison is cancelled, the results computed so far are not lost and are automatically displayed.

5. Results: Show the results of the detection as a histogram of similarity values and an ordered list. The user can select one or several pairs of programs, display the original sources side-by-side and browse the detected matches.
3.4 Application integrated into Eclipse Platform

The choice was made to develop this software plagiarism detection application as an Eclipse RCP application. This section presents first the internal architecture of Eclipse Platform and then how this platform is used as base for rich client applications and what advantages it gives.

When Eclipse Project was launched in 2001, its aim was to provide a base for the creation of development environments. Over the years, many Eclipse’s users started to use parts of Eclipse for the development of client applications. Therefore many demands were made to extend Eclipse objectives to the development of client applications and not only development environments. The reactivity of Eclipse’s community leaded to the launch of Eclipse Rich Client Platform in 2004.

3.4.1 Eclipse Platform

3.4.1.1 Internal structure of Eclipse Platform

The main objective of Eclipse developers was to provide an extensible and modular framework. Therefore, Eclipse platform is organized around the concept of plug-in. A plug-in is a small component hosting code and/or data in order to provide new functionalities to the system. Certain plug-ins provide code libraries; others add visible functionalities to Eclipse user interface.

The core of Eclipse consists of a component managing the plug-ins lifecycles. This component administers the discovery, registration, loading, unloading of the plug-ins depending of the evolution of the environment. In fact, in order to avoid an overloading of the system Eclipses uses a lazy-start philosophy by loading the plug-ins only when they are absolutely required and not at the start of the system. This core component is responsible of keeping up-to-date a central registry of all the plug-ins registered in the running installation of Eclipse. Then, when a functionality is required, the component explores the registry for the plug-in providing the function and loads it. Apart from this core component, all the rest of Eclipse Platform consists in plug-ins providing the development tools, creating the graphical user interface, etc.
3.4.1.2 Adding functionalities to Eclipse

Adding new functionalities to Eclipse requires the development of new plug-ins. The Eclipse Software Development Kit (or SDK) provides two main components for the development of Eclipse Platform based applications:

- Java Development Tools (JDT) is a full featured Java development environment.
- Plugin Developer Environment (PDE) provides multiples assistances during the whole development cycle of a plug-in.

Therefore, the modules of the source-code plagiarism application’s architecture are developed as plug-ins adding several new functionalities to Eclipse platform: load a set of programs, compare these programs, display and explore the results, etc. These modules are linked together by the extension points and the plug-in dependencies systems.

Figure 3.3 from Eclipse website shows the internal structure of Eclipse SDK. Eclipse Platform includes:

**Workbench:** Eclipse graphical user interface which uses Eclipse’s Standard Widget Toolkit and JFace framework (a layer built on top of SWT providing graphical components).

**Workspace:** a dedicated folder on the file system storing the user preferences, the projects and all the data used by the application.

**Team:** a component used for collaborative work including a CVS client.

**Help:** a system providing a web server and mechanism allowing plug-ins to contribute to the documentation and create plug-ins based links rather than filesystem paths.

Eclipse PDE and JDT are plugged in on this platform.

Eclipse SDK allows plug-ins to define extension points where other plug-ins can add functionalities. An extension point consists in a unique id, prefixed by the id of the plug-in which defines it, and a XML schema describing the
characteristics of the extension point. Any plug-in can register an implementation of this extension point. The functionality provided can then be used transparently by the plug-in defining the extension point.

To register an implementation, an XML node is published by the plug-in, assigning values to the attributes defined in the extension point’s schema. This schema gives a list of the attributes who have to be provided by a plug-in who wishes to use the extension point. An attribute has a name, a boolean deprecated property, a use (required/optional, if optional a default value can be specified) and a type (Boolean, String, Java or Resource).

If the type is Java, a name of a class to extend or an interface to implement can also be provided. This variety of details allows the extension points to be very complex links between the plug-ins. The definition and use of an extension point will be detailed later in this report. This feature is crucial for the purpose of the project as one of the main goals was to provide an application easily extendable. In fact, with the mechanism of extension point, many features can be added to an application without changing even one line of code in the original set of plug-ins.

Moreover, if the new plug-ins can define extension points, a lot of extension points are already defined by Eclipse core set of plug-ins. Almost every aspect of Eclipse Platform is extendable in this way.

The distribution of plug-ins is also very easy because of the dependency systems. When a plug-in uses the classes or extension point defined by another plugin; the provider needs to figure in the list of dependencies of the user plugin. This dependency can specify a minimum and/or maximum version of the plugin.
If the installed version does not match the dependency requirement, a conflict is signalled to the developer or the user who can update his configuration.

### 3.4.2 Eclipse Rich Client Platform application

As described in 3.4.1.2, one can add functionalities to Eclipse Environment by developing plug-ins. However, the new plug-ins are in this case available through Eclipse IDE. It is also possible to run Eclipse Platform with a minimal set of plug-ins in order to produce a lightweight client application with a more specific GUI. This set of plug-ins is called Eclipse Rich Client Platform.

In a RCP application, the developer is responsible for the creation of all the elements of the graphical interface. The application class (the main class to be run as the application) has also to be specified using the extension point org.eclipse.core.applications.

The only direct dependencies required for a RCP application are org.eclipse.core.runtime and org.eclipse.ui. However, every plug-in installed in Eclipse IDE can be selected to be used in the application. Thus, Eclipse Rich Client Platform provides a stable base with multiples built-in functionalities to build rich client applications.

### 3.5 Summary

The first section of this chapter gives an outline of the features of the application and presents its top-level architecture. The system is composed of two parts: a generic front-end and a plagiarism detection engine, on top of which is built a graphical user interface. The second section presents the generic model who assures the interface between the generic front-end and the detection engine. Its meta-model is organised around the concepts of block, function and simple statements.

The third section details the various components of the plagiarism detection system. The engine is the main part of the system, centralizing the data and linking the other components. The core comparison part is responsible for the comparison of two models and returns a result structure containing information about the pair and a similarity value. The result part manages the processing and the display of the results. A histogram of the similarity values between 0
and 100% permits to spot the suspicious pairs and a view of the two programs side-by-side facilitate the manual comparison.

Finally the last section presents Eclipse Platform and explains the concept of a Rich Client Platform application. It highlights the crucial features provided by Eclipse Platform and how they facilitate the development of stable and scalable applications.
Chapter 4

Implementation

This chapter outlines the implementation of the various components of the application. In a first section the details of developing Eclipse plug-ins are presented. Then the generation of the generic model implementation and the loading of models using EMF are explained. The third section details how the extension point defined by the engine allows plug-ins to provide matchers for the comparison. The fourth section presents the creation of an eclipse job running the comparison task and the functioning of the two matchers developed during this project. Finally, the display of the result’s histogram and the view of programs side-by-side are described.

4.1 Developing Eclipse Plug-ins

As presented in 3.4.1.2, adding functionalities to an Eclipse application is achieved by developing new plug-ins. Eclipse plug-in development environment (or PDE) provides a dedicated IDE for the realisation of Eclipse plug-ins. It includes import and export features, dedicated graphical editor, run and debug environment, creation wizards, etc. This section presents the structure of a plug-in and how the plug-ins developed for this project make contributions to the user interface.
4.1.1 Eclipse and OSGi

Since version 3.0, the support of plug-ins in Eclipse is implemented using the OSGi specifications. OSGi framework specifies a model based on components. Eclipse implementation of OSGi framework is called Equinox and is the base of Eclipse framework. In this framework all the components called bundles can be dynamically installed, started, stopped, updated and uninstalled while the system is running. Thus, a plug-in is in fact an OSGi bundle.

In the OSGi specifications, the characteristics of a bundle are described in a text file MANIFEST.MF. This file contains the following information about the plug-in:

- Name: the bundle name, written in a “nice” way
- Symbolic name: bundle name without spaces, called plug-in ID in Eclipse.
- Version: the version of the bundle.
- Vendor: the bundle provider.
- Activator: the class to be called when the bundle is activated.
- Required bundles: the bundles that have to be installed for this bundle to be used.
- Exported packages: the Java packages published by the bundle to be used by other bundles.
- Imported packages: the Java packages imported by the bundle, without specifying the bundle that provides them.

In addition of the MANIFEST.MF file, an Eclipse plug-in is also defined via an XML file: plugin.xml. This file gives information about the features specific to Eclipse and not covered by OSGi specifications. It contains all the characteristics of extension points defined or implemented by the plug-in.

The main class of the plug-in contains a start() and stop() method containing the tasks required by the plug-in when it is started (initializations, listeners registration, creation of data structures, etc.) or stopped (saving of data, removal of listeners, etc.).
CHAPTER 4. IMPLEMENTATION

4.1.2 User Interface contributions

This section details the graphical elements used in Eclipse Platform user interface and explains how the plug-ins of the application make contributions to it. In Eclipse Platform the Workbench Window (the window opened when the application is started) contains several types of elements. Figure 4.1, from eclipse.org, shows some of them.

This application has a simple user interface. Therefore, the workbench window shows a menu bar and a single page containing several views. The menu bar is used for general basic actions: restart or exit the application, display the property pages or the about dialog.

The views are the main part of the application as they are used to retrieve data from the user and display the results of the detection. Views are defined in Eclipse through the extension point org.eclipse.ui.views. A plug-in displaying a view adds an entry in its plugin.xml giving details about the view: the class implementing it, the name of the view, etc.

Figure 4.2 displays an extension point’s entry contributing a view. “allowMultiple” indicates if this view can be added more than one time to the same page.
The attribute “class” points to a class implementing org.eclipse.ui.IViewPart or extending org.eclipse.ui.part.ViewPart. The id attribute will be used to refer to this view when some component needs to add it to the page; it is prefixed by the id of the plug-in. Finally, the “name” defines the title displayed on the top part of the view.

Views can be minimized, resized, moved into the workbench and even closed. A view can be added through the menu Window > Show View > ... but it can also be added to a page programmatically.

Views are meant to be used with SWT elements. SWT for Standard Widget Toolkit is a graphical library. When Eclipse was designed in 1999, its developers judged Java graphical libraries, Swing and AWT, not reactive enough and not well-integrated within the exploitation systems. Therefore they decided to implement their own graphical library: SWT which is, under Windows, not differentiable from the interfaces of others applications. However, a view can contain any graphical element either from SWT, Swing or AWT.

### 4.2 Loading models

This section details the loading of models from XMI files into a Java structure using EMF API. First it presents the Java implementation of the meta-model, then the use of EMF API is described and finally the functionalities of the user interface concerning this operation are presented.
4.2.1 Code hosting plug-in

The Java implementation of the generic meta-model is generated by EMF. This operation is achieved by the creation of an EMF plug-in project. The plug-in, called Generic, will contain the Java implementation of the meta-model and the Ecore file describing it.

The meta-model was first written in KM3 because the first prototype of the application parsed the program files into generic models, without the ATL transformation step. An Ecore meta-model was thus extracted from the KM3 meta-model, written for TCS.

This meta-model was however modified to add prototypes of operations. These operations do not modify the models but allow the extraction of additional information about the structure. For instance a method was added to get the size of a block. The Generic.ecore file is edited through a graphical editor provided by EMF.

A Generic.genmodel file is then created from the meta-model file through a wizard. It controls the code generation. The developer is able to specify the Java compliance level, the emplacement of the generated code, etc. Then, a right click on the root element of the Generic.genmodel file gives the possibility to generate the code. Five packages are then generated:

- Generic contains the interfaces of all the meta-model’s classes.
- Generic.iml regroups the implementations for the interfaces of generic package.
- Generic.util provides an adapter factory.
- PrimitivesTypes contains two interfaces for the support of the primitive types defined in the meta-model.
- PrimitivesTypes.impl provides the implementations of these two classes.

The skeleton of the additional methods are added to the code. The body of the method is then implemented by the developer who has to suppress the @generated tag. If the meta-model is modified, the code can be regenerated and the changes will be automatically merged with the previous sources, without overriding the customs implementations.

The role of the plug-in is to make the code available to plug-ins willing to work on generic models. Therefore, all the generated packages are exported through the MANIFEST.MF. This file is edited with the PDE editor for MANIFEST file as shown on figure 4.3. One can see a meta-model package in the list; it is a folder containing the Generic.ecore file used by the ATL transformation.

4.2.2 XMI files to Java objects

The models obtained from the generic front-end are stored in XMI files. These files are loaded and transformed into Java classes using the implementation previously generated.

On algorithm, 1 one can see the process of loading an EMF resource, containing the model. A resource set is used in EMF framework to manage document with possible cross-references. It creates the right type of resource for a given URI using a registry.

Here, the XMI implementation is registered as default type of resource in the Resource.Factory.Registry. Then, line 11 checks that the package is registered by accessing the instance. The resource is finally loaded from the file URI.

Once the resource is loaded, the objects contained in the resource are browsed in order to find the root element: the program object. Lines 25 to 31 illustrate this phase. When this object if found, several verifications and additional initializations are performed and the model is returned.
Algorithm 1 Loading of EMF models

```java
// Create a resource set.
ResourceSet resourceSet = new ResourceSetImpl();

// Register the default resource factory
resourceSet.getResourceFactoryRegistry().getExtensionToFactoryMap().
    put(
        Resource.Factory.Registry.DEFAULT_EXTENSION,
        new XMIResourceFactoryImpl());

// Register the package
GenericPackage Genericpackage = GenericPackage.eINSTANCE;

// Get the URI of the model file.
URI fileURI = URI.createFileURI(f.getAbsolutePath());

// Demand load the resource for this file.
Resource resource = null;
try {
    resource = resourceSet.getResource(fileURI, true);
} catch (Exception e) {
    throw new WrongModelException();
}

Program prog = null;
for (EObject o : resource.getContents()) {
    if (o instanceof Program) {
        prog = (Program) o;
        break;
    }
}
```

4.2.3 Source files view

In this section the part of the graphical interface managing the loading of the programs is presented. The philosophy of the programs loading in this application is to consider programs as contained in a source directory and not individually. As the aim of the software is to detect plagiarism in assignments, this approach is justified. The programs handed in by the students are regrouped in a unique folder by the assessor.

The graphical element is a view, as presented in 4.1.2. The first element of the view is a text field, containing the path of a source directory. The user can either type the path manually or press the “Browse” button which opens a browse directory dialog as presented on figure 4.4. One can notice on this figure that SWT is highly integrated in the operating system. The test computer is running a French OS, therefore the browse directory dialog is also written in French.

When the directory is chosen, a press on the “Load” button triggers the loading. A method from the Engine class is called. First the TCS parsing and ATL transformation are launched using a static method provided by the main class of the transformation plug-in “Specific2Generic”. Then, the directory is browsed and all the files with a gen.xmi extension are loaded as described in 4.2.2.
The resulting models are displayed in a list (see figure 4.5) and two buttons allow modification of the list. This process can be started again on another directory. So the user is able to compare files from different source directories.

### 4.3 Choice of the matcher

The application let the user choose between the installed matchers. This section describes how this choice is performed through the user interface and details the implementation of this feature.

#### 4.3.1 Registering preferences

Eclipse Platform provides a mechanism to store preferences for a given plug-in. These preferences are stored in the workspace of the application, and thus, persistent even with a restart of the application.

A call to the method `getPreferenceStore()` in the main class of a plug-in returns an object usable through the `IPreferenceStore` interface. It gives the ability to store simple types of data such as Boolean, Integer, Double, String, etc, indexed by string keys.
4.3.2 Engine’s extension point

4.3.2.1 Definition of the extension point

The link between the Engine plug-in and the matchers is an extension point. This is defined in the plugin.xml file of the Engine plug-in and is linked to an extension schema, also in the Engine plug-in. Let see how this extension point is defined.

When the MANIFEST.MF and plugin.xml files are edited through Eclipse PDE Manifest editor, a tab “Extension points” shows the extension points defined by the plug-in. As shown on figure 4.6, a click on the Add button opens a dialog asking for the extension point’s id, name and schema. Once this dialog is validated, the extension point is added in the list and the schema is opened through a dedicated editor.

In this editor, the first tab gives the possibility to describe the extension point in detail, give examples, list supplied implementations or add copyright information. The second tab defines the structure of the extension point. Figure 4.7 shows the edition of the “class” attribute of the extension point.

The extension element, automatically added by Eclipse, represents the extension point. It has three attributes, also added by Eclipse: point, id and name. Here, a new element is added: Matcher. The plug-in willing to contribute to
this extension point will do so by supplying a Matcher element. This Matcher contains four attributes:

**Id:** a unique string used to identify the matcher. The contributing plug-in should prefix this id by its own id in order to guarantee the unicity.

**Name:** a user-friendly title. The name will be used in the list displaying all available matchers.

**Class:** The class represents the main class of the matcher that will be instantiated in order to process the comparison. This attribute is of type Java and must implement the interface `engine.IMatcher`, produced in appendix B.1.

**parametersClass:** it is the only optional attribute. As the matchers may have different parameters, each matcher allowing the user to modify parameters related to the comparison has to provide a parameters class. This class extending `engine.IParametersForm` interface, presented in appendix B.2, is responsible for the creation of the parameters panel (detailed later) and the save of the values in a given preference store.

Once the Matcher element is created, a “Compositor” is added to the extension element. It defines that the extension point is a sequence of Matcher elements.
4.3.2.2 Contributing to the extension point

The Engine.matchers extension point is now fully defined. The contributing plug-ins possess all the information to implement it. This section presents how an implementation is created. The plug-in considered is CoreComparison, which provides the two matchers currently implemented.

Once again, we edit the MANIFEST.MF file in PDE editor. Under the “Extension” tab are listed all the extension points to which the plug-in contributes. The Add button browses the extension points defined by all the plug-in installed on the platform. The Engine.matchers point is selected and a right click on it gives the possibility to add a new Matcher. Figure 4.8 shows the Token Matcher’s contribution.

The four attributes defined in the extension point’s schema are displayed on the right. As the class and parametersClass attributes reference Java classes, two browse buttons allow to search for a class within the project classpath and the attribute’s name is an hyperlink launching the New class wizard automatically filled with the right values.

Once the classes implemented, the extension point is ready to be used. This aspect is presented in the next section.

4.3.2.3 Using the contributions

The extension point is defined and a plug-in contributes to it. This section describes how the engine uses the matcher. First, assume that one of the matchers has been chosen by the user. The process of making this choice will be detailed in the next part. An entry has been added in the preference store of the plug-in, containing the id of a Matcher element.
Algorithm 2 Retrieving the list of contributors

```
// Search the contributions to the extension point
String extensionPointId = "Engine.matchers";
IConfigurationElement[] contrib = Platform.getExtensionRegistry().
    getConfigurationElementsFor(extensionPointId);

IConfigurationElement matcherExtension = null;

// Go through the list to find the selected Matcher
for (int k = 0; k < contrib.length; k++) {

    // If no Matcher selected, the first one is chosen
    if (prefStore.getString("matcher_id") == null || prefStore.
        getString("matcher_id").equals("")) {
        prefStore.setValue("matcher_id", contrib[k].getAttribute("id"));
        prefStore.setValue("matcher_name", contrib[k].getAttribute("name"));
        matcherExtension = contrib[k];
        break;
    } else if (contrib[k].getAttribute("name").equals(prefStore.
        getString("matcher_id"))) {
        matcherExtension = contrib[k];
        break;
    }
}
```
Algorithm 3 Instantiation of a contributing class

```java
if (monitor != null) {
    if (monitor.isCanceled())
        return;
    monitor.subTask("Comparing programs ' + programs.elementAt(i).
        getName() + ' and '
        + programs.elementAt(j).getName());
}

m = (IMatcher) matcherExtension.createExecutableExtension("class");

m.setFirstProgram(programs.elementAt(i));
m.setSecondProgram(programs.elementAt(j));
m.setParameters(prefStore);
results.add(m.compare());

// Report progress
if (monitor != null) {
    monitor.worked(1);
}
```

Algorithm 2 shows how a list of IContributionElement is obtained from the extension point’s id. These objects represent the implementations to the extension point by the contributing plug-ins. Once these elements are obtained, the array is scanned in order to find a matcher with the same id as the one stored in the preference store. The test on line 11 checks that a matcher was selected. If not the first one is chosen and the loop is exited. Otherwise, line 17’s test selects the right matcher and exits the loop.

This snippet of code illustrates how the attributes of an extension point’s implementation are retrieved from the IContributionElement. The following code in algorithm 3 instantiates the matcher’s class and launches the comparison.

Line 7 instantiates the class designed by the attribute “class”. The disadvantage of this method is that no arguments can be passed to the constructor. Therefore, the programs to be compared and the parameters are set separately.
Figure 4.9: Preference page for the matcher’s selection

4.3.2.4 Preference page

The preference store allows plug-ins to store preferences permanently. As presented earlier, this store is used for the selection of the matcher and its parameters. This section details how the user can make this choice.

This process uses Eclipse Preference pages, accessible in Eclipse IDE through Window > Preferences. Here, an “Options” menu is created in the menu bar with an access to the preference pages. Once again, the contribution is made with an extension point: org.eclipse.ui.preferencePages. This extension points comprises an id, a name, a class and an optional category that will not be used here. The name is used to display the page’s entry in the left side of the preference pages window as shown on figure 4.9. The class must extend the PreferencePage class and implement the IWorkbenchPreferencePage interface.

A createContent method builds the page’s graphical content. Then, the performOK, performApply methods can be overridden to assign actions to the buttons. The main element of the created page is a drop down field displaying the various matchers detected. As the matchers have not the same parameters, the last attribute of the extension point, parametersClass is then used. This field allows the contributing plug-ins to publish a parameter’s class in addition to the matcher. The contributed class implements two methods, the first for creating an option panel, and the second to retrieve the values of this panel and store them in the preference store.
4.4 Comparing programs

Now that the link between the engine component and the matchers has been presented, the comparison process will be detailed. The creation of a comparison job will be presented and the algorithms of the two matchers implemented in this project will be detailed.

4.4.1 Comparison job

The entire comparison process is run inside a separated thread. In order to give control over the execution to the user, Eclipse’s job infrastructure is used. It provides a structure to create, schedule, and manage the progresses of jobs, unit of work running asynchronously.

To perform a task, a plug-in creates a job and schedules it. It is then added in a queue of jobs waiting to be run. The platform uses a background thread to manage the scheduling and the queue. When a job is selected to be run, its run() method is called. Eclipse Platform also provides an IProgressMonitor object controlling the execution of the job and permitting it to report its progress.

At the beginning of the run() method, the job calculates the number of comparison that have to be made. In this case for n programs: \( \frac{n(n-1)}{2} \). Then, for each comparison the job can report its progress through the IProgressMonitor. Algorithm 3 at line 16 illustrates this feature. From this progress monitor a progress bar can then be created as shown on figure 4.10.
Sometime the comparison of large set of long programs is expensive and takes several minutes. The user can then decide to cancel the job and modify the parameters of the matcher to perform a less detailed but faster comparison. This interaction is once again managed by the progress monitor who can be interrogated about the state of the job through the isCanceled method. Algorithm 3 at line 2 shows how the running job checks at every iteration if it has been canceled.

If the job is canceled the compare function returns prematurely, therefore all the results calculated before the cancelation are kept and displayed.

### 4.4.2 Level algorithm

This algorithm is based on the greedy-string-tiling algorithm presented in 2.1.3.2. However it does not consider the whole program as a sequence of tokens but compares the blocks separately. The steps of this algorithm are presented below.

1. For each pair of block (called b1 and b2), a sequence of tokens is extracted from the block’s statements. The two sequences of tokens are compared by the greedy-string-tiling algorithm, returning a similarity value between 0 and 1 called local similarity. The children blocks of b1 are classified in three categories: loops, controls and others. The same operation is done with b2’s blocks.

2. Each sub-block from b1 is compared by this algorithm with all the sub-blocks of b2 in the same category. The results are collected in three separated lists.

3. For each category, pairs of blocks are formed, starting with the pairs with the highest similarity value. A block can be used only for one pair.

4. The average similarity of each category is calculated from the pairs formed at the previous step. It is the mean of the similarities weighted by the size of the blocks.

5. The similarity resulting from the comparison of b1 and b2, called final similarity is calculated as the average of the similarities of the three categories and the local similarity.
This recursive algorithm can be controlled with two parameters. The minimum match length concerns the greedy-string-tiling algorithm. This parameter is detailed in 2.1.3.2. The second parameter is the minimum block size. The algorithm ignores all the blocks with a total number of statements (including the sub-blocks statements) smaller than this parameter. It allows to ignore matches caused by very small blocks and to reduce the computational time.

4.4.3 Token algorithm

This algorithm is simpler than the previous one. It extracts the sequences of tokens from the two programs and compares them directly in only one step using the greedy-string-tiling algorithm. It is controlled by only one parameter, the minimum match length. In this case too, the minimum match length controls the greedy-string-tiling algorithm and is detailed in 2.1.3.2.

However, the minimum match length used for this algorithm should be higher than in the previous case because the sequences of token are compared in one time and not separately for each block. The same implementation of the greedy-string-tiling algorithm is shared in the implementations of both algorithms.

4.5 Display of results

The loading of programs, choice of matcher and parameters, comparison process have been presented. This section is focused on the outcome of the application: the presentation of the results. As explained in 3.3.3, the display of the results is a crucial part of plagiarism detection software. The final judgment has to be made by a human assessor; therefore the software has to provide tools facilitating the analysis of the results. This section first presents the results’ model, from which the results’ data structure code is generated. Then the creation of histograms is briefly described and the visualization tool for pairs of programs is presented.
4.5.1 Results data structure

As the generic meta-model has a hierarchical structure, the results model has been designed in the same way. Therefore, the results model has a base element, the Comparison. This element includes several attributes: a final and a local similarity value, and lists of sub-comparison elements. The local similarity is destined to host the value of similarity resulting from the comparison of the two elements compared and the final similarity represents the combination of this local similarity and the similarities of the children comparisons.

Two elements extend this basic class: the comparison of two blocks and two programs. They respectively contain references to the two blocks and programs compared. Once again, the Java code is generated by EMF and hosted by a plug-in: ComparisonResult.

4.5.2 Histograms

Part 3.3.3 explained that a histogram will be used for the presentation of the results. This histogram shows the repartition of similarity values between 0 and 100%. It makes it easy for the user to spot the pairs presenting a high similarity with respect to the other programs of the set.

The creation of the histogram uses JfreeChart library. JFreeChart is an API distributed under the GNU Lesser General Public Licence. The principle of the API is to build and populate a data structure corresponding to the chart needed. Then, a chart factory provides methods creating any type of chart from the dataset and several others parameters.

Here, a histogram is needed. Therefore, the set of similarity values retrieved from the comparison has to be clustered in a finite number of ranges and then stored in the JFreeChart data structure corresponding to histograms.

For a histogram an object of type SimpleHistogramDataset is used. The clusters are represented by HistogramBin objects, they are added to the dataset and their item count is set manually. Then a call to ChartFactory.createHistogram(…) returns a JFreeChart object. A ChartPanel is then created from this chart. ChartPanel is a JFreeChart component extending the Swing container JPanel.
4.5.3 Creation of the views

The results are first displayed as two views: the histogram view and a view displaying an ordered list of the comparison results and their respective similarity value. The view showing the list gives the possibility to open a third view showing the both programs side-by-side. This view is detailed in 4.5.4.

The first approach to display these results would be to call the opening view function at the end of the ComparisonJob’s run method. However, as the job is run in a different thread from the UI thread, this operation is not permitted. In order to allow non UI thread to make contribution to the UI, Eclipse’s Display class has to be used. It provides a method taking a Runnable parameter which run() method is called by the UI thread.

A DisplayResults class, extending Runnable was therefore created to perform the creation of the results views.

4.5.4 View of programs side-by-side

To decide if a pair of programs contains plagiarism, a similarity value does not suffice. This view, showing the source-code of the two programs side-by-side, provides support for manual investigation. It is a view allowing multiple instances of itself to be created in the same workbench’s part. Thus, the user can open several pairs at the same time and compare the results.

At the creation of the view, the results structure is scanned and all the blocks with a final similarity greater than a specified threshold are flagged. They are then used for the construction of a list of matches. Each match uses the BlocksComparison’s references to retrieve the position of the blocks in the original programs through the LocatedElement, as mentioned in 3.2.

Two StyledText elements are then created and used to display the source-codes of the two programs. The list of matches is used to color all the lines flagged as matches. The user can navigate through the matches with two Previous/Next buttons, as shown on figure 4.11. A match is displayed by selecting the matching blocks for each program. In this manner, the limits of the match are clear and the user does not need to scroll to find it.

A control box gives the possibility to modify the similarity threshold and to
display only the matches longer than a specified number of instructions. For each
refresh, the color of the text areas is updated accordingly.

4.6 Summary

The first section of this chapter presents the relation between the OSGi framework and Eclipse Platform and the files defining the properties of a plug-in. It also gives an overview of the various elements of an Eclipse graphical interface and details the concept of view used by the plug-ins to retrieve and display information.

The second section describes the process of generating the Java implementation of the generic model. It explains the code used for the loading of the XMI files into Java instances and details the functions of the view used to control this loading.

The third section goes through the various steps of the definition and the implementation of an extension point using the example of the engine’s extension point. It presents how Eclipse preference store is used to register the preferences of the user concerning the matcher used for the detection and its parameters.

The fourth section first explains how the comparison job is created and the functionalities provided by Eclipse Platform to report the progress and control
the execution. The “Level Matcher” and the “Token Matcher” are then presented.

Finally the last section explains how JFreeChart library is used to create an histogram of the similarity value and how the hierarchical results structure is scanned to display the detected matches in the side-by-side view.
Chapter 5

Results and testing

This chapter presents the results and the testing of the application. Unit test have been realized using JUnit but they will not be explained in detail. JUnit is an open source framework for the realization of tests for Java programs. It was used for testing the loading of models and the matchers’ components. However, it could not be used with the engine because of the extension points that need the application to be run as an Eclipse application.

The first section presents accuracy tests on sample data. It highlights the robustness of the application against the main plagiarism hiding techniques. The second section presents a test on real data using a set of Java programs handed in for an assignment. Finally, the third section goes through performance testing. All the tests use the “Level Matcher”.

5.1 Sample data results

The sample programs use in this section has been found on the internet on www.java2s.com and plagiarized. The first section explains how the application detects the simplest hiding techniques, then tests are performed on more advanced plagiarism. The similarity values presented in this part are obtained with the Level Matcher with the following parameters: a minimal match size of 2 and a minimum block size of 2. These values may seem too small to mean real plagiarism but some tests realized on real data, presented in 5.2.2.2 with these parameters lead to an average similarity around 30 %.
5.1.1 Simple plagiarism hiding techniques

The simplest plagiarism hiding methods are the use of the Search and Replace function of the editor, the modification of the program’s indentation and the suppression, addition or modification of comments. All these changes are completely ineffective because of the nature of the generic meta-model.

The program’s structure is stored into the model which does not include comment or indentation. In the same manner, if the functions’ names are stored for reference resolution purposes, they are not used by the detection engine. The comparison is done on sequences of tokens; therefore, any modification of the variables or functions name is useless.

To disguise plagiarism, one can also break declaration and assignment statements in two lines by first declaring the variable and then assigning a value to it. In order to be robust against this kind of plagiarism hiding, declaration and assignments are always stored separately in the generic model. When a declaration and assignment are found on the same line, they are separated and treated as two different statements.

5.1.2 More advanced techniques

The second level of hiding methods consists of reordering statements. The plagiarizer has to have a minimal understanding of the program’s syntax in order to find out which statements can be reordered without changing the meaning of the code.

5.1.2.1 Statements reordering

First, only the simple statements are reordered. The variables declarations are moved at the top of the blocks, all the independent parts of the code are moved in order to change the look of the program. Here some renaming has also been performed and white lines have been added to change the look of the program.

The result of this detection is presented on figure 5.1. The program on the right is the original and on the left the plagiarized version. The match displayed has a similarity value greater than 92 % for 90 statements even if the two parts displayed look very different. The overall similarity for these two programs is 98 %
Figure 5.1: Plagiarism detection’s result on statement reordering

despite the fact that most of the possible reordering have been made. The original program is displayed in appendix C.1 and the plagiarized version in appendix C.2.

This robustness against reordering is a property of the greedy-string-tiling algorithm and the use of tokens. Because of the generality of the tokens used, interverting two types’ declarations, as for example:

```java
Int i;
BrowserListener listener;
```

is useless because they are both treated as declaration by the comparison engine.

5.1.2.2 Blocks reordering and control structure modifications

The second operation is the reordering of blocks and the change of control structure, for instance changing “for” loops to “while” loops and in the case of if/else structure, add a negation to the condition and switch the “if” and “else” blocks.

The following program has been plagiarized from the version of 5.1.2.1 by doing blocks reordering without changing the meaning of the program. The classes and methods have been reordered, the “for” loops have been transformed into “while”. The plagiarized code is given in appendix C.3.
Figure 5.2 shows that this last modification is useless. In fact there is no distinction between “for” and “while” loops in the generic model. The first ones are always decomposed by adding a declaration if necessary, an assignment before the loop and an assignment inside the loop. These operations are the ones required to transform a “for” loop into a “while”, so any modification at this level by a plagiarizer is ineffective. Figure 5.2 shows that the similarity of the block containing this loop is 100%. The overall similarity between this program and the original is 96%.

The diminution of 2% of the similarity may have been caused by the presence of class declarations inside the Browser class. As methods and classes are represented by different tokens, this reordering prevented the two sequences of tokens to match exactly and lowered the similarity. However, this diminution is small with respect to the modifications made to the program.

5.1.3 Code deportation into procedures

One of the most confusing plagiarism hiding for a human eye is to move code into procedures. It causes important changes to the body of the method, is easy to do and does not need a deep understanding of the code, only some notions about variables’ range.

This method is almost ineffective with the generic model. In fact, when a
Figure 5.3: Detection with code deportation into procedures

FunctionCall object is found, the code used for the detection is the code of the corresponding FunctionDeclaration. This is possible because of the capabilities of TCS, which deals with reference resolution. This could cause problems with recursive functions but the reference is followed only for one occurrence, after that, the recursion is detected.

Figure 5.3 shows a match with a 100% similarity despite the fact that pieces of code have been moved into other functions. The only problem is that the code of the added function cannot be matched with other code, therefore the similarity value of the pair drops to 89%. However, the value is still much higher than the average value observed during the use of the algorithm which means that this pair would have been selected for further examination in a real detection situation. The examiner would have seen the perfect match (100% of similarity) and spotted the attempt of plagiarism disguise.

5.1.4 Variation of the parameters

The parameters chosen for these tests were small. The reason of this choice is that most of the blocks of the original program contained less than 4 statements. Therefore, a minimal match length greater than 4 would have caused a less accurate detection. This section presents the progression of the similarity values.
with the variation of the parameters. A new program with a comparable number of lines is also added to the set in order to give an example of a non plagiarized pair.

Table 5.1 shows the results for this sequence of tests. The first line indicates the parameters used, the first number being the minimum match length and the second the minimum block size. One can notice that the minimum block size used is always greater than the minimum match length. It is a principle of common sense because the size of a block will always be greater or equals than the length of a sequence of tokens extracted for one level of this block. Therefore, if the minimum block size used is smaller than the minimum match length, the algorithm will attempt to compare the blocks with a number of lines greater than the size threshold even if it is smaller than the minimum match length. For these last blocks the comparison will fail anyway, giving no interesting results.

One can see that the similarity values decrease when the values of the parameters increase. The similarity values stay high except for the parameters couple 10/10 when they drop. This brutal drop is due to the nature of the programs compared composed of small blocks. When the parameters are two high, the results obtained are meaningless. One can also notice that the similarity value of the “Other” program is always much lower than the plagiarized programs showing that very different programs cannot have a high similarity value even with small parameters.

### 5.2 Real data results

#### 5.2.1 Presentation of the data

The set of programs used in this section has been obtained from an assignment in a Master’s course at the University of Manchester. The data were originally
provided as folders containing several Java files. The folders were labelled by numbers to respect the privacy of the authors. A common code file that was given to the students was also supplied.

The assignment was separated in 6 parts, however, not all the students had made different classes for each part. Therefore, for each student, all the Java files contained in the handed-in folder were merged into a unique file called studentXX.Java. Therefore, the test data are composed by a common_code.Java file and 13 files from student01.Java to student13.Java. It represents 91 comparisons.

5.2.2 Analysis of the data

This section will go through the analysis of the programs, the exploration of the results and the effect of the parameter’s variation.

5.2.2.1 First analysis

The first analysis is done with high parameters’ values in order to get a first idea of the time needed for the comparison and eventually first cases of plagiarism. Figure 5.4 presents the histogram resulting from the analysis of the programs with a minimum match length of 6 and a minimum block size of 6.
CHAPTER 5. RESULTS AND TESTING

Figure 5.5: Results of the analysis as a list

The histogram presents 100 ranges from 0-1% to 99-100% of similarity. For each class, the number of pairs with a corresponding similarity value is displayed. The expected result is to see a cluster of programs around a low similarity value and several pairs with a much higher value. These pairs are the suspicious ones. Figure 5.4 shows 13 pairs with a 0% to 1% similarity. Then, most of the other pairs are homogenously distributed between 0% and 73% and one pair shows a high similarity value of almost 93%.

This indicates that the pair near 93% has a high probability of containing long parts of plagiarized code, and that the pairs around 70% should be checked carefully. The second view, presented on figure 5.5, presents the ordered list of similarity values. It shows that the suspicious pair is composed by the students 11 and 12 files.

The selection of a pair in the results’ list view gives access to a side-by-side comparison view. This view, as presented in 4.5.4, gives the possibility to explore the detected matches between the two source files. A quick exploration of the 11/12 pair shows undeniable plagiarism. Apart from the common code, several classes were found in the two programs almost unchanged.

By contrast, the exploration of the results for the pairs around 70% of similarity could not reveal common parts apart from the common code. The exploration revealed that the common code’s class was modified and renamed in one of the program but without knowing the conditions of the exercise, it is impossible to determine if it should be considered as plagiarism.

A second analysis is then made with smaller parameters to determine if other
matches are revealed.

5.2.2.2 Second analysis

A second analysis is performed with smaller values of the parameters. A minimum match length and a minimum block size of 2 are chosen. Figure 5.6 shows the resulting histogram. As expected, the pair 11-12 stays at a high similarity value. Seven pairs are then located around 70%. As there is no need to check again the pair 11-12, the exploration begins with the second highest similarity value: the pair 5-10.

For the display of the matches, the default similarity threshold is the final similarity of the pair of programs. In addition, the minimum block size set for this analysis is small, which means that insignificant matches, such as auto-generated catch blocks, are displayed. Therefore, the exploration view shows, on this example, 46 matches with a similarity value greater than 72%. The option panel gives the possibility to modify the similarity threshold and to set a size threshold in order to accelerate the exploration.

For this pair, apart from the functions belonging to the common code provided, only one match is noticed. It has a similarity value of 85% and its length is long enough to be significant. It is shown on figure 5.7. Apart from a line printing a time, these two methods are identical.
5.3 Performance testing

5.3.1 Presentation

The aim of this section is to give an idea of the time needed by the application to analyse the set of programs used in the previous section. The measures are realised using a utility class registering the current time in milliseconds at given points of the execution. These tests are performed on a Compaq nx8220 (2005) laptop with an Intel Pentium M 740 cadenced at 1729 MHz and 2 GB RAM.
Table 5.2: Number of lines of the programs

<table>
<thead>
<tr>
<th>Name of the file</th>
<th>Number of lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>common_code.Java</td>
<td>486</td>
</tr>
<tr>
<td>student01.Java</td>
<td>1213</td>
</tr>
<tr>
<td>student02.Java</td>
<td>46</td>
</tr>
<tr>
<td>student03.Java</td>
<td>2015</td>
</tr>
<tr>
<td>student04.Java</td>
<td>1691</td>
</tr>
<tr>
<td>student05.Java</td>
<td>1740</td>
</tr>
<tr>
<td>student06.Java</td>
<td>1792</td>
</tr>
<tr>
<td>student07.Java</td>
<td>902</td>
</tr>
<tr>
<td>student08.Java</td>
<td>388</td>
</tr>
<tr>
<td>student09.Java</td>
<td>188</td>
</tr>
<tr>
<td>student10.Java</td>
<td>1833</td>
</tr>
<tr>
<td>student11.Java</td>
<td>4275</td>
</tr>
<tr>
<td>student12.Java</td>
<td>4322</td>
</tr>
<tr>
<td>student13.Java</td>
<td>1778</td>
</tr>
</tbody>
</table>

Table 5.2: Number of lines of the programs

<table>
<thead>
<tr>
<th>MML \ MBS</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>93.9</td>
<td>70.1</td>
<td>66.0</td>
<td>57.4</td>
<td>55.2</td>
</tr>
<tr>
<td>4</td>
<td>X</td>
<td>66.0</td>
<td>62.5</td>
<td>54.4</td>
<td>53.1</td>
</tr>
<tr>
<td>6</td>
<td>X</td>
<td>X</td>
<td>59.3</td>
<td>52.1</td>
<td>49.2</td>
</tr>
</tbody>
</table>

Table 5.2: Number of lines of the programs

running Windows Vista.

Table 5.2 shows the sizes (numbers of lines) of the programs presented in 5.2.1. The differences of the sizes are due to design choices and students who did not complete all the tasks required.

5.3.2 Time and parameters

Table 5.3 shows the time required to run the analysis of the entire set of programs for various values of parameters. The analysis was run for 3 values of minimum match length (MML) and 5 values of minimum block size (MBS).

These times do not include the parsing, transformation or loading of the programs. As explained in 5.1.4, the algorithm should always be used with a minimum length match smaller than the minimum block size. Even though not presented in this diagram, one should notice that for each test, a significant amount of time was taken by the comparison of students 11 and 12. These are
by far the longest programs of the set and more similar, causing a comparison
time between 8 and 20 seconds where the average time for the other pairs varies
between 0.6 and 1 second.

As one can see the time required for the analysis decreases when the param-
eters increase. A huge improvement is noticed when the parameters are raised
from 2/2 to 4/4 or 2/4. The threshold of 4 for the size of the blocks results in
auto-generated catch blocks and small parameters controls being ignored, causing
a noticeable time improvement. At the same time, a minimal match length of 4
seems to be long enough to guarantee that the matches are meaningful and short
enough to ensure a certain degree of robustness against reordering.

The second noticeable improvement is the raising of the minimum block size
from 6 to 8. In proportion the improvement is better than all the other changes
except from the previous one. This could mean than a lot of blocks have a length
between 6 and 8 statements, causing a big improvement in the performances when
they are ignored. In the same time however, it represents a possible deterioration
of the detection’s quality.

5.4 Summary

This section has shown that the application produces believable results on
a set of real data. The histogram view permits the spotting of the suspicious
pairs and the view showing the programs side-by-side gives a practical solution
for a manual exploration of the results. The detection is accurate enough to spot
identical or almost identical pieces of code in some of the programs. For instance,
the common code elements were always given high similarity values. These parts
could have been removed prior the analysis but if some students had kept the
common code in a separated class and some others had mixed it with their own
code, making it difficult.

These results also highlight the fact that the application, as does all cur-
rent plagiarism software, does not give absolute results but signals the programs
requiring further investigation. The performance tests show a considerable influ-
ence of the parameters on the computational time. However, the tests realised on
sample data highlight the problems caused by the utilisation of high values for
the parameters if the average block’s size of the program is small. In addition,
the exploration of the results through the view of the programs side-by-side is
easier and more efficient with a small value of the minimum match length.
Chapter 6

Conclusion

This report presents the various aspects of the design and development of a source-code plagiarism detection application. This application uses a generic front-end to convert programs from several programming languages into generic models. The plagiarism detection is then performed on these generic models. This architecture gives the possibility of developing and maintaining only one plagiarism detection engine for use on many languages.

Model-driven development tools were used to implement the link between the generic front-end and the detection engine. As the generic front-end outputs models as XMI files, Eclipse Modeling Framework was used to generate a Java implementation of the generic meta-model. EMF was also used to generate the structure hosting the detection’s results. The use of model-driven software development concept significantly dropped the development time of these parts of the application. Once the model specified, the generation of the code using EMF is straightforward and further modifications are easy, the new code being automatically merged with the results of the previous generations.

The application was developed as an Eclipse Rich Client Platform application. Each component is hosted by a plug-in and all the plug-ins are linked together by extension points and the dependencies system. Eclipse Platform provides a stable and extendable architecture for the development of applications. It also supplies a complete framework for the creation of graphical interfaces. However, the biggest advantage of this choice is the possibility to use extension points. This concept allows the application to be indefinitely extendable. The generic front-end’s extension point permits the addition of a parser and ATL transformation for
any programming language and the engine’s extension point gives the possibility of adding plagiarism detection algorithms that can be immediately used.

Currently the generic front-end supports three programming languages and the detection engine is able to use two different matchers. If the “Token Matcher” is a simple implementation of techniques used in other plagiarism detection software, the “Level Matcher” is a complex algorithm robust against many types of plagiarism hiding methods. It is particularly robust against blocks reordering and code deportation into procedures.

Tests on sample data have illustrated the capabilities of the algorithm and a complete analysis of real data has validated its use for real plagiarism detection. However, the development’s objectives were focused on the accuracy of the detection algorithm and not its performance. Therefore, even if the performance tests show acceptable results on relatively long programs, several optimizations should be made before using this application on a regular basis.

6.1 Possible improvements

Apart from the development of other parsers to support new programming languages, the possible improvements concern mostly performance. The comparison job could be multi-threaded relatively easily, improving the performances of the application, particularly on the new computers generally equipped with multi-core processors. This perspective was taken into account during the development by using structures providing synchronized access to merge the source models and the results of the detection. Several refactoring may also lead to significant improvements in the performances.

The performance may also be improved by the use of a heuristic function to determine if the comparison of two sub-blocks gives the chance for significant similarity value, avoiding a potentially long useless comparison if not.

Another aspect would be the realisation of detection tests on many types of data. Unfortunately the only suitable test data available for this project, a set of programs in Java, Delphi or Emfatic, achieving the same results, were the programs used in 5.2. However, tests on other sets of programs may lead to less accurate results and eventually cause changes in the algorithm.
Bibliography


Appendix A

Plagiarism detection algorithms

A.1 Greedy-String-Tiling algorithm

1 \texttt{length\_of\_tokens\_tiled} := 0
2 \texttt{Repeat}
3 \hspace{1em} \texttt{maxmatch} := \texttt{minimum–match–length}
4 \hspace{1em} \texttt{starting at the first unmarked token of } P, \texttt{for each } P[p] \texttt{ do}
5 \hspace{2em} \texttt{starting at the first unmarked token of } T, \texttt{for each}
6 \hspace{3em} T[t] \texttt{ do}
7 \hspace{4em} j := 0
8 \hspace{5em} \texttt{while } P[p]+j=T[t]+j \texttt{ AND unmarked}(P[p]+j) \texttt{ AND}
9 \hspace{6em} \texttt{unmarked}(T[t]+j) \texttt{ do}
10 \hspace{5em} j := j + 1
11 \hspace{4em} \texttt{if } j = \texttt{maxmatch}
12 \hspace{5em} \texttt{then add match}(p, t, j) \texttt{ to list of}
13 \hspace{6em} \texttt{matches of length } j
14 \hspace{5em} \texttt{else if } j > \texttt{maxmatch}
15 \hspace{5em} \texttt{then start new list with}
16 \hspace{6em} \texttt{match}(p, t, j) \texttt{ and}
17 \hspace{6em} \texttt{maxmatch} := j
18 \hspace{1em} \texttt{for each match}(p, t, \texttt{maxmatch}) \texttt{ in list}
19 \hspace{2em} \texttt{if not occluded}
20 \hspace{3em} \texttt{/* Create new tile */}
21 \hspace{4em} \texttt{for } j := 0 \texttt{ to } \texttt{maxmatch} - 1 \texttt{ do}
22 \hspace{5em} \texttt{mark\_token}(P[p]+j)
23 \hspace{5em} \texttt{mark\_token}(T[t]+j)
24 \hspace{2em} \texttt{length\_of\_tokens\_tiled} :=
25 \hspace{3em} \texttt{length\_of\_tokens\_tiled} + \texttt{maxmatch};
26 \texttt{Until maxmatch = minimum–match–length}
Appendix B

Interfaces used for extension points

B.1 IMatcher interface

```java
package engine;

import org.eclipse.jface.preference.IPreferenceStore;
import ComparisonResult.ProgramComp;
import Generic.Program;

public interface IMatcher {
    /**
     * Performs the comparison once the parameters and input programs have been set.
     * @return result of the pairwise comparison
     */
    ProgramComp compare();

    /**
     * Set the first program to be compared.
     * @param p input model.
     */
    void setFirstProgram(Program p);

    /**
     * Set the second program to be compared.
     * @param p input model.
     */
    void setSecondProgram(Program p);

    /**
     * Set the parameters corresponding to the matcher.
     */
    void setParameters()
```
APPENDIX B. INTERFACES USED FOR EXTENSION POINTS

28     * @param prefStore plugin’s preference store.
29     */
30     void setParameters(IPreferenceStore prefStore);
31 }

B.2 IParametersForm interface

package engine;
import org.eclipse.jface.preference.IPreferenceStore;
import org.eclipse.swt.widgets.Composite;

public interface IParametersForm {

    /**
     * Get the parameters panel corresponding to the matcher.
     * @param content parent panel.
     * @param prefStore plugin’s preference store.
     * @return updated composite.
     */
    public Composite getForm(Composite content, IPreferenceStore prefStore);

    /**
     * Retrieve input data from the panel and store them.
     * @param prefStore plugin’s preference store.
     */
    public void saveValues(IPreferenceStore prefStore);
}
Appendix C

Sample programs used for testing

C.1 Original program

```java
public class Browser extends JFrame {

    protected JEditorPane m_browser;

    protected MemComboBox m_locator = new MemComboBox();

    public Browser() {
        super("HTML Browser");
        setSize(500, 300);
        getContentPane().setLayout(new BorderLayout());

        JPanel p = new JPanel();
        p.setLayout(new BoxLayout(p, BoxLayout.X_AXIS));
        p.add(new JLabel("Address"));

        m_locator.load("addresses.dat");
        BrowserListener lst = new BrowserListener();
        m_locator.addActionListener(lst);

        MemComboAgent agent = new MemComboAgent(m_locator);

        p.add(m_locator);

        getContentPane().add(p, BorderLayout.NORTH);

        m_browser = new JEditorPane();
        m_browser.setEditable(false);
        m_browser.addHyperlinkListener(lst);

        JScrollPane sp = new JScrollPane();
        sp.getViewport().add(m_browser);
```

84
APPENDIX C. SAMPLE PROGRAMS USED FOR TESTING

```java
32 getContentPane().add(sp, BorderLayout.CENTER);
33
34 WindowListener wndCloser = new WindowAdapter() {
35   public void windowClosing(WindowEvent e) {
36     m_locator.save("addresses.dat");
37     System.exit(0);
38   }
39 }
40 addWindowListener(wndCloser);
41
42 setVisible(true);
43 m_locator.grabFocus();
44 }
45
46 class BrowserListener implements ActionListener, HyperlinkListener {
47   public void actionPerformed(ActionEvent evt) {
48     String sUrl = (String) m_locator.getSelectedItem();
49     if (sUrl == null || sUrl.length() == 0)
50       return;
51
52     BrowserLoader loader = new BrowserLoader(sUrl);
53     loader.start();
54   }
55
56   public void hyperlinkUpdate(HyperlinkEvent e) {
57     URL url = e.getURL();
58     if (url == null)
59       return;
60     BrowserLoader loader = new BrowserLoader(url.toString());
61     loader.start();
62   }
63 }
64
65 class BrowserLoader extends Thread {
66   protected String m_sUrl;
67
68   public BrowserLoader(String sUrl) {
69     m_sUrl = sUrl;
70   }
71
72   public void run() {
73     setCursor(Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR));
74     try {
75       URL source = new URL(m_sUrl);
76       m_browser.setPage(source);
77       m_locator.add(m_sUrl);
78     } catch (Exception e) {
79       JOptionPane.showMessageDialog(Browser.this, "Error:");
80     }
81   }
82 }
```
public static void main(String argv[]) {
new Browser();
}

class MemComboAgent extends KeyAdapter {
    protected JComboBox m_comboBox;
    protected JTextField m_editor;
    
    public MemComboAgent(JComboBox comboBox) {
        m_comboBox = comboBox;
        m_editor = (JTextField) comboBox.getEditor().getEditorComponent();
        m_editor.addKeyListener(this);
    }
    
    public void keyReleased(KeyEvent e) {
        char ch = e.getKeyChar();
        if (ch == KeyEvent.CHAR_UNDEFINED || Character.isISOControl(ch))
            return;
        int pos = m_editor.getCaretPosition();
        String str = m_editor.getText();
        if (str.length() == 0)
            return;
        for (int k = 0; k < m_comboBox.getItemCount(); k++) {
            String item = m_comboBox.getItemAt(k).toString();
            if (item.startsWith(str)) {
                m_editor.setText(item);
                m_editor.setCaretPosition(item.length());
                m_editor.moveCaretPosition(pos);
                break;
            }
        }
    }
}

class MemComboBox extends JComboBox {
    public static final int MAX_MEM_LEN = 30;
    
    public MemComboBox() {
        super();
    }
}
APPENDIX C. SAMPLE PROGRAMS USED FOR TESTING

```java
130   setEditable(true);
131 }
132
133 public void add(String item) {
134   removeItem(item);
135   insertItemAt(item, 0);
136   setSelectedItem(item);
137   if (getItemCount() > MAX_MEM_LEN)
138     removeItemAt(getItemCount() - 1);
139 }
140
141 public void load(String fName) {
142   try {
143     if (getItemCount() > 0)
144       removeAllItems();
145     File f = new File(fName);
146     if (!f.exists())
147       return;
148     FileInputStream fStream = new FileInputStream(f);
149     ObjectInput stream = new ObjectInputStream(fStream);
150     Object obj = stream.readObject();
151     if (obj instanceof ComboBoxModel)
152       setModel((ComboBoxModel) obj);
153     stream.close();
154     fStream.close();
155   } catch (Exception e) {
156     System.err.println("Serialization error:");
157   }
158 }
159
160 public void save(String fName) {
161   try {
162     FileOutputStream fStream = new FileOutputStream(fName);
163     ObjectOutput stream = new ObjectOutputStream(fStream);
164     stream.writeObject(getModel());
165     stream.flush();
166     stream.close();
167     fStream.close();
168   } catch (Exception e) {
169     System.err.println("Serialization error:");
170   }
171 }
172```

C.2 Statement reordering

```java
public class Browser extends JFrame {

  protected MemComboBox locatorMemBox = new MemComboBox();
  protected JEditorPane browserEditorPane;

  public Browser() {
    super("HTML Browser");

    BrowserListener listener = new BrowserListener();

    locatorMemBox.load("addresses.dat");
    locatorMemBox.addActionListener(listener);

    MemComboAgent memComboBoxAgent = new MemComboAgent(locatorMemBox);

    browserEditorPane = new JEditorPane();
    JPanel panel1 = new JPanel();

    browserEditorPane.setEditable(false);
    browserEditorPane.addHyperlinkListener(listener);

    panel1.setLayout(new BoxLayout(panel1, BoxLayout.X_AXIS));
    panel1.add(new JLabel("Address"));
    panel1.add(locatorMemBox);

    setSize(500, 300);
    getContentPane().setLayout(new BorderLayout());
    getContentPane().add(panel1, BorderLayout.NORTH);

    JScrollPane scrollPane = new JScrollPane();
    scrollPane.getViewport().add(browserEditorPane);
    getContentPane().add(scrollPane, BorderLayout.CENTER);

    WindowListener wndCloser = new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        locatorMemBox.save("addresses.dat");
        System.exit(0);
      }
    };

    addWindowListener(wndCloser);
    setVisible(true);
    locatorMemBox.grabFocus();
  }
}
```
class BrowserListener implements ActionListener, HyperlinkListener {

    public void actionPerformed(ActionEvent evt) {
        String stringURL = (String) locatorMemBox.getSelectedItem();
        if (stringURL == null || stringURL.length() == 0)
            return;

        BrowserLoader loader = new BrowserLoader(stringURL);
        loader.start();
    }

    public void hyperlinkUpdate(HyperlinkEvent e) {
        URL url = e.getURL();
        if (url == null)
            return;

        BrowserLoader loader = new BrowserLoader(url.toString());
        loader.start();
    }
}

class BrowserLoader extends Thread {

    protected String m_sUrl;

    public BrowserLoader(String stringURL) {
        m_sUrl = stringURL;
    }

    public void run() {
        try {
            URL source = new URL(m_sUrl);
            browserEditorPane.setPage(source);
            locatorMemBox.add(m_sUrl);
        }
        catch (Exception e) {
            JOptionPane.showMessageDialog(Browser.this, 'Error: Ου'
        }
    }
APPENDIX C. SAMPLE PROGRAMS USED FOR TESTING 90

```java
public static void main(String argv[]) {
    new Browser();
}

class MemComboAgent extends KeyAdapter {
    protected JComboBox comboBox;
    protected JTextField editorTextField;
    public MemComboAgent(JComboBox comboBox) {
        comboBox = comboBox;
        editorTextField = (JTextField) comboBox.getEditor().getEditorComponent();
        editorTextField.addKeyListener(this);
    }
    public void keyReleased(KeyEvent e) {
        char ch = e.getKeyChar();
        if (ch == KeyEvent.CHAR_UNDEFINED || Character.isISOControl(ch))
            return;
        int pos = editorTextField.getCaretPosition();
        String str = editorTextField.getText();
        if (str.length() == 0)
            return;
        for (int k = 0; k < comboBox.getItemCount(); k++) {
            String item = comboBox.getItemAt(k).toString();
            if (item.startsWith(str)) {
                editorTextField.setText(item);
                editorTextField.setCaretPosition(item.length());
            }
        }
    }
}
```
editorTextField.moveCaretPosition(pos);
break;
}
}
}
}
}
}
}
}

class MemComboBox extends JComboBox {

public static final int MAX_MEM_LEN = 30;

public MemComboBox() {
    super();
    setEditable(true);
}

public void add(String item) {
    removeItem(item);
    insertItemAt(item, 0);
    setSelectedItem(item);

    if (getItemCount() > MAX_MEM_LEN)
        removeItemAt(getItemCount() - 1);
}

public void load(String fName) {
    try {

        if (getItemCount() > 0)
            removeAllItems();

        File f = new File(fName);

        if (!f.exists())
            return;

        FileInputStream fileInpStream = new FileInputStream(f);
        ObjectInput objectOutStream = new ObjectInputStream(fileInpStream);

        Object obj = objectOutStream.readObject();

        if (obj instanceof ComboBoxModel)
            setModel((ComboBoxModel) obj);

        fileInpStream.close();
        objectOutStream.close();
    } catch (Exception e) {
        System.err.println('Serialization error: ' + e.toString());
    }
}
public void save(String fName) {
    try {
        FileOutputStream fileInpStream = new FileOutputStream(fName);
        ObjectOutput objectOutStream = new ObjectOutputStream(fileInpStream);
        objectOutStream.writeObject(getModel());
        fileInpStream.close();
        objectOutStream.flush();
        objectOutStream.close();
    } catch (Exception e) {
        System.err.println("Serialization error: "+e.toString());
    }
}

C.3 Blocks reordering and control structures changes

public class Browser extends JFrame {
    protected MemComboBox locatorMemBox = new MemComboBox();
    protected JEditorPane browserEditorPane;

    public Browser() {
        super("HTML Browser");
        BrowserListener listener = new BrowserListener();
        locatorMemBox.load("addresses.dat");
        locatorMemBox.addActionListener(listener);
        MemComboBoxAgent memComboBoxAgent = new MemComboBoxAgent(locatorMemBox);
        browserEditorPane = new JEditorPane();
        JPanel panel1 = new JPanel();
        browserEditorPane.setEditable(false);
        browserEditorPane.addHyperlinkListener(listener);
        panel1.setLayout(new BoxLayout(panel1, BoxLayout.X_AXIS));
        panel1.add(new JLabel("Address"));
        panel1.add(locatorMemBox);
    }
}
APPENDIX C. SAMPLE PROGRAMS USED FOR TESTING

```java
setSize(500, 300);
getContentPane().setLayout(new BorderLayout());
getContentPane().add(panel1, BorderLayout.NORTH);

JScrollPane scrollPane = new JScrollPane();
scrollPane.getViewport().add(browserEditorPane);
getContentPane().add(scrollPane, BorderLayout.CENTER);

WindowListener wndCloser = new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
        locatorMemBox.save("addresses.dat");
        System.exit(0);
    }
};
addWindowListener(wndCloser);
setVisible(true);
locatorMemBox.grabFocus();

public static void main(String argv[]) {
    new Browser();
}

class BrowserLoader extends Thread {

    protected String m_sUrl;

    public BrowserLoader(String stringURL) {
        m_sUrl = stringURL;
    }

    public void run() {
        setCursor(Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR));
        try {
            URL source = new URL(m_sUrl);
            browserEditorPane.setPage(source);
            locatorMemBox.add(m_sUrl);
        } catch (Exception e) {
            JOptionPane.showMessageDialog(Browser.this, "Error: 
                " + e.toString(), "Warning", 
                JOptionPane.WARNING_MESSAGE);
        }
    }
```
APPENDIX C. SAMPLE PROGRAMS USED FOR TESTING

```java
73 setCursor(Cursor.getDefaultCursor());
74 }
75 }
76 }
77
78 class BrowserListener implements ActionListener, HyperlinkListener {
79     public void actionPerformed(ActionEvent evt) {
80         String stringURL = (String) locatorMemBox.getSelectedItem();
81         if (stringURL == null || stringURL.length() == 0)
82             return;
83         BrowserLoader loader = new BrowserLoader(stringURL);
84         loader.start();
85     }
86
87     public void hyperlinkUpdate(HyperlinkEvent e) {
88         URL url = e.getURL();
89         if (url == null)
90             return;
91         BrowserLoader loader = new BrowserLoader(url.toString());
92         loader.start();
93     }
94 }
95
96 class MemComboBox extends JComboBox {
97     public static final int MAX_MEM_LEN = 30;
98 
99     public MemComboBox() {
100         super();
101         setEditable(true);
102     }
103
104     public void save(String fName) {
105         try {
106             FileOutputStream fileOutStream = new FileOutputStream(fName);
107         }
108     }
109 
110     public static final int MAX_MEM_LEN = 30;
111 
112     public MemComboBox() {
113         super();
114         setEditable(true);
115     }
116
117     public void save(String fName) {
118         try {
119             FileOutputStream fileOutStream = new
120             FileOutputStream(fName);
121         }
122     }
```
public void load(String fName) {
    try {
        if (getItemCount() > 0)
            removeAllItems();

        File f = new File(fName);
        if (!f.exists())
            return;

        FileInputStream fileInpStream = new FileInputStream(f);
        ObjectInput objectOutStream = new ObjectInputStream(fileInpStream);
        Object obj = objectOutStream.readObject();
        if (obj instanceof ComboBoxModel)
            setModel((ComboBoxModel) obj);

        fileInpStream.close();
        objectOutStream.close();
    } catch (Exception e) {
        System.err.println('Serialization error:' + e.toString());
    }
}

public void add(String item) {
    removeItem(item);
    insertItemAt(item, 0);
    setSelectedItem(item);

    if (getItemCount() > MAX_MEM_LEN)
        removeItemAt(getItemCount() - 1);
}
class MemComboAgent extends KeyAdapter {

protected JComboBox comboBox;
protected JTextField editorTextField;

public MemComboAgent(JComboBox comboBox) {
    comboBox = comboBox;
    editorTextField = (JTextField) comboBox.getEditor().getEditorComponent();
    editorTextField.addKeyListener(this);
}

public void keyReleased(KeyEvent e) {
    char ch = e.getKeyChar();
    if (ch == KeyEvent.CHAR_UNDEFINED || Character.isISOControl(ch))
        return;

    int pos = editorTextField.getCaretPosition();
    String str = editorTextField.getText();

    if (str.length() == 0)
        return;

    int k = 0;
    while (k < comboBox.getItemCount()) {
        String item = comboBox.getItemAt(k).toString();

        if (item.startsWith(str)) {
            editorTextField.setText(item);
            editorTextField.setCaretPosition(item.length());
            editorTextField.moveCaretPosition(pos);
            break;
        }

        k++;
    }
}
}

C.4 Deportation of code into procedures

public class Browser extends JFrame {

protected MemComboBox locatorMemBox = new MemComboBox();
protected JEditorPane browserEditorPane;

public Browser() {
    super("HTML_Browser");
    setSize(500, 300);
    getContentPane().setLayout(new BorderLayout);
    buildPanel();
    windowListener();
    setVisible(true);
    m_locator.grabFocus();
}

public void buildPanel() {
    JPanel p = new JPanel();
    p.setLayout(new BoxLayout(p, BoxLayout.X_AXIS));
    p.add(new JLabel("Address"));
    m_locator.load("addresses.dat");
    BrowserListener lst = new BrowserListener();
    m_locator.addActionListener(lst);
    MemComboAgent agent = new MemComboAgent(m_locator);
    p.add(m_locator);
    getContentPane().add(p, BorderLayout.NORTH);
    m_browser = new JEditorPane();
    m_browser.setEditable(false);
    m_browser.addHyperlinkListener(lst);
    JScrollPane sp = new JScrollPane();
    sp.getViewport().add(m_browser);
    getContentPane().add(sp, BorderLayout.CENTER);
}

public void windowListener() {
    WindowListener wndCloser = new WindowAdapter() {
        public void windowClosing(WindowEvent e) {
            m_locator.save("addresses.dat");
            System.exit(0);
        }
    };
    addWindowListener(wndCloser);
}

public static void main(String argv[]) {
    new Browser();
}
APPENDIX C. SAMPLE PROGRAMS USED FOR TESTING

```java
class BrowserLoader extends Thread {
    protected String m_sUrl;

    public BrowserLoader(String stringURL) {
        m_sUrl = stringURL;
    }

    public void run() {
        setCursor(Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR));
        try {
            URL source = new URL(m_sUrl);
            browserEditorPane.setPage(source);
            locatorMemBox.add(m_sUrl);
        } catch (Exception e) {
            JOptionPane.showMessageDialog(Browser.this, "Error: \\
                " + e.toString(), "Warning", JOptionPane.WARNING_MESSAGE);
            setCursor(Cursor.getPredefinedCursor(Cursor.DEFAULT_CURSOR));
        }
    }

    class BrowserListener implements ActionListener,
            HyperlinkListener {
        public void actionPerformed(ActionEvent evt) {
            String stringURL = (String) locatorMemBox.getSelectedItem();
            if (stringURL == null || stringURL.length() == 0)
                return;
            BrowserLoader loader = new BrowserLoader(stringURL);
            loader.start();
        }

        public void hyperlinkUpdate(HyperlinkEvent e) {
```
APPENDIX C. SAMPLE PROGRAMS USED FOR TESTING

```java
URL url = e.getURL();

if (url == null)
    return;

BrowserLoader loader = new BrowserLoader(url.toString());
loader.start();
```

class MemComboBox extends JComboBox {

    public static final int MAX_MEM_LEN = 30;

    public MemComboBox() {
        super();
        setEditable(true);
    }

    public void save(String fName) {
        try {
            FileOutputStream fileInpStream = new FileOutputStream(fName);
            ObjectOutputStream objectOutStream = new ObjectOutputStream(fileInpStream);

            objectOutStream.writeObject(getModel());
            fileInpStream.close();
            objectOutStream.flush();
            objectOutStream.close();
        }
        catch (Exception e) {
            System.err.println('Serialization error: ' + e.toString());
        }
    }

    public void load(String fName) {
        try {
            if (getItemCount() > 0)
                removeAllItems();

            File f = new File(fName);
            if (!f.exists())
                return;
        }
```
APPENDIX C. SAMPLE PROGRAMS USED FOR TESTING

```java
FileInputStream fileInpStream = new FileInputStream(f);
ObjectInput objectOutStream = new ObjectInputStream(fileInpStream);

Object obj = objectOutStream.readObject();

if (obj instanceof ComboBoxModel)
    setModel((ComboBoxModel) obj);

fileInpStream.close();
objectOutStream.close();

} catch (Exception e) {
    System.err.println("Serialization error: "+e.toString());
}

public void add(String item) {
    removeItem(item);
    insertItemAt(item, 0);
    setSelectedItem(item);

    if (getItemCount() > MAX_MEM_LEN)
        removeItemAt(getItemCount() - 1);
}

class MemComboAgent extends KeyAdapter {

    protected JComboBox comboBox;
    protected JTextField editorTextField;

    public MemComboAgent(JComboBox comboBox) {
        comboBox = comboBox;
        editorTextField = (JTextField) comboBox.getEditor().getEditorComponent();
        editorTextField.addKeyListener(this);
    }

    public void keyReleased(KeyEvent e) {
        char ch = e.getKeyChar();
        if (ch == KeyEvent.CHAR_UNDEFINED || Character.isISOControl(ch))
            return;

        int pos = editorTextField.getCaretPosition();
        String str = editorTextField.getText();

        if (str.length() == 0)
return;

int k = 0;
while (k < comboBox.getItemCount()) {
    String item = comboBox.getItemAt(k).toString();
    if (item.startsWith(str)) {
        editorTextField.setText(item);
        editorTextField.setCaretPosition(item.length);
        editorTextField.moveCaretPosition(pos);
        break;
    }
    k++;
}
}